Adobe® Central Output Server Migration Guide

A technical guide for migrating to Adobe LiveCycle® Output ES

4Point

Version 1.1 – October 24th, 2007

Table of Contents

1. Introduction 1
    - Intended Audience 1
    - Goals and Scope 1
    - Organization of this Document 2
2. Product and Technology Overview 3
    - Why Move to LiveCycle ES? 3
    - Form Design 5
    - Data Integration 5
    - Terminology 5
    - Processing 6
    - Comparison Summary 7
    - Related Documentation 9
3. Forms 10
    - Importing Output Designer Forms 10
        - Import Goals and Constraints 10
        - Form Objects 11
        - Grouped Objects 12
        - Foundation Pages 12
        - Fonts 13
        - Subforms 13
        - Preamble Handling 14
        - Data Transformation Rules 14
        - Working with Imported Forms 16
    - Basic Forms 18
    - Multi-Part Forms 19
    - Dynamic Forms 19
        - Master Pages 19
        - Preambles 20
        - Expandable Objects 21
        - Field Overflow 22
        - Tables 23
        - Floating Fields 23
        - Calculations 23
        - Fragments 24
        - Form Variables 25
        - Page Counts 25
    - Locale Settings 26
4. Data 28
    - Data Formats 28
        - Central Transformation Agent 28
        - Moving to XML Data 28
        - DAT is flat, XML is structured 29
        - XML Data is Case-Sensitive 30
        - Formatting Rich-Text Data 31
        - Embedded Field References 32
        - Unicode 33
    - Binding Data to Forms 34

Global Fields and Global Data ... 34
Form-Driven Data Binding ... 36
Data-Driven Data Binding ... 36
Multi-Record Data ... 36
Modifying Form Objects from Data ... 37
5. Document Generation ... 39
Agents and Services ... 39
Invoking LiveCycle Output ... 39
Identifying the Form ... 39
Document Generation and Print Options ... 40
Testing and Previewing ... 40
Device Profiles ... 41
Font Handling ... 42
Font Availability and Mapping ... 42
Font Embedding ... 43
Paper Handling ... 43
Duplexing ... 43
Tray Handling ... 44
Faxing ... 45
6. Web Output Pak ... 46
XPR and Transaction Processing ... 46
Agents ... 46
7. Hosting Environment ... 47
8. Field-Nominated Commands ... 48
^continue ... 48
^copies ... 48
^currency ... 48
^data ... 48
^define ... 48
^duplex ... 49
^eject ... 49
^field ... 49
^file ... 49
^form ... 49
^global ... 50
^graph ... 50
^group ... 50
^inlinegraphbegin, ^inlinegraphend ... 50
^key ... 50
^macro ... 51
^page ... 51
^passthru ... 51
^popsymbolset ... 51
^pushsymbolset ... 51
^record ... 51
^symbolset ... 51
^shell ... 52
^subform ... 52
^tab ... 52
^trayin, ^trayout ... 52
^$page ... 52

About the contributing authors

4Point (Four Point Solutions Ltd.) is a leading Adobe Solution Partner in North America. Drawing from extensive experience in planning, implementing, and supporting intelligent document solutions and rich Internet applications (RIAs), 4Point helps enterprise customers extend the reach of their information, processes, and services across any medium. 4Point resells the complete line of Adobe enterprise products including Central Pro Output Server, LiveCycle, Adobe Flex™ and Acrobat® Connect, and provides customers with consulting, systems integration, and support services to keep their business running smoothly. Through numerous successful customer engagements and extensive knowledge of Adobe's enterprise products, 4Point has quickly become an industry reference when considering an Adobe LiveCycle, document output, or electronic forms solution.

4Point is dedicated to ensuring customers realize the full potential of the Adobe Engagement Platform vision. Headquartered in Ottawa, with offices in Washington D.C., Florida, Denver, Vancouver, and Halifax, 4Point is accessible, working closely with clients to ensure that current and future business goals are met. 4Point consultants and subject matter experts offer best practices for a range of industries, including financial services, government, manufacturing, life sciences, education, telecommunications, and publishing.

1. Introduction

Intended Audience

This document is intended for technologists who have experience building document generation and print solutions with the Adobe Central Output Server family of products, including Adobe Central Pro Output Server, Web Output Pak, and Output Designer. It will help individuals whose organizations are either considering, or beginning a migration to Adobe LiveCycle® Output ES by comparing and contrasting the capabilities of the two product lines, describing varying migration scenarios, identifying Central features that do not exist in LiveCycle Output, and describing some of the challenges and remedies.

It is assumed that readers have prior hands-on experience with the Central family of products and are proficient in their knowledge and understanding of these products. In particular, it is assumed that readers have an understanding of how forms (also known as templates) are designed using Adobe Output Designer, how dynamic forms are constructed using subforms, what the role of preambles is, and how the field-nominated data format utilized by Central is used to drive document generation and printing.

In terms of knowledge required to understand LiveCycle Output ES, it is assumed that readers have some familiarity with XML and related standards and tools, an awareness of the major features of LiveCycle Output, and some experience with LiveCycle Designer.

Chapter 2, *Product and Technology Overview*, provides an overview intended to help readers quickly understand basic differences between Central and LiveCycle Output; the remaining chapters provide significantly greater detail.

Goals and Scope

This document details how the features of the Adobe Central Output Server family of products compare to LiveCycle Output ES, often in a technical way. Assuming that readers have previous experience developing document generation and printing solutions with Central, this document seeks to leverage their existing experience and knowledge so they can quickly understand how equivalent solutions may be developed with LiveCycle Output.

Central and LiveCycle Output are intended to address the same core document generation and printing goals, but they differ in terms of their technological foundations, breadth of functionality, use of standards, and countless details. Nonetheless, the common heritage and goals of these products provides for a common basis of understanding and learning.

This document is not a tutorial on the overall LiveCycle Output product or any of its solution components, nor can it replace any of the product documentation or samples provided with LiveCycle Output. Instead, this document provides an introduction to many aspects of LiveCycle Output, and then refers readers to the LiveCycle Output product documentation for additional learning.

The document does not attempt to discuss Output Pak for SAP or Output Pak for Oracle EBS; while customers who hold licenses for these products qualify for a trade-up to LiveCycle Output ES, there is no SAP or Oracle EBS integration.

Organization of this Document

This document is divided into the following sections:

- Chapter 2, *Product and Technology Overview*, provides a high-level overview of the major aspects and solution components of the Central family of products and LiveCycle Output ES. This chapter offers a quick understanding of the basic differences between Central and LiveCycle Output.

- Chapter 3, *Forms*, provides a detailed examination of forms in both Central and LiveCycle Output from two perspectives. First, LiveCycle Designer (the form design product within LiveCycle Output) has the capability to import Central forms created with Output Designer. How this import capability works, the results it produces, and some of the areas that may require additional attention once the import is complete are discussed. Second, this chapter offers an overview of the major features of forms created with LiveCycle Designer and how those capabilities are similar to or different from forms created with Output Designer.

- Chapter 4, *Data*, provides information relevant to transitioning from the primary data format of Central (the field-nominated format) to the XML data capabilities available with forms created with LiveCycle Designer. First, it discusses XML as a data format, along with comparisons to the field-nominated format in Central. Then, it explains how LiveCycle Output combines XML data with forms to generate output.

- Chapter 5, *Document Generation*, explores important aspects of how LiveCycle Output and LiveCycle Designer can be used to generate documents and printed output. Topics of discussion include common methods of invoking LiveCycle Output, how to test and preview forms in LiveCycle Designer, how formats and print devices are handled and configured, and specific aspects of output generation such as font and paper handling.

- Chapter 6, *Web Output Pak*, offers a brief discussion of how Web Output Pak relates to LiveCycle Output.

- Chapter 7, *Hosting Environment*, provides a brief discussion of the different platform and hosting requirements of Central and LiveCycle ES.

- Chapter 8, *Field-Nominated Commands*, lists most of the commands that compose the field-nominated format in Central and directs readers to relevant resources for more information on how to accomplish similar tasks, such as the original field-nominated command.

2. Product and Technology Overview

Why Move to LiveCycle ES?

It is important to note that while this document seeks to explain the effort to migrate applications from the Central Output Server family to LiveCycle Output ES, Adobe acknowledges that migration represents work which requires planning and is often carried out over time as part of an overall application support lifecycle. It is with this understanding that Adobe is committed to the continued sale and support of the Central Output Server family so that clients have the flexibility they need to either maintain their existing installations or migrate to the LiveCycle ES platform on a schedule that meets their organizational requirements.

Current information regarding the Central family as well as support and migration options can be found on Adobe's website;

- www.adobe.com/products/server/outputserver
- www.adobe.com/support/programs/policies/policy_enterprise_lifecycle.html
- www.adobe.com/products/server/outputserver/move_lces.html

Adobe LiveCycle ES (Enterprise Suite) software is an integrated enterprise server solution you can use to extend the reach of your business processes to engage customers, partners, or suppliers. It consists of over a dozen individual solution components, including Adobe LiveCycle Output ES. It offers many benefits, including:

- An award-winning XML template design environment in LiveCycle Designer and the ability to manage form libraries through the use of reusable referenced fragments
- LiveCycle Workbench, an integrated process design environment
- An industry-leading interactive forms and electronic document processing solution
- Robust support for interactive, on-demand, or high-volume document output processes

LiveCycle ES takes advantage of industry-standard J2EE application servers and extends beyond document generation to build end-to-end processes that include interactive forms, process management, and document security. LiveCycle leverages the ubiquity of Adobe Reader®, Adobe Flash® Player, and web browsers — along with PDF and XML standards — to capture information from users, automate document processes, and integrate with existing enterprise infrastructure. Its robust architecture makes it ideally suited for organizations taking advantage of today's best-in-class enterprise architectures.

LiveCycle ES includes the following solution components:

Interactive Forms:

- LiveCycle Forms ES–Create and deploy interactive XML-based forms as HTML, PDF, or SWF.
- LiveCycle Data Services ES–Integrate RIAs with LiveCycle services, J2EE applications, and business logic.
- LiveCycle Reader Extensions ES–Fill in, sign, comment on, or save Adobe PDF files using only Adobe Reader software.
- LiveCycle Barcoded Forms ES–Automate the capture of form data using dynamic 2D barcodes.

Process Management:

- LiveCycle Process Management ES–Streamline human-centric business processes across your firewall.
- LiveCycle ES Connectors for ECM–Extend and return information from back-end and legacy systems.

Document Security:

- LiveCycle Rights Management ES–Manage usage rights to protect sensitive documents.
- LiveCycle Digital Signatures ES–Automate the validation of digital signatures in PDF documents.

Document Generation:

- LiveCycle Output ES–Dynamically generate personalized documents for processes that require on-demand or medium volume batch output support. Assemble PDF packages from existing files or pages. Convert PDF files to PostScript® or image file formats.
- LiveCycle Production Print ES*–Dynamically generate personalized documents for high-volume production environments. Support production print languages, such as AFP and IJPDS, as well as a wide range of post-processing requirements.
- LiveCycle PDF Generator ES–Automate the conversion of office and industry-standard file formats to PDF documents. Assemble PDF packages from existing files or pages. Convert PDF files to PostScript or image file formats.

* LiveCycle Production Print ES provides advanced functionality that is not offered in either the Central product family or LiveCycle Output ES and is therefore not a migration target for Central customers who wish to move to the LiveCycle environment. This guide focuses on the migration from Central Output Server family products to LiveCycle Output ES. While design time information relating to LiveCycle Designer ES would apply to both Output and Production Print, the runtime environments are substantially different. As a result, this guide does not attempt to address runtime information for Production Print.

Form Design

Central forms are designed using Output Designer, which offers a graphical environment for designing both the visual aspects of a form, and the behavioral characteristics of the form. Forms created with Output Designer are saved in an `.ifd` binary format, and deployed to Central in a `.mdf` binary format.

LiveCycle Output forms are also designed using the graphically rich environment provided by LiveCycle Designer. The set of tools and capabilities offered within LiveCycle Designer are significantly greater than Output Designer. In many cases, features and capabilities that required hand-coding of preambles or data in Central are now available as equivalent first-class features of LiveCycle Designer. Forms created with LiveCycle Designer are primarily saved in a `.xdp` format, which is an XML markup format, and then deployed to LiveCycle Output.

Chapter 3, *Forms*, provides a more detailed discussion of topics related to migrating and designing forms.

Data Integration

While Central has supported XML data as an input format for many years, the primary data format Central supports is known as the **field-nominated format**. Central also provides support for a number of legacy data formats and includes a software application for assisting with transforming other non-supported legacy formats to the field-nominated format: the Visual Transformation Editor.

Forms created with LiveCycle Designer can connect with data sources such as databases and web services. However, the primary supported data format is generic XML data that may optionally conform to a custom schema of your choosing. Aside from this emphasis on XML, there is no specific LiveCycle data format.

LiveCycle Designer provides a robust set of tools for integrating XML data with forms. There is no equivalent to the Visual Transformation Editor in the LiveCycle Output suite of solution components.

Chapter 4, *Data*, provides a more detailed discussion of topics related to moving from field-nominated data, to XML data.

Terminology

Output Designer is used to create *templates* and Central generates *documents*. Some users of Output Designer and Central may use the term *form* as an alternative to *template*, especially if their experience with Central or Output Designer originated in earlier versions of these products. Other people may strongly associate the word *form* with an interactive electronic form, such as an HTML or PDF form.

LiveCycle Designer is a product for designing forms regardless of whether the forms will be filled interactively, printed, or used to generate a document with LiveCycle Output. Therefore, LiveCycle Designer is always used to create *forms*. LiveCycle reserves the word *template* for predesigned forms that can be used as a basis for designing new forms. This document exclusively uses the term *form* according to the LiveCycle usage, even when referring to *forms* created with Output Designer (formerly *templates*).

