FlexNet Publisher 2018 R2 (11.16.0)

Programming Reference for Trusted Storage–Based Licensing

Legal Information

Book Name: FlexNet Publisher 2018 R2 (11.16.0) Programming Reference for Trusted Storage–Based Licensing

Part Number: FNP-11160-PRTS00

Product Release Date: June 2018

Copyright Notice

Copyright © 2018 Flexera

This publication contains proprietary and confidential information and creative works owned by Flexera and its licensors, if any. Any use, copying, publication, distribution, display, modification, or transmission of such publication in whole or in part in any form or by any means without the prior express written permission of Flexera is strictly prohibited. Except where expressly provided by Flexera in writing, possession of this publication shall not be construed to confer any license or rights under any Flexera intellectual property rights, whether by estoppel, implication, or otherwise.

All copies of the technology and related information, if allowed by Flexera, must display this notice of copyright and ownership in full.

FlexNet Publisher incorporates software developed by others and redistributed according to license agreements. Copyright notices and licenses for these external libraries are provided in a supplementary document that accompanies this one.

Intellectual Property

For a list of trademarks and patents that are owned by Flexera, see https://www.flexera.com/producer/company/about/intellectual-property/. All other brand and product names mentioned in Flexera products, product documentation, and marketing materials are the trademarks and registered trademarks of their respective owners.

Restricted Rights Legend

The Software is commercial computer software. If the user or licensee of the Software is an agency, department, or other entity of the United States Government, the use, duplication, reproduction, release, modification, disclosure, or transfer of the Software, or any related documentation of any kind, including technical data and manuals, is restricted by a license agreement or by the terms of this Agreement in accordance with Federal Acquisition Regulation 12.212 for civilian purposes and Defense Federal Acquisition Regulation Supplement 227.7202 for military purposes. The Software was developed fully at private expense. All other use is prohibited.

Contents

1 Overview of Trusted Storage–Based Licensing . . . 13

Introduction . . . 13

Description of the Trusted Storage–Based Transactions . . . 14

Systems Involved in an Activation Transaction . . . 14

Choosing an Activation Mechanism . . . 15

Activation Transaction Types . . . 15

Composite Transactions . . . 15

Composite Transaction Features . . . 16

Composite-Transaction Mechanisms and APIs . . . 16

Composite Transaction Example . . . 17

Components Used With Trusted Storage–Based Licensing . . . 18

Application Activation Utility . . . 18

Server Activation Utility . . . 19

Activation Specification Record . . . 20

FlexNet Licensing Service . . . 20

Activation Library . . . 20

Trusted Storage . . . 21

SOAP Module . . . 22

Activation Server . . . 23

License Server . . . 23

FlexEnabled Applications . . . 25

License Activation in a Virtual Environment . . . 25

Features Added or Updated to Support Virtualization . . . 26

2 Transactions to a License Server . . . 27

Introduction . . . 27

Components Required on the License Server . . . 28

Summary of the Activation Process . . . 29

Activation Using the Publisher's Activation Server . . . 29
Transfer of Licenses Between License Servers . . . 30
Limiting the Number of Transfers . . . 31
Local Activation Using an ASR . . . 31
Repair Transaction . . . 32
Trusted-Storage Changes Due to Illegal Tampering . . . 32
Trusted-Storage Changes Due to Valid Actions . . . 32
Return Transaction . . . 32
Description of License Rights in Server-Side Trusted Storage . . . 33
Types of License Rights in Server-Side Trusted Storage . . . 33
Feature Definition Lines for Licenses in Server-Side Trusted Storage . . . 33
License Counts . . . 34
Managing the License Pool Counts . . . 35
Limiting License Duration of Activated Overdraft Licenses . . . 36
Fulfillment Record in an Activation Response . . . 36

3 Configuring Security for Trusted Storage . . . 38
Overview of Trusted Storage Security . . . 38
Defining the Trusted Configuration . . . 39
Windback Detection . . . 39
Binding . . . 40
Binding Default . . . 40
Possible Binding Identities . . . 41
Special Conditions Affecting Binding Identities . . . 41
Anchoring . . . 41
Deploying Trusted Configuration . . . 43
The Prepped Trusted Configuration . . . 43
How Prepped Trusted Configuration Works . . . 44

4 Implementing Trial Licenses and Trial Packs . . . 45
Creating a Trial License . . . 45
Deleting Trial Anchors . . . 47
Using Trial Packs . . . 47
Trial Pack ID XML Snippet . . . 49
Support for Trials in Virtual and Non-Virtual Environments . . . 49

5 Adding Function Calls to a FlexEnabled Application . . . 50
Using the Flexible API . . . 50
Adding Function Calls to Your Application . . . 51
lc_set_attr Attributes for Trusted Storage . . . 54
Initialization . . . 54
Handling Checkout Request Denials . . . 55