There is one additional area of potential confusion around the use of the term *form*. One of the LiveCycle products is LiveCycle Forms (note the capitalization of Forms). This document focuses on LiveCycle Output, not LiveCycle Forms; therefore, references to LiveCycle Forms are rare. However, it is often important to distinguish between forms created with Output Designer, and forms created with LiveCycle Designer. Output Designer forms are often referred to as *Central forms*, whereas forms created with LiveCycle Designer are referred to as *LiveCycle forms* (note the lowercase *forms*).

Processing

Central is a processing engine that receives data, optionally combines it with a form, and invokes one or more software components (known as **agents**) according to process descriptions contained within a text file known as the **Job Management Database** (**JMD)**.

Central typically integrates with other software systems by monitoring file-system directories (known as **collector directories**). Central can also leverage agents as adapters between its collector directories and other means of communication, such as printer queues or e-mail.

Unlike Central, which is a native application designed to run on several platforms, LiveCycle Output is a Java™ enterprise application, also known as a J2EE application. As a J2EE application, LiveCycle Output must be run within a supported J2EE application server environment, installed on a supported platform. These distinguishing architectural features of LiveCycle Output provide a basis for significant scalability and integration opportunities. However, this approach also firmly defines LiveCycle Output as a server application not intended to be deployed to desktop systems, except for development and testing. For more information on this topic, see Chapter 7, *Hosting Environment*.

LiveCycle Output provides a solution component known as LiveCycle Process Management. The component is a processing engine intended to carry out tasks designed within the graphically rich process development tool known as LiveCycle Workbench. Processes are designed by assembling the various LiveCycle Output solution components for manipulating XML data, generating documents, sending output to printers, and more. Custom components can also be developed and incorporated into processes.

Chapter 5, *Document Generation*, provides a more detailed discussion of how generating documents with LiveCycle Output differs from Central.

Comparison Summary

The following table provides an additional comparison of Adobe Central Pro Output Server v5.6 and LiveCycle Output ES. Where noted, it compares capabilities delivered through related products such as Web Output Pak or LiveCycle Forms.

Feature or Capability	Central Pro Output Server	LiveCycle Output ES
Hosting environment	Native application with limited opportunities for scaling and load balancing	J2EE application that is scalable and can be clustered and managed by the application server
Operating system	Solaris AIX Linux Redhat/Suse Windows NT/2000/2003 HP-UX OS/400	Solaris (WebSphere or Weblogic), AIX (WebSphere), Linux Redhat/Suse (JBOSS, WebSphere or Weblogic) Windows 2003 (JBOSS, WebSphere or Weblogic) Not supported Not supported
Invocation	Command-line, collector directory, e-mail	Web service, Java API, e-mail, watched folders (collector directory)–All methods can invoke a LiveCycle API or Workbench defined process
Design environment	Output Designer	LiveCycle Designer Rich WYSIWYG modern GUI, drag-and-drop XML schema integration, declarative sub-form design, scripting models for calculations and logic, widow and orphan control, page n of m, nested tables, spell checking, and more
Data handling	XML, field-nominated data, comma-delimited, fixed-record Other formats accommodated with Visual Transformation Editor and Transformation Agent.	XML data, transformation between XML formats provided with support for XSLT Not supported

Feature or Capability	Central Pro Output Server	LiveCycle Output ES
Output formats	PDF documents PostScript PCL ZPL (Zebra Label) Other label formats Windows driver Fax drivers HTML (via Web Output Pak)	PDF documents, PDF/A (1a,1b) PostScript PCL (PCL 5e/PCl5c–not PCL XL) ZPL (Zebra Label), Not supported Not supported See Chapter 6, *Faxing* HTML (via LiveCycle Forms)
Ability to assemble multiple PDF documents	None	PDF Assembler service included • Combine new and existing PDF files or pages • Combine multiple documents into one single file, or create a document package with multiple documents • Automatically generate a hyperlinked table of contents • Add watermarks, backgrounds, or overlays • Add headers, footers, or numbering • Add, remove, rotate, and scale pages • Import, export, and manipulate attachments, annotations, links, and bookmarks in XML • Manipulate document metadata
PDF conversion formats	None	PostScript for server-based PDF printing to non-PDF printers Multipage TIFF to support archiving requirements Other image formats such as PNG and JPEG
Form management	Object library PDF preview, test print	Integrated repository, form and fragment library (fragments are referenced form objects) PDF preview, test print (Print Form With Data)
Process Design Environment	Central Pro Output Server JMD, Web Output Pak, XML Processing Rules (XPR)	Orchestration of LiveCycle solution components via LiveCycle Workbench

Related Documentation

Two major categories of additional documentation referenced in this document provide deeper levels of understanding: documentation related to the LiveCycle Output ES product and documentation related to the various standards leveraged by LiveCycle Output.

Additional product-related documentation includes:

- *XML Forms Architecture Specification, version 2.6*

  The technology underlying LiveCycle Designer, the forms it produces, and the manner in which these forms behave is largely a result of a family of XML markup languages known as the *XML Forms Architecture* (XFA). This markup language is defined by a formal specification document available from the Adobe website at: http://partners.adobe.com/public/developer/xml/index_arch.html.

  Note: At the time of initial publication of this document, version 2.5 of the XFA Specification was the latest available version.

Additional standards-related documentation includes:

- *PDF Reference, Sixth Edition, version 1.7*

  A formal specification of PDF is available from the Adobe website at: www.adobe.com/devnet/pdf/pdf_reference.html

- *Internationalization*

  The International Organization for Standardization (ISO) provides several standards associated with the accurate interchange of locale-specific information, such as identifying locales and languages (ISO 639-1, ISO 3166-1) and expressing currencies (ISO 4217) and dates and time (ISO 8601). These standards are used by XML, XFA, and solutions components of LiveCycle Output.

- *Unicode*

  Unicode is a standard for the encoding and interchange of characters and symbols used in the languages of the world. XML leverages Unicode for all content. More information on the Unicode standard can be found on the Unicode website at: www.unicode.org

- *XML 1.1*

  LiveCycle Output utilizes XML and XML-related technologies throughout. The formal specification of XML is available from the World Wide Web Consortium (W3C) website at: www.w3.org/TR/xml11

- *XSL Transformations (XSLT), version 2.0*

  XSLT is a markup language and a technology used to transform XML. Within the context of LiveCycle Output, XSLT is often referenced as a mechanism for transforming XML data into a format more suitable for use with a particular LiveCycle form.

3. Forms

This chapter will explore topics related to migrating from Central forms to LiveCycle forms, but does not attempt to be a tutorial or guide to the process of designing forms with LiveCycle Designer. The topics explored in this section include:

- How to directly import your existing Central forms, originally created with Output Designer, into LiveCycle Designer; and, a discussion of the goals and limitations of the import process. See the section called "*Importing Output Designer Forms*".

- Information related to designing basic forms, dynamic forms, and handling multi-part Central forms. See the section called "*Basic Forms*", the section called "Dynamic Forms", and the section called "Multi-Part Forms" respectively.

- A discussion of a capability with LiveCycle forms to create partial forms, known as fragments, that can be assembled into whole forms. See the section called "*Fragments*".

- The ability in LiveCycle forms to store arbitrary information in the form, in a manner equivalent to variables or **docvars** in forms created with Output Designer. See the section called "*Form Variables*".

- The mechanism in LiveCycle forms to produce the page count information (e.g. page 3 of 5) that commonly appears on generated documents. See the section called "*Page Counts*".

Importing Output Designer Forms

> **NOTE**: There is a known problem in the import capability of LC Designer 8.1 which affects importation of Output Designer forms with sub-forms.
>
> Upgrade to LC Designer 8.1 SP1 to resolve this problem; available here http://www.adobe.com/support/products/enterprise/support_knowledge_center_livecycle_designer.html

LiveCycle Designer has the capability to import forms originally created with Output Designer. The Output Designer form files, with `.ifd` file extensions, can be opened from the LiveCycle Designer standard Open dialog by selecting the Output Designer Form (`*.ifd`) display option from the Files of type drop-down list. Detailed instructions on the steps required to import an Output Designer form are described in the "Importing Adobe Output Designer Form Files" section of the LiveCycle Designer Help.

Although the LiveCycle Designer user-interface exposes the capability to import Output Designer forms via a **File > Open** operation, it is important to remember that Output Designer form files are imported into the LiveCycle format – in other words, LiveCycle Designer does not provide a capability to save forms in an Output Designer compatible `.ifd` format.

Import Goals and Constraints

In order to successfully import Output Designer forms into LiveCycle Designer, the following minimum requirements must be met:

- There must be a functioning installation of Output Designer present on the same system as LiveCycle Designer. The import process utilizes components of your existing Output Designer installation.

- Best results will be achieved when importing forms from recent versions of Output Designer: ideally versions 5.4 and later.

- If the Output Designer forms have dependent resources, such as image files, those resources must be present in the locations referenced by the Output Designer forms.

- The presentment target `.icf` configuration file required by the Output Designer should be available within the Output Designer installation.

The import capability does not import interactive form features (such as field validations) that may be present in form files created with older versions of Output Designer.

The overall intent of the LiveCycle Designer capability to import Output Designer forms is to jump-start the process of migrating existing forms to LiveCycle Designer, not to replicate the precise behavior of existing forms. However, LiveCycle Designer will attempt to migrate features of Output Designer forms, as follows:

- Form objects (e.g. text, graphics, fields, barcodes, etc.) are migrated to equivalent LiveCycle form objects.

- Grouped objects are migrated to subforms.

- Foundation pages (`JFMain` pages) are migrated to LiveCycle Designer master pages.

- Fonts used in the original forms are recognized, along with Output Designer font mappings. LiveCycle Designer will provide for additional font mapping during the import process.

- Subforms are migrated to LiveCycle Designer subforms, with consideration of the original subform types, relationships, and dependencies on foundation pages.

- Preamble information is processed, in order to assist in the migration of subforms.

- The set of fields contained within a subform produce LiveCycle data transformation rules intended to group together corresponding data. Depending on how the imported form will be combined with XML data, these transformations may need to be manually removed.

The following sections elaborate on the aforementioned aspects of the import capability.

Form Objects

LiveCycle Designer supports the same basic form objects as Output Designer: static content such as text, lines, boxes, circles, images, and barcodes; dynamic content such as text fields, radio-buttons, checkboxes, barcode fields, and image fields. Beyond these individual objects, there are equivalent features for groups, subforms, and other equivalent capabilities that are discussed in the following sections.

In general, LiveCycle Designer provides a richer set of form objects, with a wider range of properties and features, than Output Designer. However, the remainder of this section describes any notable issues with importing individual form objects from Output Designer forms into LiveCycle Designer.

Global Fields

It is not uncommon for Output Designer forms to have more than one field with the same name, possibly in different subforms, where some (but not all) of these fields are configured as global fields. Global fields in LiveCycle forms behave differently than global fields in an Output Designer form, as described in the section called "*Global Fields and Global Data*". It is not permissible for a LiveCycle form to have multiple same-named fields with varying global/non-global states within the same form – the fields must either be all global or non-global.

Tables

LiveCycle Designer does provide a capability to create table objects. However, table objects within an Output Designer form are not migrated to LiveCycle table objects. Instead, the fields that comprise the cells of the imported table are migrated to individual fields, and the graphical aspects of the imported table are migrated to a series of lines and boxes.

For more information on tables in LiveCycle forms, see the section called "*Tables*".

Graphics Formats

LiveCycle Designer will import logo objects (images) represented in the most common formats (e.g. `.tiff`, `.bmp`, `.gif`, `.png`, `.eps`, `.jpg`), but it does not support all Output Designer image formats (e.g. `.lgo`, `.pcx`, `.hgl`).

Grouped Objects

When importing an Output Designer form into LiveCycle Designer, any form objects that are grouped together are migrated to subforms in the resulting LiveCycle form. These subforms are given the same position, dimension, and name, as the corresponding groups from the Output Designer form.

In both Output Designer and LiveCycle Designer, grouping form objects together is often performed simply to make operating on these objects more convenient. LiveCycle Designer does have the notion of a group, distinct from the notion of a subform. Grouping a number of objects together in LiveCycle Designer does not automatically create a new subform, although both Output Designer and LiveCycle Designer permit you to select multiple objects and request that a new subform be created to enclose the select objects.

Regardless, the import processing of Output Designer forms considers groups to be significant form features, and migrates any groups to subforms in the resulting LiveCycle form.

Foundation Pages

Dynamic forms created with Output Designer require at least one full-sized page to be defined, known as a foundation page or `JFMain` page. Subforms within an Output Designer form can also

be associated with specific foundation pages. When Central generates output, it instantiates as many foundation pages as required to hold the range and variety of subforms.

Foundation pages also determine physical page attributes, such as page size and orientation. Common document features, such as page header and footer content, are often designed onto foundation pages rather than subforms.

Each foundation page of an Output Designer form is imported into LiveCycle Designer as a master page in the resulting LiveCycle form. Any associations between subforms and foundation pages are also migrated so that the subforms within the resulting LiveCycle form will be associated with the corresponding master pages.

Fonts

LiveCycle Designer will detect the fonts used in Output Designer forms, and can honor any font mapping defined for Output Designer with a jfontmap.ini file.

When importing an Output Designer form, for each font encountered that is not present on your system and is not already mapped via `jfontmap.ini`, LiveCycle Designer will present you with a dialog where you may select a substitute font. For a more detailed discussion of fonts and font mapping with LiveCycle forms, see the section called "*Font Handling*".