6 Initializing Trusted Storage . . . 57

FlexNet Licensing Service 57
Installing the FlexNet Licensing Service on Windows 58
Installing the FlexNet Licensing Service on Linux 58
Installing the FlexNet Licensing Service on OS X 60
Sample Applications 60
installanchorservice.exe 60
uninstallanchorservice.exe 61
Trusted Storage Directory Initialization (UNIX Platforms) 61

7 Customizing the Vendor Daemon 63
Basic Vendor Variables 63
ls_allow_tz_override 65
ls_allow_vm 65
ls_compare_vendor_on_increment and ls_compare_vendor_on_upgrade 65
ls_daemon_periodic 66
ls_incallback 67
ls_infilter 67
ls_min_lmremove 67
ls_minimum_user_timeout 67
ls_outfilter 67
ls_show_vendor_def 70
ls_support_custom_daemon_select_timeout 70
ls_user_init1 71
ls_user_init2 71
ls_user_init3 71
ls_user_init_attributes 71
ls_vd_shutdown 72
ls_vendor_msg 73
Advanced Vendor Variables 73
ls_a_license_case_sensitive 74
ls_do_checkroot (UNIX Only) 74
ls_dump_send_data 74
ls_entitlement_based_pooling 74
ls_single_xver_signature 75
ls_ts_borrow_reclaim_percentage 75
ls_ts_update_poll_interval 75
ls_use_all_feature_lines 76
ls_use_exclusive_commRev4 76
ls_user_lockfile 76
ls_user_lock (Windows only) 77

8 Transactions with a FlexEnabled Client 79
Introduction 79
Components Required on a FlexEnabled Client 79
Summary of the Activation Process 80

When Using Short-Code Activation . . . 81
Activation Using the Publisher's Activation Server . . . 82
Activation Using a License Server . . . 83
Local Activation Using an ASR . . . 83
Repair Transactions . . . 83
Trusted-Storage Changes Due to Illegal Tampering . . . 83
Trusted-Storage Changes Due to Valid Actions . . . 84
Return Transaction . . . 84
Short-Code Transactions . . . 84
Fulfillment Record on a FlexEnabled Client . . . 85

9 Using the Activation Borrow API . . . 86
License Borrowing with Activation Borrow API . . . 87
Using the Activation Borrow API . . . 88
Using Fulfillment Records for Borrow . . . 90
Comparison with BORROW Keyword . . . 91

10 Building an Application Activation Utility for Single-Action Transactions . . . 92
About Single-Action Transactions . . . 92
Activation Utilities for Single-Action Transactions . . . 92
Path and Directory Conventions Used in This Chapter . . . 93
Creating an Activation Utility . . . 93
Building and Deploying Your Activation Utility . . . 95
Sending a Reason with a Request Message . . . 96
Using the Vendor Dictionary to Send Custom Data . . . 96
Local Activation . . . 97
Fulfillment Record Manipulation . . . 98
View the Fulfillment Record from an Activation Response . . . 98
Display License Rights in Client-Side Trusted Storage . . . 98
Display Server-Side License Rights Remotely On the Client . . . 99
Delete a Single Fulfillment Record from Trusted Storage . . . 101
Delete Multiple Fulfillment Records from Trusted Storage . . . 102
Enable Fulfillment Record when No Response to Return Request . . . 102
Transactions with a License Server . . . 102
Activation from a License Server . . . 102
Return Fulfillment Record to a License Server . . . 103
Repair from a License Server . . . 104
Using the Manual Response Generator . . . 105
Activation from a Manual Response Generator . . . 106
Return Fulfillment Record to Manual Response Generator . . . 107
Repair Using a Manual Response Generator . . . 108
Transactions with an Operations Server . . . 109
Activation from an Operations Server . . . 110

Return to an Operations Server . . . 110
Repair from an Operations Server . . . 112

11 Building a Server Activation Utility for Single-Action Transactions . . . 113

About Single-Action Transactions . . . 113
Activation Utilities for Composite Transactions . . . 113
Path and Directory Conventions Used in This Chapter . . . 114
Creating an Activation Utility . . . 114
Building and Deploying Your Activation Utility . . . 116
Sending a Reason with a Request Message . . . 117
Using the Vendor Dictionary for Custom Data . . . 117
Local Activation . . . 118
Transactions with a FlexNet Operations Server . . . 118
Activation from a FlexNet Operations Server . . . 118
Return to a FlexNet Operations Server . . . 119
Repair from a FlexNet Operations Server . . . 120
Transactions Between License Servers . . . 121
Transfer of Licenses from a License Server . . . 122
Return a Fulfillment Record to a License Server . . . 122
Using the Manual Response Generator . . . 124
Activation From a Manual Response Generator . . . 124
Return of Fulfillment Record to Manual Response Generator . . . 125
Repair From a Manual Response Generator . . . 127
Extracting Information from Trusted Storage . . . 128
Display Current License Rights in Trusted Storage . . . 129
Display License Rights Activated by This Server . . . 130
Deduction Records . . . 131
Display Complete Contents of Trusted Storage . . . 132
Enable Fulfillment Record when No Response to Return Request . . . 134
Display License Rights Held in Remote Server-Side Trusted Storage . . . 134