Output Designer, regardless of which presentment targets a particular form was designed for, provides a special font setting that can be used with any text field: the `*NOPRINT*` font setting. Any field that uses the `*NOPRINT*` font setting will, as the name implies, not print the contents of the field. LiveCycle Designer does not provide a `*NOPRINT*` font setting, but it does provide several different ways to conceal content. For instance, a field may be defined to only appear on-screen, only when printed, or be defined as hidden entirely (though the field remains visible within LiveCycle Designer it will not be visible on any generated output). This property of a LiveCycle form object is known as the **presence** property. Any fields within an Output Designer form that use a `*NOPRINT*` font setting are imported as fields with the presence property set to hidden.

Subforms

Dynamic forms created with Output Designer may contain subforms that represent header, trailer, or detail types of subform content.

Subforms are instantiated and arranged by Central according to rules and relationships defined within Output Designer, such as indicating the relationship between the header, detail subforms, and trailer, representing tabular information within a form, or a specific sequencing of subforms specified by indicating that each subform has a parent subform that must appear before it.

LiveCycle forms have equivalent subform capabilities, but they are not expressed in the same way as subforms created with Output Designer. For instance, within Central forms, subforms can be specified to occur in a specific sequence, but not by indicating that the preceding subform is a parent of, or encloses, the current subform – a parent/child relationship between subforms in LiveCycle forms expresses that one subform is actually contained within another subform. Unlike Central forms, LiveCycle forms may have subforms actually nested within other subforms. Therefore, when importing a Central form, LiveCycle Designer attempts to migrate the subform characteristics of the original form to an equivalent LiveCycle subform definition.

One example of an Output Designer subform characteristic for which there is no LiveCycle Designer equivalent, is the capability to declare that a subform must reserve an amount of space on the foundation page large enough to encompass itself and other subforms, plus an additional arbitrary amount of space. This mechanism permits a form to ensure that a subform will not be separated by a page break from a specified range of following subforms. LiveCycle forms provide equivalent functionality by different means, such as specifying that a subform must be kept on the same page as the following subform, or that a subform cannot be split across pages.

Preamble Handling

In both Output Designer and Central, dynamic subform behavior is largely dependent upon the instructions within a form's preamble. Output Designer automatically creates a number of different custom variables, containing preamble instructions, derived from the subforms designed within the form. These default preamble instructions, given that they are generated by Output Designer itself, are interpreted during the import to LiveCycle Designer. By considering the preamble, LiveCycle Designer will attempt to replicate the behaviors of the original subforms, thus producing a form that will generate either the same, or similar, output.

Beyond the automatically generated preamble, Output Designer also permits the creation of additional preamble instructions that may augment or override the default preamble. Preambles can also be specified as part of a Central job definition, or from within the data associated with a Central job. The ability of LiveCycle Designer to leverage preamble instructions decreases with custom preambles, and clearly preamble information that may have accompanied a Central job or data stream is unavailable to the LiveCycle Designer import process.

Data Transformation Rules

As Central merges field-nominated data with a dynamic form, it can automatically detect when a data item is encountered that does not correspond to a field within the current subform. When this occurs, processing continues on the next subform that does contain a field that matches the data item. In this way, the form can automatically be constructed from a set of subforms that are automatically assembled based upon the sequential processing of unstructured data, and recognizing when data is encountered that signals a transition to a new subform. LiveCycle forms provide a similar mechanism known as automatic data binding.

Based on an assumption that the LiveCycle form will be used with unstructured XML data that is equivalent to the Central form's original field-nominated data, the import process adds a number of data transformation rules into the resulting LiveCycle form. The transformation rules actually alter the structure of any data processed with the form. For more information on the range of possible data transformation rules that can be used within LiveCycle forms, see the section "Extended Mapping Rules" in the XFA Specification. Note that this transformation capability should not be confused with the generalized transformation capability (Transformation Agent) in Central; these transformation rules are strictly used to make minor alterations to XML data when it is processed by LiveCycle Output.

If the LiveCycle form will be used with unstructured XML data that is equivalent to the Central form's original field-nominated data, then these data transformation rules may be helpful. However, if the LiveCycle form will be used with structured XML data, or another data source, then these data transformation rules created by the import process will need to be adjusted or removed entirely. These data transformation rules are not exposed by the rich user-interface of LiveCycle

Designer, but can be modified or removed by selecting the LiveCycle Designer's **XML Source** tab that exposes the underlying XML markup of the form.

In the following example, after importing the `exsub3.ifd` sample form supplied with Central 5.6 (located in the `Samples/exprint` directory within the Central installation) into LiveCycle Designer, the following transformation rules are be found within the resulting form's XML markup, accessed by selecting the **XML Source** tab in LiveCycle Designer:

```
<transform ref="none">
  <presence>dissolveStructure</presence>
</transform>
<transform ref="PO_DATE REQNO VENDOR_CODE VENDOR">
  <groupParent>Header</groupParent>
</transform>
<transform ref="ITEM QUANTITY DESCRIPTION UNITS UNIT_PRICE TOTAL_PRICE">
  <groupParent>Detail</groupParent>
</transform>
<transform ref="SUB_TOTAL TAX TOTAL DEL_INSTRUCTN">
  <groupParent>Detail_Trailer</groupParent>
</transform>
<transform ref="REQNO">
  <groupParent>Req_Number</groupParent>
</transform>
```

For example, one of the above data transformation rules state that any set of XML elements `PO_DATE`, `REQNO`, `VENDOR_CODE`, and `VENDOR`, shall be enclosed within a new XML element named `Header`.

To further illustrate, the affect of this data transformation rule would transform the following XML data accordingly:

Original XML data before transformation:

```
<PO_DATE>2007-01-01</PO_DATE>
<REQNO>1</REQNO>
<VENDOR>Example Corp</VENDOR>
<VENDOR_CODE>EXC</VENDOR_CODE>
<REQNO>2</REQNO>
<PO_DATE>2007-01-02</PO_DATE>
```

XML data result from transformation:

```
<Header>
  <PO_DATE>2007-01-01</PO_DATE>
  <REQNO>1</REQNO>
  <VENDOR>Example Corp</VENDOR>
  <VENDOR_CODE>EXC</VENDOR_CODE>
</Header>
<Header>
  <REQNO>2</REQNO>
  <PO_DATE>2007-01-02</PO_DATE>
</Header>
```

Unless your XML data will be an unstructured equivalent to the Central form's original field-nominated data, these transformation rules produced by the import process will impede your ability to integrate the form with your XML data. In addition, because the data transformation rules are only accessible via the XML Source tab in LiveCycle Designer and not supported by it's rich user-interface, these data transformation rules will be hard to maintain, and possibly forgotten between form maintenance cycles. Learning to use the data binding features of LiveCycle Designer is recommended over utilizing the data transformation rules created by the import process. To

remove these transformation rules, select the XML Source tab in LiveCycle Designer and delete the markup directly.

Working with Imported Forms

As described earlier, the LiveCycle Designer import process operates according to a number of assumptions. The import process strives to produce a form that, when combined with unstructured XML data that is equivalent to the original field-nominated data, will generate a document very similar to the original Output Designer form.

Consider the following two examples intended to illustrate what is meant by unstructured XML data, versus structured XML data:

Example of unstructured XML data

```
<data>
  <vendor_code>1001</vendor_code>
  <vendor_name>A1 Business Products</vendor_name>
  <vendor_address>234 Second St., Anytown, ST</vendor_address>
  <billto_name>John Doe</billto_name>
  <billto_address>15 Fourth St., Anytown, ST<billto_address>
</data>
```

Example of structured XML data

```
<data>
  <vendor>
    <code>1001</code>
    <name>A1 Business Products</name>
    <address>234 Second St., Anytown, ST</address>
  </vendor>
  <billto>
    <name>John Doe</name>
    <address>15 Fourth St., Anytown, ST<address>
  </billto>
</data>
```

The example unstructured XML data represents data in a manner very similar to the Central field-nominated data format, with uniquely named data items. Such an equivalent field-nominated data file might look like the following:

Example field-nominated data

```
^field vendor_code
1001
^field vendor_name
A1 Business Products
^field vendor_address
234 Second St.
Anytown, ST
```

In contrast, the structured XML is able to reuse the names of data items that represent the same type of information (e.g. a name, an address) and use structural information (the `vendor` and `billto` elements) to distinguish between information about the vendor versus the buyer. The data transformation rules created by the import process instruct LiveCycle Output to automatically add the additional structural elements when processing the data.

One consequence of import process goal to create a form that will work with unstructured XML data, is that a number of data transformation rules are embedded within the resulting form.

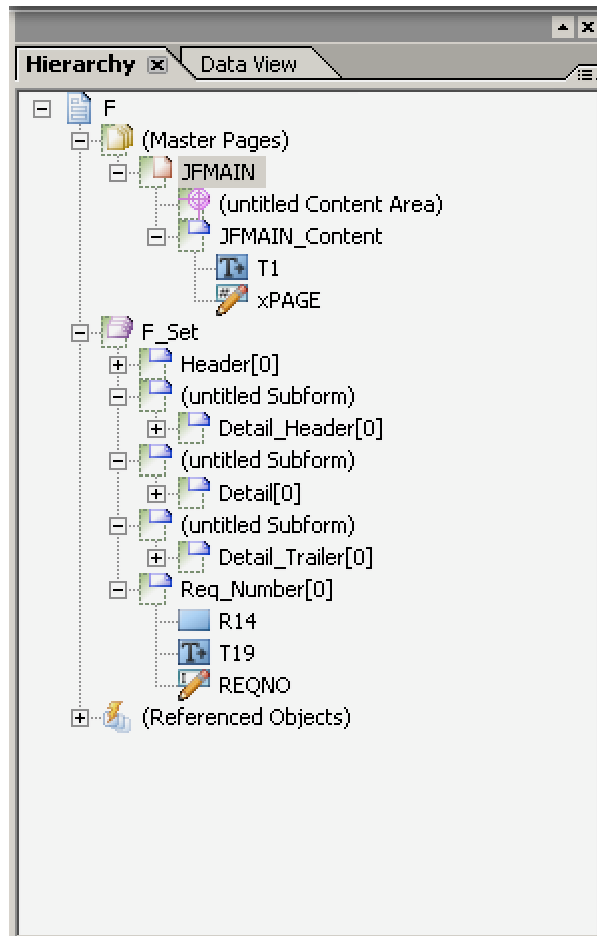
Another consequence is that the import process creates a hierarchy of subforms intended to replicate the behavior of the original form.

However, another potential use of the import capability is to simply avoid redesigning the content of the original form, without attempting to strictly replicate the original Central behavior of the form. When migrating from Central to LiveCycle, with its requirement for XML data, it is likely that the aforementioned assumptions of the import process are not applicable. For instance, migrating to LiveCycle may be an opportunity (or requirement) for changing the content and structure of the XML data that will be processed with the forms. Rather than handling data in a manner similar to Central, where data is automatically merged with the form according to matching names of data items with named fields, it may be more appropriate to utilize the powerful data-binding mechanisms available within LiveCycle Designer (see the section called "*Binding Data to Forms*").

In order to undo the artifacts of the import process assumptions, data transformation rules may need to be adjusted or removed (as described in the section called "*Data Transformation Rules*"), and the additional subform structures created by the import process may need to be adjusted or removed, as described below.

As previously described (see the section called "*Subforms*"), LiveCycle forms generated by the import process will contain subforms that represent the original subforms. In addition to basic subforms, LiveCycle forms also permit subforms to nest (within other subforms) in order to achieve a number of useful layout effects, or to appropriately represent the corresponding structure within an XML data source or schema. LiveCycle forms also provide a new concept, the **subform-set**, that is exclusively designed to describe that a number of subforms have a common relationship that will impact the layout and generation of a document. For example, a subform-set could be configured to ensure that a set of subforms must appear in a specific order, no specific order, or up to a maximum number of instances. As another example, a subform-set might indicate that, based upon the data requirements, one or more out of a set of possible subforms should be instantiated. Within LiveCycle Designer, this hierarchy of subforms and subform-sets can be observed and manipulated in the Hierarchy palette within the LiveCycle Designer workspace.

In the following example, after importing the `exsub3.ifd` sample form supplied with Central 5.6 (located in the `Samples/exprint` directory within the Central installation) into LiveCycle Designer, the following hierarchy of subforms and subform sets can be seen in the Hierarchy palette:



In order to undo the subform and subform set artifacts of the import process, and reduce the form to its basic collection of subforms and form objects, simply select the anonymous subforms in the hierarchy palette (subforms without a name, shown as "`(untitled Subform)`") and select the **Unwrap Subform** command via the palette menu, or via the context menu obtained by right-clicking on the subform icon in the palette. To remove the subform set, named `F_Set` by the import process, use the **Unwrap Subform Set** command via the palette menu or context menu.

Basic Forms

Output Designer and Central process two largely distinct types of forms: **static forms**, and **dynamic forms**.

Static forms generally aren't comprised of many subforms – all of the fields and other form objects are placed in specific locations across as many pages as required to hold the potential maximum range of data. Regardless of how much, or how little, data is combined with the form, the form will look largely the same. In this way, static forms are equivalent to manual paper forms.

Dynamic forms are all about supporting a range of potential document content and layout. During the process of designing a dynamic form, the form is decomposed into its component regions of content: the subforms. Depending on the requirements of the data, a unique document will be generated that satisfies the requirements of that particular range of data. Because the content

within a dynamic form is elastic, and the number and types of pages are not static, Output Designer requires that the underlying physical page requirements of the form be determined via a mechanism known as **foundation pages** (also known as **master pages** in LiveCycle Designer).

LiveCycle forms do not significantly differentiate between static and dynamic forms. All forms require master pages, and all forms make use of subforms. What determines whether a LiveCycle form behaves like a static form, or a dynamic form, is how the individual subforms are defined – whether the subforms are located at fixed positions and not configured to conditionally appear based on a relationship with a data source, or whether the subforms are configured to appear based on the requirements of the data and flow into a location on a page determined at the time of output generation.

LiveCycle does expose the notion of a static versus dynamic form in relationship to creating PDF forms for Adobe Acrobat. When designing a form for use in Adobe Acrobat, LiveCycle Designer permits you to state whether the form should behave as a static or dynamic form. However, when designing forms for use with LiveCycle Output, the distinction between static and dynamic forms is largely irrelevant.

Thankfully, the lack of this distinction does not raise the level of effort required to design simpler forms in LiveCycle Designer. Simple forms can be designed in LiveCycle Designer without being encumbered by features required only when designing more advanced, more dynamic, forms.

Multi-Part Forms

Output Designer provides a capability to design forms that replicate multi-part forms that are printed in impact printers or filled manually. The form has a number of layers, or parts, that may each display or hide a portion of the data. There is one set of data, and that data appears in varying degrees, on all parts of the form.

By default, a new form created with Output Designer, is a single-part form. A form may be defined as either a multi-part form, or multi-part sorted form, by selecting **Format > Template Design > Template Properties**, and choosing the desired option in the **Collate** area of the dialog box.

LiveCycle does not provide a multi-part form capability equivalent to the behavior of Central. However, it is certainly possible to create a LiveCycle form that has multiple pages, and utilizes data binding or scripting to replicate the data across the pages.

Dynamic Forms

As described in the section called "*Basic Forms*", LiveCycle Designer does not significantly distinguish between static forms, and dynamic forms. Hence, while the form capabilities described in the following sections are enclosed within a major section entitled "*Dynamic Forms*", these capabilities can also be utilized in simpler forms that don't appear to exhibit the characteristics commonly associated with dynamic forms.

Master Pages

Master pages are a feature of LiveCycle forms similar to foundation pages in Output Designer (as briefly described in the section called "*Foundation Pages*").

In Output Designer, foundation pages are only necessary when working with dynamic forms, and they can only contain static content (no fields). In LiveCycle Designer, all forms have at least one master page, and master pages are not significantly restricted in the type of objects they may contain. Master pages may contain fields, and even subforms. However, this capability for placing objects on master pages should not be abused; even when designing simple static forms, resist the temptation to design the form within master pages, and only use master pages for information commonly found in document headers and footers.

Common form requirements such as a page header or footer that contains calculated data (e.g. today's date, page number) or information retrieved from a data source are satisfied by placing fields on master pages. For more information on page numbering, see the section called "*Page Counts*".

Similar to foundation pages, master pages also determine physical page attributes, such as page size and orientation.

The content of a LiveCycle form is placed onto the master page at a position, and within a region, defined by an object known as a content area. By varying the position and size of a content area, the content of the form may be moved and constrained. Master pages may have more than one content area, allowing the content of a form to flow into more than one region on the page, similar to multi-column layouts.

Master pages may be defined as automatically repeating in order to completely enclose all of the form content for a given document, or may be defined to occur a specific number of times. Depending on this definition, when a master page is full, it will automatically repeat, or the remaining form content will resume on the next available master page (assuming multiple master pages were designed). Form objects may be split across page boundaries, or may be configured as unbreakable.

Master pages may be grouped into a series of master pages with a particular ordering, known as a **master page set**. These sets can also be grouped.

The range of capabilities present in LiveCycle master pages is significantly greater than foundation pages in Output Designer and Central. For more information on master pages, and how to use them, consult the LiveCycle Designer product documentation.

Preambles

As described in the section called "*Preamble Handling*", both Output Designer and Central, achieve dynamic subform behavior by leveraging the instructions within a form's preamble. Output Designer automatically creates a number of different custom variables, containing preamble instructions, derived from the subforms designed within the form.

LiveCycle forms do not provide a mechanism similar to preambles. Instead, LiveCycle forms provide a rich set of capabilities for achieving the same goals. The following list summarizes some of the common tasks performed by preambles, and the equivalent mechanism in LiveCycle forms:

- Preambles detect references to fields that are not present in the current subform and switch to the appropriate subform containing the requested field. This behavior is a consequence of the limitation inherent to Central forms where only one subform is active at any given moment and data must fully populate a subform before moving to the next

subform. LiveCycle forms do not have these limitations, and also benefit from robust data-binding mechanisms. LiveCycle forms are capable of automatically detecting when a different subform must be instantiated, or a new instance of a subform, without restricting access to other parts of the form.

- Preambles express how newly instantiated subforms are positioned relative to previous subforms. LiveCycle forms permit subforms to be positioned in a specific location within an enclosing subform, or arranged in sequence, and flowing in a specific direction.

- Preambles describe how much space is potentially required on the current page, or must be reserved, for a subform and its potential subsequent subforms. LiveCycle forms provide a rich set of properties on subforms that determine whether a subform may be split across a page boundary, or whether it must be kept on the same page as the subsequent subform.

- Preambles determine the header, detail, and footer behaviors that are very common in forms. LiveCycle forms permit subforms to be nominated as leading or trailing a detail subform (thus the LiveCycle terminology leader and trailer instead of header and footer). In addition, LiveCycle forms provide a powerful table capability where subforms can be arranged in a tabular layout.

- Preambles provide a mechanism similar to scripting, with the ability to perform conditional logic, test conditions, and format data. LiveCycle forms expose a rich set of properties and events on form objects that can be augmented with scripts defined in either the **JavaScript** or **FormCalc** language.

Expandable Objects

Forms created with Output Designer may leverage a capability where form fields may expand vertically to accommodate a range of data that would not otherwise fit within the field. By specifying in Output Designer that a field can expand, any data that would otherwise overflow the field is continued onto as many instances of a similar field in another subform as required. These additional subforms typically appear below the original subform containing the original expandable field enclosing the first line of data. A common use case for this capability is where a particular column within a table may contain free-form text associated with a line-item, or row, of the table. All of the descriptive text would appear within a visually taller table cell contained within the row, and the overall height of the row expands such that successive rows of content are properly located below.

The Output Designer approach works by repeating a particular subform for each line of content that overflows the original expandable field. However, while this approach works for the above simple and common scenario, it has two notable limitations. First, only one field representing a column within a subform representing a row can be configured as an expandable field, and the first field that begins to overflow produces the additional subforms. Once these additional subforms, representing the overflow area, are produced, Central's processing is unable to return to the original subform to handle additional expandable fields. Further, any additional data intended for the original subform that occurred after the overflowing data will be lost. Second, expandable fields can only expand vertically. There is no capability for fields to expand horizontally.

LiveCycle forms permit objects, such as fields and subforms, to be defined as expandable in either a vertical or horizontal dimension. LiveCycle Designer provides, via the Layout palette, the ability

to specify the minimum size of an object and whether the object should expand vertically or horizontally to accommodate its content, and in which directions the object should expand.

This ability for objects on LiveCycle forms to expand either vertically or horizontally is a first-class feature of the objects themselves; it does not require, unlike Output Designer, that the additional content be accommodated by fields and subforms designed to capture the overflowed content; and therefore, LiveCycle forms do not exhibit the limitations or side-effects of Central forms described above.

As an expandable object on a LiveCycle form increases in size, the decorative attributes of the object (such as the border) will also automatically adjust. Depending on whether the object has been anchored to a particular location on the page, or whether the object floats alongside the other objects in the form, the object's expansion may overlap other objects or cause the layout of subsequent objects to adjust accordingly. Eventually, an object may grow so large that it may need to be split across pages, and LiveCycle Designer allows the object to be configured so that its border will appear open or closed on either side of the page break. A subform may also be configured to permit, or disallow, the splitting of its content across pages.

Field Overflow

Related to the previous discussion of expandable objects (see the section called "*Expandable Objects*"), is the topic of how oversized content is handled within fields defined with a fixed size.

Consider the scenario where a single-line field, such as a field intended to represent a telephone number, is configured to hold only seven digits. When such a field is given more than seven digits of data, such as a telephone number that includes an area or country code, an overflow condition occurs.

In Central, by default, special handling is provided for single-line fields such that data is permitted to extend beyond the outer boundary of the field. In the case of the too-long telephone number, all of the digits would appear to extend beyond the edge of the field. In addition to this default behavior for single-line fields, any field object in a Central form may be configured, via preamble processing, with instructions on how to respond to an overflow event. Multiple-line fields may also take advantage of preamble handling of overflow conditions, but by default multiple-line fields will wrap content onto successive lines within the field until the field is filled with content. The handling of any remaining content is dependent upon the current **reformat** mode of Central (consult the Central documentation for more information on reformat processing).

LiveCycle forms do not exhibit different behavior for single-line fields versus multiple-line fields, except for the expected behavior that multiple-line fields will automatically wrap content onto successive lines. Fields are either designed with a fixed width and height, or may be designed to expand in width or height. A field with a fixed dimension will eventually truncate any content that does not fit, whereas an expanding field will extend to accommodate the content. Fields in LiveCycle forms do not provide a mechanism similar to the preamble-based overflow handling in forms created with Output Designer.

LiveCycle Designer also provides a type of field, the image field, intended to present an image rather than textual content. Image fields can be configured to scale the image to fit the dimensions of the field (optionally preserving the aspect ration of the image), or render the image according to its intrinsic dimensions.

Tables

Tabular layout of data is a very common feature of forms, and both Output Designer and LiveCycle Designer provide features for designing tables. However, often tabular layout is accomplished in Output Designer and Central by using a series of subforms, rather than a table object, often because subforms provide additional features (such as pagination) and allow the content of a table to vary by selecting from a variety of subforms.

Recognizing that tabular layout, for all but the simplest of forms, often requires functionality commonly associated with subforms, LiveCycle Designer exclusively utilizes subforms to produce tables, yet provides a rich user-interface suitable for both simple tables and complex tabular layout.

Tables in LiveCycle forms provide the common features of headers and footers, with control over the repeating of headers and footers across page breaks. The content of a table cell can be any form object, and may be bound to a data source with a varying number of rows dependent upon the range of data. Additional features of LiveCycle tables include:

- The number of rows can be fixed, or bound to a range of data.
- The content of a table cell can be any form object, including a subform, or a nested table.
- The table content can be subdivided into sections with independent headers, footers, etc.
- The rows or columns of the table can be configured to be automatically equally sized.
- The layout of a table can be configured to automatically break based upon a scripted condition.

For more information on designing tables in LiveCycle Designer, consult the LiveCycle Designer product documentation.

Floating Fields

Output Designer provides a feature known as **boilerplate fields** that permits the content of an otherwise static region of text to contain regions of variable content that will be populated from data, and the layout of the surrounding content automatically adjusts to accommodate the variable content.

LiveCycle forms provide an equivalent mechanism, known as **floating fields**, where a field can be inserted within the content of a region of text on a form, the field can be populated with data when the document is generated, and the layout of the surrounding text is adjusted. The underlying mechanism that LiveCycle forms use to accomplish this is to embed a specific unit of XML markup within the text content of the form object. This mechanism can also be used to reference and substitute values into the data that will be combined with a form (see also the section called "*Embedded Field References*").

Calculations

LiveCycle Designer is a full-featured form design application that can be used to create interactive forms, forms designed solely for generating output documents, or any combination in between.

Hence, features commonly associated with interactive forms, such as scripting and fields that contain calculated values, are available for use within document generation scenarios. When a form is processed, along with any associated data, any dependent scripting and calculations contained within the form are automatically executed.

In comparison, Central field-nominated data files and form preambles may use a simple command language with conditionals and common operations known as **calculation expressions**. These calculation expressions are very useful, but they are not a complete scripting solution. The LiveCycle approach leverages the existing scripting capabilities of LiveCycle forms, using either JavaScript or FormCalc scripting languages. It is worth noting that a set of functions is provided within the LiveCycle FormCalc scripting language that are very similar to the functions available to calculation expressions.

Fragments

The field-nominated format used by Central provides mechanisms for referencing multiple subforms from within a single data-stream. In this way, a document can be constructed by assembling parts of different forms together. The field-nominated `^subform` command permits the data file to call out to another subform within either the current, or a different, form file.

LiveCycle provides a similar mechanism known as fragments. Where Central provided this functionality by using a `^subform` command from within a field-nominated data file, LiveCycle provides the ability to create, manage, and reference reusable fragments from within the rich user-interface of LiveCycle Designer.

With LiveCycle Designer, any collection of form objects, including script objects, can be turned into a fragment, stored in a library of fragments or within a saved form file, and recalled from another form. Fragment libraries are storage locations, either on the local system or a shared network location, where fragments are stored. Multiple libraries can be created to aid in the organization, sharing, and reuse of fragments.

The LiveCycle Designer user-interface for fragment libraries is a palette equivalent to the Object Library palette. Fragments are available from the fragment library palette to be inserted into the current form.

Inserting a fragment into a form provides the same visual feedback and appearance as if the individual objects had been manually inserted into the form. However, when inserting a fragment, a reference to the storage location of the fragment within the fragment library is retained within the form. In this way, any changes made to a fragment will be automatically reflected when the referencing form is next opened in LiveCycle Designer, or processed by the LiveCycle Output server components.

Fragments in a form can also be disassociated from their original definition within a fragment library, allowing a fragment to be effectively copied into a form and isolated from any future changes that might be made to the original fragment in the library.

As briefly mentioned above, a fragment can be created within a form and simply stored as part of a form, rather than stored within a fragment library. While such a fragment will not be available from within the fragment library palette, LiveCycle Designer also provides a mechanism to insert form fragments located within other XDP form files. This capability is conceptually very similar to the Central `^subform` command that directly references a subform stored within another form file.

For more information on creating and using fragments, consult the LiveCycle Designer product documentation.

Form Variables

Forms created with Output Designer may have additional name/value pairs of information stored within the form. Output Designer facilitates the creation of these name/value pairs as **Custom Properties**, whereas Print Agent exposes them via the **DocVar** dictionary.