12 Building an Activation Utility for Composite Transactions . . . 135

About Composite Transactions . . . 135
Activation Utilities for Single-Action Transactions . . . 135
Path and Directory Conventions Used in This Chapter . . . 136
About Creating Activation Utilities for Composite Transactions . . . 136
API Used to Create an Activation Utility . . . 136
Role of the Activation Utility in the Composite-Transaction Process . . . 136
Creating an Activation Utility . . . 137
Creating a Request and Processing the Response . . . 138
Adding Actions to the Transaction . . . 138
Completing the Transaction . . . 140
Completing the Transaction Online . . . 140
Completing the Transaction Offline . . . 140

Querying the Response Content . . . 141
Obtain Vendor Data at the Response Level . . . 141
List Actions in the Response . . . 142
Obtain Vendor Data for a Specific Action . . . 142
Obtain Attribute Value for a Specific Action . . . 142
Obtain Results of Performing a Specified Action . . . 142
Re-Enable Returned Fulfillment Records . . . 143
Managing Stored Requests . . . 143
Building and Deploying Your Activation Utility . . . 144
Additional Information About Activation Utilities for Composite Transactions . . . 145
Managing Outstanding Requests . . . 145
Re-using Stored Requests . . . 146
About Composite Transaction Requests . . . 146
Identifying the Requesting Machine . . . 147
Composite Request Data Items . . . 147
Additional Composite Request Data Items for Virtual Environments . . . 147
Composite Request Actions . . . 148
About Composite Transaction Responses . . . 150
Requester Verification . . . 151
Information Used to Identify the System Initiating the Request . . . 151
Unique Machine Numbers (UMNs) . . . 151
Machine Identifier (MID) . . . 155
Platform Type Information . . . 155
Error Reporting and Recovery . . . 157
Transactions Using Request and Response Codes . . . 158
Activation Using Request and Response Codes . . . 159
Return Using Request and Response Codes . . . 160
Repair Using Request and Response Codes . . . 161
Display Pending Request Code . . . 161
Cancel Outstanding Short-Code Transaction . . . 162
Error Handling . . . 162
Adding Custom Data to a Short-Code Request . . . 162

13 Working with the Software Publisher Utilities . . . 164
lmcrypt . . . 164
License File . . . 165
Using lmcrypt . . . 165
Error Returns . . . 165
preptool . . . 166
Examples of Preptool Configuration Files . . . 168
Publisher-Specific Preptool . . . 173
Identifying Publisher preptools . . . 173
tcencrypt . . . 174
xmlsign . . . 175
tsreset . . . 176

tsactdiags 177
asrgen 178
License Template 180
Trusted Configuration 181
Functions of asrgen 182
Editing the ASR 182
Example Files To Use with asrgen 183
Manual Response Generator 183

14 The Manual Response Generator 184
XML Documents Required 185
Response Files 185
Activation Response—fulfillmentType 186
Key Files 186
Schema Files 187
ASR Files 187
Using the Manual Response Generator 188
Composite Request 190
Activation Request 191
EntitlementData 192
Trusted Sections on Requesting System 193
Return Request 193
Repair Request 194
Analyzing a Repair Request 195
Short-Code Request 196
Generate a Short-Code Activation Response 198
Generate a Short-Code Return Response 198
Generate a Short-Code Repair Response 199
Command-Line Interface 199
responsegen Syntax 199
responsegen Errors 201
Operator Using responsegen 202
Automated Use of responsegen 203
Config Response 203
Automated Selection of Response Type 203
Automated Failure Response 204
Demonstration Using responsegen 204
Short-Code Activation 204
Short-Code Return Request 206
Using responsegen API 207

15 Defining Feature Definition Lines for a Fulfillment Record 209
Feature Definition Line Syntax 210
Comment Lines 211

Line Continuation . . . 211
Overdraft Conditions . . . 211
License Pools . . . 212
Differences between FEATURE and INCREMENT Lines . . . 213
Pooling by Entitlement ID . . . 213
Vendor Keywords . . . 214
DUP_GROUP . . . 215
HOST_BASED . . . 215
ISSUED . . . 215
ISSUER . . . 216
MINIMUM . . . 216
NOTICE . . . 216
ONE_TS_OK . . . 216
SN . . . 217
sort . . . 217
START . . . 217
SUITE_DUP_GROUP . . . 217
TS_OK . . . 218
USER_BASED . . . 218
VENDOR_STRING . . . 219