LiveCycle Designer ES also permits the creation of custom variables, associated with a form, by using the **Variables** tab of the **Form Properties** dialog. These variables can be accessed by referencing the named variable from within any scripts.

Page Counts

A common requirement of multi-page documents is the appearance of a running page count, along with the total page count, on the header or footer of a document. Often this will appear as "Page *n* of *m*" where *n* is the current page number and *m* is the total page count.

Obtaining the current page number is straightforward in Central, such as using the `^$page` or `\$page` field-nominated commands. Determining the total number of pages is more challenging, because the document must be first generated in order to determine the total number of pages, though this process can be automated with Central's job management database. With Central, the need to predetermine the total number of pages is due to the way that documents are constructed. In order to print a "Page *n* of *m*" on the first page of a document, the value of *m* must have already been determined because by the time the end of the document is encountered, it is too late to go back to the previous pages and subforms to populate the areas referencing the total number of pages.

LiveCycle takes the approach of constructing an entire document as one whole entity before committing the generated document to its output format or device. In this way, scripting and calculations have an opportunity to execute in the context of a fully constructed document, and may easily query the document for its total number of pages. No multi-step processing is required to determine the total number of pages.

Individual master pages in a LiveCycle form may indicate whether the page should be numbered, and contribute to the overall page count, or whether the master page should not be considered a numbered page. This is useful for scenarios where one or more pages, or possibly the back side of pages in a duplex document, are not intended to have their own page number. A master page may also be configured to start a new page count, or continue from an existing page count.

LiveCycle forms also distinguish between the total count of pages (based on the accumulation of numbered master pages) and the total number of surfaces representing the actual number of physical page surfaces generated.

LiveCycle forms retrieve information about the current page, the total number of pages, and the total number of surfaces by calling a built-in LiveCycle method as the calculated value for a field, or from a script expression. For instance, in order to obtain the total number of pages or surfaces, a script would call the `xfa.layout.pageCount()` method or the `xfa.layout.sheetCount()` method respectively. Obtaining the current numbered page or surface is handled by calling the `xfa.layout.page(this)` or `xfa.layout.sheet(this)` method respectively. These two methods

receive a reference of the calling object (`this`) to determine on which page or surface the calling object appears. Hence, it is also possible to determine on which page or surface a different object appears by passing in a reference to that object when calling these methods. This is useful for producing cross-references within a document. Note that when calling these methods, the first page is (by default) numbered `1` (one) whereas the first surface is numbered `0` (zero).

To ease the creation of "Page *n* of *m*"areas on a form, LiveCycle Designer provides a "Page *n* of *m*" object in the Object Library. A similar "Sheet *n* of *m*" object is also available from the Object Library that produces surface count information instead of numbered page count information.

LiveCycle forms also provides page and sheet values in the context of a document batch allowing for page counts to be determined outside of the context of a document and in the context of a complete batch. Some Central solutions mark document batches for use with enveloping / insertion machines. This is typically via a 2 pass process, capturing page information via preamble TRACE command in the first formatting pass, running a custom program to manipulate the batch data file based on the captured TRACE information and then doing a final format with enveloping / insertion marks on each page. These batch level values can be used to script enveloping / insertion solutions without the need for 2 formatting passes or a custom program. For more information on accessing these values see the LiveCycle Designer ES Scripting Reference.

Locale Settings

A document may need to be used within a particular environment with its own language requirements, conventions for representing dates, times, numbers, and monetary values. International standards exist for categorizing the world's various collections of geopolitical and linguistic expectations around representing and interpreting such data content; commonly these individual collections are known as locales.

LiveCycle forms leverage these standards and provide an easy way to build forms, and generate documents, that can adapt to a particular locale, or present information according to the expectations of multiple locales within the same document. For example, a document could correctly present its dates and numbers regardless of whether the document was generated in an English or French locale, or an American or British locale. A document might need to present its information simultaneously in two or more languages, or ensure that its content is always consistently presented according to its country and language of origin, regardless of the country where the document generation or viewing will occur. All of this is possible with the locale functionality of LiveCycle forms, and LiveCycle forms also permit their locale information to be customized for the creation of new locales, or modification of standard locales.

It is important to note that the XML standard provides a conceptually similar mechanism, known as `xml:lang`, for declaring that the content of a particular XML element is expressed in a particular language. This is similar, but distinct from LiveCycle's use of locale information, and LiveCycle forms do not process any `xml:lang` attribute encountered when processing an XML data source.

Within a LiveCycle form, most objects that have a capability to contain or format data content also expose a `locale` property that can be adjusted to one of the available locale definitions. The form itself also has its own overall default locale setting, which can be selected from the LiveCycle Designer Form Properties dialog. Because the content of a LiveCycle form can be comprised of a hierarchy of objects contained within multiple nested subforms, each object defaults to inheriting the locale setting of is enclosing subform, or the overall default locale for the form. An object, or

the form itself, may also choose to not constrain itself to a particular locale, and instead may choose to operate according to the locale of the system or application environment.

Central provides configuration settings for adjusting common locale features, including the grouping separator (thousands separator) or decimal point for numeric values, currency symbol, and calendar content (e.g. day names, month names). Although LiveCycle leverages a built-in standards-based configuration for a wide variety of locales, it is possible to add your own locale information, or adjust an existing locale. Custom or modified locale information can be incorporated within the XML definition of a form itself, according to an XML markup language that is documented as part of the Adobe XML Forms Architecture Specification.

Here is an excerpt from the locale definitions automatically embedded by LiveCycle Designer within a form designed for an American English locale, showing the definitions for the abbreviated day names, numeric punctuation, and currency symbols:

Excerpt of form locale information:

```
</calendarSymbols>
  <dayNames abbr="1">
    <day>Sun</day>
    <day>Mon</day>
    <day>Tue</day>
    <day>Wed</day>
    <day>Thu</day>
    <day>Fri</day>
    <day>Sat</day>
  </dayNames>
</calendarSymbols>
<numberSymbols>
  <numberSymbol name="decimal">.</numberSymbol>
  <numberSymbol name="grouping">,</numberSymbol>
  <numberSymbol name="percent">%</numberSymbol>
  <numberSymbol name="minus">-</numberSymbol>
  <numberSymbol name="zero">0</numberSymbol>
</numberSymbols>
<currencySymbols>
  <currencySymbol name="symbol">$</currencySymbol>
  <currencySymbol name="isoname">USD</currencySymbol>
  <currencySymbol name="decimal">.</currencySymbol>
</currencySymbols>
```

4. Data

Data Formats

Central is a product whose evolution began well before the **World Wide Web Consortium** (**W3C**) and the common use of XML as a data format. This history is reflected in the range of non-XML data formats that Central is capable of consuming, including comma-delimited and fixed-length record formats. However, the preferred, and most widely used, Central data format is known as the **field-nominated format**, and often referenced by its commonly used file extension as the `DAT` format.

In recent releases Central has provided support for XML data, by transforming XML into field-nominated format either automatically or explicitly via the XML-Import Agent. If you have prior experience using XML with Central, many aspects of that experience will apply to building solutions with LiveCycle. XML is the only data format support by LiveCycle Output, though any number of different XML-based data formats can be supported.

This section will explore what you need to know when moving from the range of data formats supported by Central, including the XML supported by Central, to the XML data consumed by LiveCycle Output.

Central Transformation Agent

Central provides a Transformation Agent, and Visual Transformation Editor, that facilitate the transformation of one data format into another, including data formats that may not be supported by Central.

LiveCycle Output exclusively consumes XML data, and does not provide an equivalent capability to transform non-XML data. LiveCycle Output does provide support for leveraging XML transformations expressed in the standard **XSL Transformations** (**XSLT**) language.

Moving to XML Data

It is valuable to first understand the fundamental similarities and differences between how Central consumes field-nominated data, compared to the XML data consumed by LiveCycle Output.

Both formats are textual, and annotate the data with markup to, at a minimum, provide each unit of data with a name that aids the process of merging the data into a form. A simple example of data in field-nominated format, and equivalent example1 in XML format, follows:

Very simple field-nominated data:

```
^field vendor_code
1001
^field vendor_name
A1 Business Products
^field vendor_address
234 Second St.
Anytown, ST
```

Very simple XML data:

```
<vendor_code>1001</vendor_code>
<vendor_name>A1 Business Products</vendor_name>
<vendor_address>234 Second St., Anytown, ST</vendor_address>
```

Beyond the syntactical differences of expressing markup in field-nominated format using the caret (^) symbol versus angle-brackets in XML, the two formats differ significantly on the meaning of their markup. The field-nominated statement `^field vendor_code` is actually a command instructing Central to apply the following data to a field known by the specified name of `vendor_code`. In contrast, the XML markup `<vendor_code>` only states that the following data shall be known as `vendor_code`.

Field-nominated format is actually a command language, not a data description language such as XML. The field-nominated format provides a range of instructions intended to be consumed by individual Central Agent processing modules. This command language typically encloses data intended to be merged with a form. This is more apparent when considering field-nominated statements such as `^page` or `^eject` that instruct Central to advance or eject a page, and clearly have nothing to do with describing data. In contrast, the XML data consumed by LiveCycle Output is almost exclusively constrained to the task of adequately describing the data, often according to a custom schema expressed in XML Schema format.

The capability to affect the document construction and rendering process at a low-level, with commands like `^page`, undoubtedly provide the creator of field-nominated data with a significant degree of intervention, from within the data, over the generated document. However, the use of such commands does increase the degree of coupling between a set of data and a form. It can also become a challenge in solutions where the data is provided by a third-party and there is concern over the fact that the data is driving the document generation process.

There is no equivalent, in the XML data consumed by LiveCycle Output, to the majority of field-nominated commands that are focused on affecting the document construction and rendering. For a discussion of field-nominated commands, and equivalent functionality in LiveCycle forms, see Chapter 8, "*Field-Nominated Commands*".

DAT is flat, XML is structured

The Field-nominated format expresses data in a linear manner, with limited mechanisms to describe structure or relationships that might exist within the data. As a consequence, in many cases the order of the statements within field-nominated data is significant, and different results may be produced depending on the order.

XML data is inherently hierarchical, and each unit of data, is enclosed within tags that indicate the beginning, and the end, of each data item as an XML element. Data may also be expressed as XML attributes . The structure, relationships, and ordering of data, implied by XML data is often explicitly defined by an accompanying **schema**. While a schema can make assertions about the relative ordering of the XML data elements, such order is often insignificant. Consider the following reworking of the XML example introduced in the previous section:

Structured XML:

```
<vendor>
  <code>1001</code>
  <name>A1 Business Products</name>
  <address>234 Second St., Anytown, ST</address>
</vendor>
<recipient>
  <address>678 Fourth Ave., Anytown, ST</address>
  <name>Tony Blue</name>
</recipient>
```

In the above example, element names such as `name`, and `address` appear in two different contexts, describing the name and address of the vendor and recipient. It is the presence of the enclosing `vendor` and `recipient` tags that allow the vendor's name and address to ultimately be distinguished from the recipient's name and address. An equivalent example in field-nominated format would likely employ `^field` commands, and possibly `^group` commands, with the fully-qualified names `vendor_name` and `recipient_name` in order to adequately differentiate between the data items.

It may appear at first that the common practice of repeating element names in different contexts should be discouraged, that it creates ambiguity. One common reason for this apparent re-use of element names often stems from the use of a schema where the benefit is the ability to define a data item name or address once, and then further define the contexts in which these data items can appear. As described before, it is the surrounding context within the XML that disambiguates these data items. XML data, when used in concert with LiveCycle Output's XML-based forms, utilizes these matching hierarchies of structural information, along with explicitly defined data-binding instructions within the form, to successfully merge the data into the form.

The sequence of name and address elements varies between the vendor and recipient to illustrate that the ordering of this information is not significant for the correct processing of this information.

XML Data is Case-Sensitive

One significant difference between using field-nominated data with Central and using XML data with LiveCycle forms is case-sensitivity. Central data formats and forms are not case-sensitive, whereas XML data and LiveCycle forms are case-sensitive. Case-sensitivity is simply the accepted practice with XML technologies.

Consider that Central would consider the following two field-nominated data files to be equivalent and process them in the same way:

Lowercase field-nominated data

```
^field vendor_code
1001
^field vendor_name
A1 Business Products
^field vendor_address
234 Second St.
Anytown, ST
```

Uppercase field-nominated data

```
^FIELD VENDOR_CODE
1001
^FIELD VENDOR_NAME
A1 Business Products
^FIELD VENDOR_ADDRESS
234 Second St.
Anytown, ST
```

In contrast, depending on whether a LiveCycle form was designed with lowercase field names or uppercase field names, only one of the following two XML data files with the matching letter-case would properly bind to the form.

Lowercase XML markup

```
<vendor>
  <code>1001</code>
  <name>A1 Business Products</name>
  <address>234 Second St., Anytown, ST</address>
</vendor>
```

Uppercase XML markup

```
<VENDOR>
  <CODE>1001</CODE>
  <NAME>A1 Business Products</NAME>
  <ADDRESS>234 Second St., Anytown, ST</ADDRESS>
</VENDOR>
```

The best way to avoid problems with the case-sensitive nature of XML and LiveCycle forms is to establish whether your XML data, schema, and forms will use lowercase or uppercase naming, and consistently maintain that practice.

Formatting Rich-Text Data

Field-nominated data can include commands that affect the formatting of the data within a form field. These commands, known as **Inline Text Control** commands, provide capabilities such as varying the appearance of the font, defining and advancing to tab stops, and more. Data incorporating such formatting commands, or markup, is commonly called **rich-text**.

Rich-text formatting with Inline Text commands

```
^inline on
^group order
^field instructions
Deliver the order to the \b.back\b0. door!
```

Within the Central support for XML data, rich-text is not expressed using Inline Text Control commands. Instead, rich-text is expressed with a subset of XHTML markup.

Rich-text formatting with Central XHTML

```
<order>
  <instructions
    xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
    xfa:contentType="text/html">
    <p>Deliver the order to the <b>back</b> door!</p>
  </instructions>
</order>
```

LiveCycle Output supports an extended range of XHTML-based rich-text markup, including support for a subset of CSS style attributes.

Rich-text formatting with LiveCycle Output XHTML

```
<order>
  <instructions>
    <body xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
          xmlns="http://www.w3.org/1999/xhtml">
      <p>Deliver the order to the <b>back</b> door!</p>
    </body>
  </instructions>
</order>
```

Embedded Field References

Building on the earlier discussions of floating fields and rich-text data (see the section called "*Floating Fields*" and the section called "*Formatting Rich-Text Data*" respectively), LiveCycle forms can also embed referenced content from within data, in a manner equivalent to variable substitutions in field-nominated format.

Field-nominated data can contain references, identified by the "`@`" (at-symbol) prefix, to the value of another field or previously defined variable. For instance, the following field-nominated data would populate the message field with the value "You saved 20 dollars":

```
^field saved_amount
20
^field message
You saved @saved_amount dollars
```

To accomplish an equivalent substitution from the data with a LiveCycle form, special XML markup must be placed within rich-text data. The following example XML data would produce the same result as the above field-nominated example:

```
<saved_amount>20</saved_amount>
<message>
  <body xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
        xmlns="http://www.w3.org/1999/xhtml">
    <p>You saved <span xfa:embed="saved_amount"/> dollars</p>
  </body>
</message>
```

Note that in both of the above examples, it is assumed that there is a `saved_amount` field on the form (possibly a hidden field).

An optional `xfa:embedMode` attribute on the embedded reference can determine whether or not the referenced value should be inserted with or without any of its original formatting. References can also be expressed as an XML id reference or URI by providing a `xfa:embedType` attribute with a value of `uri`.

One interesting capability of field-nominated variable substitutions is that they can be combined into compound references that are evaluated from right to left. In other words, given the following example, the value of the message field would be "You saved 30 dollars":

```
^field savings_tier_1
20
^field savings_tier_2
30
^field client_tier
2
^field message
You saved @savings_tier_@client_tier dollars
```

In the above example, there are two levels of potential savings that can be provided to our imaginary client, based upon the client's rating as a tier-one or tier-two client. Because the `client_tier` substitution is evaluated first, the whole substitution expression is evaluated to "`savings_tier_2`" and the value of the `savings_tier_2 field` is returned.

While this ability to combine variable substitutions provides a useful additional level of indirection, its importance in the context of the field nominated format stems in part from the fact that this feature was established prior to advent of calculation expressions, and there are fewer overall opportunities for scripting logic. By leveraging the capability to place calculations and scripting within a LiveCycle form, the same goals can be achieved. Consider the following example XML data:

```
<savings_tier_1>20</savings_tier_1>
<savings_tier_2>30</savings_tier_2>
<client_tier>2</client_tier>
<message>
  <body xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
        xmlns="http://www.w3.org/1999/xhtml">
    <p>You saved <span xfa:embed="saved_amount"/> dollars</p>
  </body>
</message>
```

In the above example, the same data is provided to populate the two different levels of savings and the particular level of savings that should be awarded to the client. However, the embedded reference is to a field named `saved_amount` for which no data is supplied. In order for this example to work, a field must be placed on the form with a calculated value that determines the savings value. The `saved_amount` field may have a JavaScript calculation such as the following:

```
if (client_tier == 1) {
  return savings_tier_1
} else if (client_tier == 2) {
  return savings_tier_2
} else {
  return 0
}
```

Unicode

The characters within field-nominated data are encoded according to the encoding scheme specified by one or more `^symbolset` commands within the data, or as part of options specified on the `^job` command. The range of characters (the **character set**) available at any particular point in the data is the range of characters that can be accessed by the encoding scheme.

Central supports a wide range of encoding schemes, including the UTF-8 Unicode encoding. Prior to the common use of Unicode, the `^symbolset` command was often invoked to access characters that were simply unavailable in any single character set. For instance, data that incorporated

characters from both European and Asian languages would utilize the `^symbol set` command when switching between these languages and the corresponding encoding schemes. The `^symbol set` command also supports Unicode encodings such as UTF-8, providing a means to address the range of Unicode characters without switching among encodings within the data.

XML supports only characters found within Unicode. Unlike the field-nominated format, the entire content of an XML resource may only be encoded in a single encoding scheme. However, this should not be a limitation given the vast range of addressable characters in Unicode. XML also provides a mechanism to address individual characters by either a symbolic or numeric reference.

The encoding scheme of an XML resource is indicated by the XML declaration, appearing at the beginning of the resource (e.g. `<?xml version="1.0" encoding="UTF-8"?>`)

Binding Data to Forms

In Central, the process of combining a range of data with a form, to generate a final document, is commonly known as **data merging**. The data is merged into the fields in a manner that evokes the classical operation of merging names and addresses with mailing labels in a word processor.

With LiveCycle Output the equivalent process of combining a range of data with a form is not known as merging; it is known as **data binding**. This difference in terminology speaks to the different nature of how XML data is bound to an XML-based form designed with LiveCycle Designer.

Within XML languages, and within HTML oriented technologies on the web, the notion of making connections between units of information for a variety of purposes is commonplace. The location of a unit of information within an XML document may be connected to another location within the same XML document or within another XML document. These connections are the bindings, and the connections between a form and data are data bindings.

The process of combining data with a form could be considered as either a process of pushing data items into the form, or the form pulling from the data. In one case the data file is controlling the process, and in the other case the form is controlling the process. This distinction is important because it determines whether the control is in the hands of the person responsible for developing the data, or the person responsible for designing the form.

Central exhibits a **data-driven** approach, where the commands in the data are in control over the generated document. A form, especially a form utilizing dynamic subforms, can be designed as loose collection of subforms that will be assembled in a sequence and populated with data according to the field-nominated commands encountered in the data.

LiveCycle Output operates primarily in a **form-driven** approach where the form contains the rules that determine how the final document will be generated. However, LiveCycle Output also permits the designer of a form to indicate where the form should cede a degree of control to the data.

This chapter will describe these mechanisms and how they differ from Central.

Global Fields and Global Data

Globals are a mechanism for the very common requirement of having a single data value appear in more than one field on a form. Global fields can be defined with the Output Designer, and global

data values can be defined from within a field-nominated data file or accompanying preamble. In either case the effect is the same: data that is recognized by Central as global is merged into the appropriate range of like-named global fields designed into the form.

Global data is handled differently when using XML data with Central. A special XML attribute, `xfa:match="many"`, is recognized on any element, and this instructs Print Agent to process the associated data value as a global data value within the scope of the current document.

When Central has finished processing the data for the current document, the global data values created with the `xfa:match="many"` attributes are forgotten, and will not be available to be merged into the successive documents generated from the same data value. Put another way, a global data value defined within the context of one document is not carried forward into the processing of successive documents. This behavior has been a challenging aspect of how Central's support for XML data is processed. The common requirement to define a global data value that is merged into a series of consecutive documents isn't accommodated by this behavior.

The behavior of global fields in LiveCycle forms is more flexible, providing a means to merge data into multiple fields within a form, as well as mechanism to merge global data across multiple documents generated from a single range of XML data.

The first method to merge a single data value into multiple fields of a LiveCycle form is to simply use LiveCycle Designer to bind fields to the same data value. By doing this, there is nothing special, nothing global, about either the fields or the associated data value; but, the effect is equivalent to common use of globals within Central. It is not required that the name of the data value matches the name of the field, nor that the multiple fields each have the same name. By binding the fields to a data value within LiveCycle Designer, the respective naming of the data value and corresponding fields becomes irrelevant.

The second method is to indicate with LiveCycle Designer that one or more fields, each with the same name, are global. This will not produce the same behavior as the previous scenario of binding multiple fields to the same data value, because LiveCycle Output expects to find a data value, with the same name as the global fields, located outside of the current data record.

In the following example XML, data for two separate invoice documents is represented. And prior to either invoice, a data value named `date` is defined. Assuming the invoice form has been designed in LiveCycle Designer such that a field named `date` is defined as global, the value of the `date` data value will be bound to the `date` field of each generated invoice.

Global data placement in XML

```
<invoices>
  <date>Sunday March 31, 2007</date>
  <invoice>
    ...data for the first invoice...
  </invoice>
  <invoice>
    ...data for the second invoice...
  </invoice>
</invoices>
```

Both LiveCycle Output mechanisms of merging individual data values into multiple form fields produce different effects, address different use-cases, are complementary, and can be combined within a form. Choose the appropriate solution depending on whether you need to simply merge

data into multiple fields within a form, or merge data into multiple fields across consecutively generated forms.

Form-Driven Data Binding

A form-driven approach to generating a document occurs when the form contains all of the rules necessary to generate a document by binding the data to the form fields and subforms.

By explicitly connecting form objects to specific data items in LiveCycle Designer, the data-binding properties of the form objects are populated. Because of these explicitly designed bindings, the names of the form objects, and their location within the structure of the form, do not need to correspond to the names or position of the associated data items. The bindings determine how the form is constructed and populated with data.

For more detailed information on data-binding with LiveCycle Designer, consult the LiveCycle Designer product documentation.

Data-Driven Data Binding

Data-driven data-binding occurs when a form is designed to permit the data to take some control over how the form is generated from the data. Form-driven data-binding places an emphasis on defining rules that describe how the document will be constructed, and storing these rules within the form itself. In contrast, a form intended to facilitate a data-driven approach will have fewer rules defined, and will rely upon the particular sequence of data to drive the process of constructing the document.

When a form object lacks an explicit data-binding, LiveCycle Output attempts to match the form object by name to a data item that has not yet been bound to another form object. For more information on this approach, consult the XFA Specification for information on the topics "Data Binding" and "Automatic Data Binding".

Multi-Record Data

Often there is a need to generate multiple documents from a single data file that contains multiple records of data. Because XML has a requirement that there be only one outermost element, multiple records must be enclosed within a single, often arbitrary, element. Consider the following example XML data:

Multi-record data:

```
<batch>
  <invoice>
    …data elements for the first invoice…
  </invoice>
  <invoice>
    …data elements for the second invoice…
  </invoice>
</batch>
```

In the above example, two records of invoice data are enclosed within a `batch` element. The name of the outer element, in this case a `batch` element, is not special – it only serves to satisfy the XML requirement of a single outermost element. Also, the records within this example all represent the same type of form: an invoice. However, data representing different types of forms can be represented within a single data file, as follows:

Multi-record heterogeneous data:

```
<batch>
  <invoice>
    …data elements for the first invoice…
  </invoice>
  <purchase_order>
    …data elements for a purchase order…
  </purchase_order>
  <invoice>
    …data elements for the second invoice…
  </invoice>
</batch>
```

In order for LiveCycle Output to distinguish between a data file intended to produce a single document, versus data intended to produce multiple documents, LiveCycle Output must be informed of the element name that signifies the records, such as `invoice`. In cases of heterogeneous record data, such as the above mix of invoices and purchase orders, the level of the element within the XML structure is provided instead of an element name. A record element level of 2 (two) would indicate that each second level element in the above examples represent a record.

The record element name or level can be specified as a property on an output service operation from within LiveCycle Workbench, or programmatically as a property via the LiveCycle Output API.

For more information on record mode handling, consult the LiveCycle Output product documentation, or the topic "Record Mode" in the XFA Specification.

Multi-record data can be combined with a feature, known as Search Rules, where LiveCycle Output automatically selects the appropriate form for a data record (the section called "*Identifying the Form*" provides more information).

Search rules and multiple records of heterogeneous data can be usefully combined to produce a document comprised of several forms that are distinct, but related by a common purpose. This resulting document is often referred to as a document package.

Consider a scenario where a document package is comprised of a letter, followed by a financial document (such as a loan), and concluded with a document intended to capture one or more signatures. These individual documents can be designed and maintained separately, and reused within different document package scenarios. Given an appropriate set of data records, and combined with search rules to select the appropriate forms, a unique document package is constructed.

Central achieves an equivalent result through the use of the `^form` command to vary between a number of different forms within one data file. While page number will be continuous throughout the document produced by Central, such a document package produced by LiveCycle Output will exhibit page numbering that resets for each document within the overall document package.

Modifying Form Objects from Data

Typically data binding is strictly the process of connecting the content of a form field with a data value, but LiveCycle Output forms provide an additional capability to have other properties of form objects retrieve their values from the data. Subform, form fields, radio-button groups, and even static form objects (regions of text, images, etc.) can be configured to bind one or more of their properties to the data.

One example for binding the properties of a form object to data is populating the caption property of a field from the data. A field's caption is the visible label, or prompt, that informs the user of what information

Here a just a few simple scenarios where this capability is useful:

- Populating the items of a drop-down list from the data.
- Altering the field prompt, or caption, based on a data value.
- Altering the validation error messages of a field, based on a data value.

The ability to bind form object properties to data is activated by selecting **Show Dynamic Properties** from the Object Palette within LiveCycle Designer, or by selecting **Show Dynamic Properties** from the **Tools > Options > Data Binding** dialog. Properties that support binding are then highlighted in the user-interface, and may be bound to data. Consult the LiveCycle Designer product documentation for more information on "Dynamic Properties".

5. Document Generation

Agents and Services

Central functionality is exposed by individual software components known as agents, such as the most commonly used Print Agent that generates output. Agents are invoked according to a set of rules expressed in the Central Job Management Database (JMD).

LiveCycle is also comprised of individual software components, known as services, that perform specific operations. LiveCycle Output provides service operations for combining data with a form, generating output, sending output to a printer, and more.

Conceptually similar to the job definitions within the Central JMD, LiveCycle services and their operations can be assembled together to perform complex tasks by constructing processes with the LiveCycle Workbench, a graphical design environment for developing processes.

Invoking LiveCycle Output

The primary mechanism that Central utilizes to receive data files, or jobs, for processing is the collector directory: a location on the file system watched by Central. Any file written to the collector directory, depending on how Central is configured, may be eligible for processing. In addition to the collector directory, data files may be submitted to Central via name pipes, a print queue, or email.

LiveCycle Output can be invoked in a number of different ways. A Java program can take advantage of the LiveCycle Java API and invoke LiveCycle Output directly. A web-service interface is also provided to invoke LiveCycle via SOAP messages. LiveCycle may also be invoked via email, or via a mechanism called watched folders that is equivalent to the collector directory concept in Central.

Central provides a Mail Reader Agent that can be configured to monitor a mailbox for new messages, and any received messages are deposited into the Central collector directory. LiveCycle provides an Email Service that may be configured to monitor an email account for new messages (supported protocols include POP3, and IMAP) and process any data file attachments. The result of processing these data files, possibly generated PDF files, can be sent back to the originating user or another user, by the Email Service.

LiveCycle watched folders provide a similar range of functionality to the Central collector directory. The number and location of watched folders is determined by the configuration settings of the LiveCycle server. Individual watched folders can be configured to invoke a particular LiveCycle service operation, such as generating a PDF document, or printing to a PostScript printer. Separate directories can be configured to receive the resulting output, capture files that failed to process successfully, and hold preserved copies of input files.

Identifying the Form

A typical invocation of LiveCycle Output will include, at a minimum, two resources: a data file, and a form file. In a simple case, the form file may be specified as one of the properties on the Output service operation, where a process to generate output has been designed within LiveCycle

Workbench. Multiple processes could be designed, each configured to generate output with a particular form.

Instead of explicitly specifying the form as part of the process configuration, an alternative is to create a series of rules that associate a search pattern with a form. This mechanism is similar to the Central `JFNOJOB` capability. For each received data stream, LiveCycle Output will perform a text search from the beginning of the data, seeking to match against any of the supplied search patterns. It is important to note that the search is a basic text search over the data, and does not first parse the XML data. Therefore, the search pattern could be designed to match against either the markup, or content, of the data. The number of bytes from the beginning of the file is also configurable. Upon finding a match, the data will be processed with the form associated with the search pattern. A default form file may also be specified, to be used when no search patterns are matched against the data. When this feature is combined with multi-record data files, it is possible to automatically select the appropriate form for each record.

For more information on this functionality, consult the LiveCycle Output documentation for a discussion of "Search Rules".

Document Generation and Print Options

In addition to providing a data file, and optionally specifying a form file, a number of other processing options are available for customizing how the generated PDF, or printed output, will be produced.

Generic options include: specifying whether LiveCycle Output should generate one output stream, or multiple output streams; the form identification search rules; and, identifying the data elements that signify record boundaries. PDF specific options include the desired PDF version and conformance level; and, print specific options such as adjusting the page origin, and specifying the destination printer URI, printer LPD URI, or printer queue name.

For more information on the generic, PDF, and print options, consult the LiveCycle ES documentation for a discussion of the `RenderOptionsSpec`, `PDFOutputOptionsSpec`, and `PrintOptionsSpec` interfaces respectively.

Testing and Previewing

Testing your form, and its data bindings, from within the Designer software saves both time and paper, by avoiding the need to deploy the forms to the server and generate a printed test output.

Output Designer provides the ability to test both static and dynamic forms with its Test Presentment feature. Forms may either be tested with an existing data file, or Output Designer can create a sample data file suitable for testing the form. Forms can be viewed as a PDF using Reader or printed to a target printer.

LiveCycle Designer provides an equivalent capability to test and preview forms with sample data. The resulting form, populated with data, is viewed as a PDF preview directly within the Designer software. From within the Designer's Form Properties dialog, the type of preview may be chosen as either a static interactive form, a dynamic interactive form, or a non-interactive printable form. The form may be previewed with an existing XML data file, or sample data may be generated by clicking the Generate Preview Data button from within the Form Properties dialog.

Another convenient way of performing a print test is available from the Print dialog within LiveCycle Designer. In addition to the common options provided by the Print dialog, LiveCycle Designer extends the dialog with settings to specify sample data, whether the form should be printed in a simplex or duplex mode, and the target print device configuration. In this way the form can be printed directly to a target printer to see a printed result.

Device Profiles

Forms created with Output Designer are designed specifically for one or more specific output formats or devices. However, forms created with LiveCycle Designer are not constrained, or targeted, for a particular format or device; they are generic forms that LiveCycle Output uses to produce output to a particular format or device at run-time, rather than making this decision at design-time.

Output Designer provides a mechanism for you to select one or more target formats and printers, by selecting from the available presentment targets. Depending on which presentment targets you select, a range of page sizes and fonts become available based upon the paper and font support of the devices. When creating a new form in Output Designer, your initial workspace, page size and orientation, and available fonts are all determined from the particular presentment target that has been nominated as the default presentment target. Before a form can successfully generate output to a presentment target, the form must be compiled after selection of the presentment target with Output Designer.

Advanced users of Print Agent and Output Designer may be aware that each supported presentment target is actually described by the contents of a text file, located within the product installation directory, with an `.ics` file extension. Hence, these files are commonly called **ICS files**. Because an ICS file describes the supported features and capabilities of a Central presentment target, the Central documentation references ICS files as the means for an advanced user to create or customize an ICS file in order to more accurately support their particular printer. For example, while Central provides built-in support for finishing options, such as stapling, on a number of supported printers, you may have a requirement to print forms on a compatible PCL or PostScript printer that supports stapling via a vendor-specific printer command. By customizing an ICS file to include the printer's specific stapling command, or creating a new ICS file based upon an existing compatible ICS file, Print Agent will be able to utilize the printer's stapling feature.

LiveCycle Designer does not require that a form be compiled, nor does it require the selection of specific output formats or printers during the process of designing a form. When designing a form with LiveCycle Designer, the page size and font capabilities of your target printer remain an important consideration. LiveCycle Designer provides a wide variety of pre-defined page sizes, and permits the creation of custom page sizes.

The set of fonts available within the LiveCycle Designer workspace is primarily determined from the fonts present on the system. In order to accommodate unique font requirements, where the font may not be present on the system, LiveCycle provides font mapping capabilities for the server-side runtime components of LiveCycle Output, and the LiveCycle Designer environment. Font mapping is further described in the section called "*Font Availability and Mapping*".

The supported features and capabilities of printers supported by LiveCycle Output are described by **XDC files**, which are conceptually similar to Output Designer's ICS files. In the same way that ICS files can be modified to accommodate the unique features of a particular printer, XDC files may

also be customized, or new XDC files may be created based upon an existing XDC file, by utilizing the XDC Editor tool within the LiveCycle Workbench.

LiveCycle Designer also utilizes its own XDC file, `Designer.xdc`, to determine the range of page sizes, additional fonts, and other features, that shall be exposed in the LiveCycle Designer workspace.

LiveCycle Output provides an additional XDC file, `Designer.xdc.zebra`, appropriately configured for designing Zebra printer labels, with page sizes and fonts appropriate for use with a Zebra printer. By copying the `Designer.xdc.zebra` file to a file named `Designer.xdc` (backup the original `Designer.xdc` first), and restarting LiveCycle Designer, these Zebra-specific features will be exposed in the LiveCycle Designer workspace.

Font Handling

Font Availability and Mapping

The set of fonts available within Output Designer is determined by the fonts available within selected presentment target. For instance, if the presentment target selected is the Generic Microsoft Windows Driver, all of the fonts available on the system are available. On, the other hand, if the presentment target is the Adobe Portable Document Format, then the available fonts correspond to the set of standard PDF fonts. Output Designer also provides mechanisms to further customize the set of available fonts associated with a presentment target. Font mapping can also occur when the final document is generated by Print Agent.

The set of fonts available within the LiveCycle Designer workspace is primarily determined from the fonts present on the system, plus any fonts enumerated in the `Designer.xdc` file (as described in the previous section the section called "*Device Profiles*"). When ensuring that fonts are present on your systems, consider that fonts must be available both on the systems running LiveCycle Designer, as well as the servers running the LiveCycle server components.

In order to accommodate the unique font requirements of printers, LiveCycle Designer and Output provide a robust font mapping capability. LiveCycle Designer, within the product installation directory, has a `Designer.xci` configuration file that contains the default set of font mappings. This configuration file can be extended to accommodate additional font mappings. However, this configuration file only impacts the LiveCycle Designer workspace. Font mappings intended to be in effect at the time of processing a form in concert with a particular XDC device profile must also be present in the target XDC file.

The following is an excerpt from the `Designer.xci`, showing the XML markup describing a number of font mapping statements that will map requests for a variety of Helvetica fonts to similar Arial fonts:

Designer.XCI font mapping

```
<equate from="Helvetica Black_*_*" to="Arial Black_*_*" force="0"/>
<equate from="HelveticaBlack_*_*" to="Arial Black_*_*" force="0"/>
<equate from="Helvetica-Black_*_*" to="Arial Black_*_*" force="0"/>
<equate from="Helvetica_*_*" to="Arial_*_*" force="0"/>
<equate from="Helv_*_*" to="Arial_*_*" force="0"/>
```

The font mapping capabilities provide for very fine-grained control over mapping. Whole typefaces can be mapped, or an individual typeface with a particular weight and posture can be

mapped. The force attribute denotes whether a font should be always mapped, or only when the requested font is not available.

The order of the equate font mapping statements is important, with the statements evaluated in order until a matching statement is encountered.

Font Embedding

LiveCycle forms can download, or embed, additional fonts into the generated output. When output is generated for either PCL or Postscript, preference will be given to fonts that are known to be standard fonts included on the printer. These fonts are known as printer-resident fonts.

Because font downloading is avoided in such cases, using printer-resident fonts results in less generated PCL or Postscript. To design a form that will use printer-resident fonts, choose typeface names in LiveCycle Designer that match those available on the printer. A list of fonts supported for PCL or Postscript can be found in the corresponding device profiles (XDC files). Alternatively, a font mapping can be created to map non-printer-resident fonts to printer-resident fonts of a different typeface name. For instance, in a Postscript scenario, references to the Myriad Pro typeface could be mapped to the printer-resident Helvetica typeface.

PDF and PostScript output can support embedded Type-1, TrueType, and OpenType fonts. PCL output can support embedded TrueType fonts. However, Type-1 and OpenType fonts are not embedded in PCL output, and therefore any content formatted with Type-1 and OpenType fonts is rasterized and generated as a bitmap image which can be large as well as slower to generate.

Downloaded or embedded fonts are automatically subsetted when generating PostScript, PCL, or PDF output. This means that only the subset of the font glyphs required to properly render the generated document will be included in the generated output.

No font downloading or subsetting capability is provided when generating output for a Zebra printer.

Central provided a capability where font downloading was tracked and managed, in an effort to only download fonts to a presentment target when Central considered that the font had not previously been downloaded. However, this capability was fragile, as the actual downloaded fonts on the presentment target could easily get out of sync with Central by simply cycling the power to the output device. LiveCycle Output does not attempt to provide an equivalent font management capability.

Paper Handling

Duplexing

Output Designer and Central provided a number of different ways of indicating whether a form should be printed simplex (one-sided) or duplex (double-sided), and whether pages should be duplexed along the left/long edge or top/short edge. These settings can be specified as a property of the whole form, from within a field-nominated data stream, or as an option on the job definition or Print Agent command-line.

LiveCycle Designer exposes control over duplex printing in two ways, depending on whether the form is intended to be generated to a PDF and subsequently printed from Adobe Reader or Acrobat, or printed directly to a PCL or PostScript device.

When the form is intended to generate a PDF, settings related to how the PDF should be printed can be configured from the Form Properties dialog within LiveCycle Designer. These settings include the number of copies to print, and duplexing. Subsequent printing of a PDF document with Adobe Reader or Acrobat will respect these settings.

Duplexing can also be specified via the `pagination` property on the LiveCycle Output service operations available within LiveCycle Workbench, and also available from the Java API and web-service interface. For more information on the generic, PDF, and print options, consult the LiveCycle ES documentation for a discussion of the `RenderOptionsSpec`, `PDFOutputOptionsSpec`, and `PrintOptionsSpec` interfaces respectively.

In addition to basic selection of simplex or duplex printing, there are additional capabilities available in LiveCycle Designer to design forms that adjust according to simplex and duplex printing scenarios.

Master pages can be assigned to the odd numbered (front side) or even numbered (back side) printed surfaces. In this way, slightly different master pages can be designed for the front and back, and LiveCycle will automatically select the appropriate master page depending on whether it is currently printing on the front or back of the page. One common use case for this capability is to create similar master pages that place the running page count on either the left or right side of the page, and assigning the master pages as odd or even to ensure that the page count is always on the inside or outside of a duplex printed document.

On a finer-grained level LiveCycle Designer provides a presence property on form objects that commonly is used to indicate that an object should be visible or hidden from a PDF screen display of the document or when printed. Form objects can also be configured with a presence setting that indicates the object should only appear when the document is printed simplex, or duplex.

Tray Handling

Output Designer distinguishes between paper (page) size, and which input tray on the printer should provide the requested paper. This accommodates scenarios where a printer may have multiple different types of a particular paper size loaded into different input trays, permitting a document to select paper from individual trays on a per-page basis.

LiveCycle forms also provides the capability to select paper from different input trays, but LiveCycle Designer does not expose paper size and input tray selection as two distinct properties. Instead, LiveCycle Designer permits any master page to be associated with a paper type selected from a set of supported paper types defined in the `Designer.xdc` device profile. Within the device profile, each paper type can be configured to select paper from a particular input tray.

LC Designer provides four duplicates of common paper types for Letter and Legal paper sizes; specifically to accommodate paper tray selection. These duplicate paper types are given the following names “Letter Plain”, “Letter Letterhead”, “Letter Color”, and “Legal Special” and there is a similar set for Legal.

The XDC Editor within the LiveCycle Workbench is used to map or assign a physical input tray to a paper type; typically the four names provided in LC Designer will cover most tray selection needs. For example, the deployed device profile could be modified such that the "Letter Color" paper type will cause the printer to select yellow paper loaded into a secondary letter-sized input tray. Because LiveCycle Output matches paper types used in the form, by name, against paper types defined in the device profile deployed to the server, only the deployed device profile needs to be modified to ensure the appropriate input tray selection. Consult the XDC Editor documentation for more information on editing device profiles and deploying them to your design and production environments.

While the 4 provided paper types will normal be adequate; it is possible to create additional paper types. Additional paper types can be used in the XDC editor simply by making up a new paper type name, but it must exist in LiveCycle Designer to be used, in order to add paper types the `Designer.xdc` file must be hand edited; copying an existing entry of the correct paper size and changing the name as desired.

Faxing

Central provides the ability to fax documents via direct integration with fax servers, and includes device profiles accordingly. In addition, the field-nominated data format includes a command, `^fax`, dedicated to this purpose. LiveCycle Output takes a different approach to fax support.

Recognizing that modern fax servers accept generic PCL or Postscript print streams (often via a watched folder), and receive fax addressing information by way of an accompanying metadata file, LiveCycle does not provide device profiles for specific PDF, Postscript, or PCL fax servers; nor does it require that a form be designed specifically for faxing, or include a command dedicated to providing fax addressing information.

Adding a fax destination to your document output solution is accomplished by creating a process definition in LiveCycle Workbench that produces the document output, along with the addressing information, and delivers both to a fax server. For an example of how to create such a fax process, see the sample fax output process included with LiveCycle Output.

6. Web Output Pak

The preceding chapters have primarily focused on Central document generation. However, the Web Output Pak, as part of the Central solution, provides a similar set of document generation capabilities. In fact, Web Output Pak uses the document generation and output software component from Central, and uses forms created with Output Designer.

Web Output Pak processes data received from a web browser, or in XML format, and data is manipulated via a programmatic interface similar to an XML DOM. Concepts and experience gained from working with XML and DOM-style interfaces in Web Output Pak is beneficial when moving to the standards-based XML tools and interfaces of LiveCycle Output.

Because Web Output Pak is a subset of the overall Central product family, the other sections of this document are also relevant in the context of many Web Output Pak solutions, and this section simply addresses several topics specific to Web Output Pak.

XPR and Transaction Processing

Web Output Pak applications are developed by coordinating the data handling, invocation of built-in and custom software agents, and output generation, by a markup language known as **XML Processing Rules** (**XPR**) and a rule processing engine known as the **Transaction Processor**. Because XPR markup is effectively a special-purpose programming language, there is no migration or conversion mechanism for XPR files to an equivalent mechanism in LiveCycle Output, such as LiveCycle process definitions.

In place of Transaction Processor and XPR, LiveCycle provides the **Process Management** component and the ability to assemble LiveCycle services and operations with the LiveCycle Workbench.

For more information, consult the product documentation on LiveCycle Process Management and LiveCycle Workbench.

Agents

Web Output Pak provides a number of built-in software agents, including the HTML Agent and PDF Agent that generate HTML and PDF output respectively.

The HTML Agent accepts data and populates regions of an HTML template file that contains Web Output Pak specific markup, generating a resulting HTML file that is a combination of the original HTML template and the supplied data. LiveCycle Output does not provide a migration or conversion mechanism for these HTML template files. However, LiveCycle Forms does have the capability to generate HTML output from forms created with LiveCycle Designer.

The functionality of PDF Agent is equivalent to the PDF output capability provided by Central Print Agent, and generating PDF output is a core capability of LiveCycle Output.

7. Hosting Environment

The Central product is a native application designed to run on a number of supported platforms (for more detail see table at Chapter 2, *Comparison Summary*. The hosting environment for Central is simply a supported operating system.

LiveCycle ES is a Java enterprise application, also known as a J2EE (Java 2 Platform, Enterprise Edition) application. As a J2EE application, LiveCycle ES must be run within a supported J2EE application server environment, installed on a supported platform. The application server, and the J2EE foundation, provides a consistent technology foundation for deploying and managing multi-tier enterprise applications. Scalability can be configured and managed at the level of the J2EE application server, whereas attempting to manage scalability with Central is usually reduced to running several instances of Central and rolling your own solutions for interacting with, and managing, these instances.

LiveCycle ES also requires a database for long-term storage of information such as process definitions and state, and the form repository. These system requirements for LiveCycle ES are summarized in the table at Chapter 2, *Comparison Summary*, and in detail on the Adobe website:

- http://www.adobe.com/products/livecycle/systemreqs.html

It is also important to recognize that LiveCycle ES is designed as a server-based solution, and while installation on a workstation is useful for application development purposes, for production use LiveCycle ES should be installed on an appropriately configured server. The practice of installing many instances of Central across a number of workstations as a means of distributing processing and achieving some manner of scalability is not practical with LiveCycle ES.

To easily achieve a production-capable LiveCycle ES installation on a supported Windows platform, Adobe provides a **turnkey installation** option that installs LiveCycle ES, a JBoss application server, and a MySQL database server providing a fully functioning installation. Alternatively, LiveCycle ES may be installed to integrate with your choice of supported application server and database server. Detailed instructions on installation and configuration are provided in the LiveCycle product documentation, including the following guide:

- "Preparing to Install LiveCycle ES"
  http://help.adobe.com/en_US/livecycle/es/prepareinstall.pdf

8. Field-Nominated Commands

The intent of this chapter is to review a number of the commands that comprise the field-nominated format used by Central agents, especially Print Agent, and provide a brief discussion of how the command relates to an equivalent capability in LiveCycle Output. Not all commands supported by Central or Print Agent are included.

In most cases, the commands will not relate directly to XML markup that can be interpreted by LiveCycle Output. This is because, as described in the section called "*Data Formats*", field-nominated data is both a command language and a data format, whereas XML data is strictly a data format. Nonetheless, given that so much of the Central and Print Agent functionality is exposed through field-nominated commands, it is valuable to present them here as a way of referring to LiveCycle Output functionality.

^continue

See the section called "^record".

^copies

This field-nominated command, when it appears at the beginning of a field-nominated data file, sets the number of printed copies that shall be produced by Print Agent.

In LiveCycle Output, the number of copies is set as a copies property on the `PrintedOutputOptionsSpec` interface or the output option properties exposed within LiveCycle WorkBench.

^currency

This field-nominated command sets a number of localization properties associated with the formatting of monetary values, such as the currency symbol, decimal point, thousands separator, and more.

LiveCycle forms contain the equivalent property values for one or more locales associated with the form. The appropriate locale information is automatically incorporated into the form definition by selecting one or more locales from within LiveCycle Designer. In addition, custom locales can be created. This is discussed in the section called "*Locale Settings*".

^data

See the section called "^record".

^define

This field-nominated command is used to assign a value to a dictionary variable.

With the field nominated format there is no other way, other than dictionary variables, to store information in the data in a manner where it can later be recalled from elsewhere in the data or

preamble processing. However, LiveCycle forms operate with all of the XML data loaded into a DOM, accessible to scripting and data bindings. Therefore the need for an additional mechanism for storing, and later recalling, data is not required with LiveCycle forms.

In addition, it should be noted that LiveCycle forms may be designed to include form variables. For more information see the section called "*Form Variables*".

^duplex

This field-nominated command controls how the document will be printed in duplex mode.

A LiveCycle form cannot be set to duplex from within the data. Duplexing may be controlled from within LiveCycle Designer as one of the form properties, or determined by specifying an appropriate `pagination` property value within a process definition or passed to the LiveCycle Output API. For more information see the sections called "*Duplexing*" and "*Device Profiles*".

^eject

This field-nominated command causes Print Agent to indicate a page break in the generated output.

Within LiveCycle forms, all control over pagination is specified within the form itself, using LiveCycle Designer.

^field

This is the most commonly used field-nominated command. It directs the following data into a particular field indicated by name.

The equivalent mechanism in XML data is the markup tags themselves, where the tag names can optionally indicate the field name to which the enclosed data belongs. For a discussion of this, see the section called "*Data Formats*".

^file

This field-nominated command causes a referenced data file to be included, at the location of the command, in the referencing data file.

One way to achieve this in XML is by using a markup feature related to XML known as **XInclude**. However, LiveCycle Output does not currently support XInclude.

^form

This field-nominated command causes Print Agent to switch processing to another form file.

While LiveCycle forms do not provide an equivalent capability of switching forms from within data, there is a capability to reference external form fragments from within a form, and a feature to automatically select an appropriate form based on the data known as search rules. For more information on search rules, see the section called "*Identifying the Form*". For more information on fragments see the section called "*Fragments*".

^global

This field-nominated command defines a global data value that will automatically populate fields with the same name as the global value.

LiveCycle forms do not permit the data to indicate that values should be globally applied to form fields that share a common name. Global fields can be designated within LiveCycle Designer, and there is a convention for defining a data value that can be available across multiple records of data. These capabilities are described in the section called "*Global Fields and Global Data*".

^graph

This field-nominated command specifies that a referenced image file, rather than a text value, shall be placed on the form at the location of a particular field.

LiveCycle forms provide a specific type of field designed to enclose an image, known as image fields. Such fields can be populated with base-64 encoded data representing the image data, and additional properties are provided by LiveCycle Designer to control how the image should be adjusted within the dimensions of the field. Alternatively, the HTML `img` element can be used to insert a referenced image file into a rich-text field. For more information, see the "Image Data and Rich-Text Reference" of the XFA Specification. The rich-text format is also briefly discussed in the section called "*Formatting Rich-Text Data*" of this document.

^group

This field-nominated command is intended to imply structure within a field-nominated format that would be otherwise unstructured. It also causes preamble processing to be executed associated with the referenced group, and is often used as a mechanism to instantiate a subform, transition to another page, or switch to another form entirely.

LiveCycle forms rely upon the intrinsic capabilities of XML to describe structured data, and utilize this structure information when binding the data to a form. In this way, structure can cause data to populate specific subforms, on specific pages. In other words, the philosophy of utilizing XML data with LiveCycle forms is to appropriately structure your data, appropriately structure your form into subforms and subform sets in LiveCycle Designer, and express the relationship between the two structures by adding data bindings to the form.

^inlinegraphbegin, ^inlinegraphend

These field-nominated commands provide a mechanism to embed image data directly within a data file. See the section called "^graph" for more information.

^key

See the section called "^record".

^macro

This field-nominated command causes a previously downloaded macro, a printer-resident cached representation of a subform, to be immediately executed and included in the output.

This command provides a means to indirectly insert PCL or PostScript commands into the output stream. There is no equivalent capability in LiveCycle forms.

^page

This field-nominated command causes Print Agent to perform a page break, and continue processing on a specific page of the current form.

Within LiveCycle forms, the decision of when to perform a page break, and on which page to resume processing, is strictly determined by the pagination rules designed into the form with LiveCycle Designer.

^passthru

This field-nominated command is a very low-level command that sends a printer-specific byte stream into the generated output, and subsequently to the print device. There is no equivalent capability in LiveCycle forms. Some customization of LiveCycle Output's printer interface is possible by modifying device profiles. See the section called "*Device Profiles*".

^popsymbolset

See the section called "^symbolset".

^pushsymbolset

See the section called "^symbolset".

^record

This command is used to create a fixed-format record definition where the data is packed into rows and columns of a textual data file. LiveCycle Output does not support fixed-record format data.

^symbolset

This field-nominated command informs Print Agent that subsequent data is encoded according to a specific character encoding. During the processing of data, Print Agent can be instructed to remember and recall specific character encodings with the `^pushsymbolset` and `^popsymbolset` commands respectively.

LiveCycle Output processes XML data, and the XML specification requires that all of the character content of an XML document be expressed in a single encoding of Unicode characters. For more information see the section called "Unicode".

^shell

This field-nominated command provides a mechanism to execute an external software program from a specific point in the processing of the data file. LiveCycle Output does not provide an equivalent capability.

^subform

This field-nominated command causes Print Agent to switch the currently active subform to a specific subform within the current form, or a subform within another form.

LiveCycle forms to not provide a mechanism to call a specific subform from within the data. Instead, LiveCycle forms rely upon the structure within XML data, and utilize this structure information when binding the data to a form. In this way, structure can cause data to populate specific subforms, on specific pages. In other words, the philosophy of utilizing XML data with LiveCycle forms is to appropriately structure your data, appropriately structure your form into subforms and subform sets in LiveCycle Designer, and express the relationship between the two structures by adding data bindings to the form.

^tab

This field-nominated command defines, or resets, tab stop positions. Tab stops can be set for the duration of applying data to the current field, or tab stops may be set that act as the default tab stops for the duration of processing the entire job.

LiveCycle Output recognizes a subset of HTML markup to express text formatting within data, and within the form itself. Additional LiveCycle-specific extensions to HTML markup are supported to specify tab stops. For more information, see the "Rich-Text Reference" portion of the XFA Specification.

^trayin, ^trayout

These field-nominated commands select a specified input tray, or output tray, respectively.

There is no capability in LiveCycle Output to indicate tray selections from within the XML data. Each page of a LiveCycle form may be configured to select a particular paper size, and tray selection is determined based on tray information associated with the paper size as defined by the device profile. For more information, see the section called "*Tray Handling*".

^$page

This field-nominated command causes the current page number to be set according to a supplied value.

Within LiveCycle forms, page numbering is strictly a property of the form, and of master pages in particular. For more information see the sections called "*Page Counts*", and "*Master Pages*"