A XML Schemas . . . 220
Locating Schema Files in the Toolkit . . . 221
Identifying Schema Versions . . . 222
B_ActivationSpecificationRecord.xsd . . . 223
Creating an Activation Specification Record . . . 223
Description of the XML Data Structure . . . 224
ActSpecRecHeader Element . . . 225
ConfigData Element . . . 226
DataDictionary Element . . . 228
ShortCodeData Element . . . 229
Default Short-Code Formats . . . 231
Default Request Code Formats . . . 233
Default Response Code Formats . . . 234
Default Short-Code Lengths . . . 234
Using Short-Code Overrides . . . 235
Predefined Override Items . . . 235
Defining the RequestOverrides and ResponseOverrides Element . . . 236
Data That Can Be Overridden (or Defined) in Request Codes . . . 238
Example Request Code Overrides . . . 239
Data That Can Be Overridden (or Defined) in Response Codes . . . 241
Example Response Code Overrides . . . 241
Sample ASRs in the Toolkit . . . 243
Sample Trial Editions and Durations for FlexEnabled Clients . . . 243
Sample Trials Supporting Prepped Trusted Configuration for FlexEnabled Clients . . . 243
Sample Trials for FlexNet License Servers . . . 244

Sample Trials for FlexNet License Servers Running in Virtual Environments . . . 244
Sample Trials for FlexEnabled Clients Running in Virtual Environments . . . 244
Sample ASR to Demonstrate Trial Misuse on a FlexEnabled Client . . . 245
Sample ASR for Emergency Situations on FlexEnabled Clients . . . 245
Sample Short-Code ASRs . . . 245
I_FulfillmentRecord.xsd . . . 246
Creating a Fulfillment Record . . . 246
Using asrgen . . . 246
Using the Schema . . . 246
XML Structure of a Fulfillment Record . . . 247
FulfillmentDictionary Data in Client-side Trusted Storage . . . 247
FulfillmentDictionary Data in Server-side Trusted Storage . . . 251
Example Fulfillment Record . . . 254
Fulfillment Record for a Trial License . . . 254
Fulfillment Record for a Concurrent License . . . 256
M1_Dictionary.xsd . . . 258
M2_NonEmptyString.xsd . . . 259
M5_ParameterHeaderType.xsd . . . 259
M5_VersionNumberType.xsd . . . 259
M6_ParameterHeaderType.xsd . . . 259
M7_TrustedIdType.xsd . . . 259
M8_UniqueMachineNumbersType.xsd . . . 259
M9_PlatformType.xsd . . . 259
M10_ConfigTypes.xsd . . . 260
M11_TrustFlags.xsd . . . 260
M12_BreakInfo.xsd . . . 260
M13_RightsId.xsd . . . 260
M14_ResponseActionReference.xsd . . . 260
M15_ResponseDenyReason_NotUsed.xsd . . . 260
M16_TypeValueComment.xsd . . . 260
O_ConfigPhaseOne.xsd . . . 260
Description . . . 261
Transaction Key Length . . . 262
Platforms . . . 263
Anchoring . . . 264
Binding . . . 264
Timing . . . 266
Usage . . . 267
Example Contents of the Input Trusted Configuration File . . . 267
Examples Shipped in Toolkit . . . 268
P1_ConfigPhaseTwo.xsd . . . 269
Examples Shipped in Toolkit . . . 269
For Virtual and Non-Virtual Platforms . . . 269
RevisionType . . . 269

P2_ConfigKeyOut.xsd 269
S_ActivationSettings.xsd 270
W1_CreateActivationResponseParameter.xsd 270
Description 270
ResponseData 271
W2_CreateReturnResponseParameter.xsd 271
Description 271
W3_CreateRepairResponseParameter.xsd 272
Description 272
W4C_CreateConfigResponseClientParameter.xsd 272
W4S_CreateConfigResponseParameter.xsd 273
ConfigData Element 274
ServerConfig Element 274
ClientConfig Element 275
W5_CreateFailureResponseParameter.xsd 275
W6_CreateResponseCodeParameter.xsd 276
W7_CreateCompositeResponseParameter.xsd 277

Glossary 278

Index 289

1

Overview of Trusted Storage–Based Licensing

This chapter provides background on how trusted-storage licensing works. It contains the following information:

- An introduction to the functionality available in the toolkit that uses trusted storage.
- Brief descriptions of toolkit components that support activation.
- Descriptions of transactions that may be used to activate licenses.

For an introduction to using trusted storage and activation transactions on a license server, see the next chapter, Transactions to a License Server.

Introduction

Activation is a method for installing license rights into trusted storage on a system. This results in license rights being held in *trusted storage* using *fulfillment records*. Fulfillment records store *feature definition lines* that define the license rights for one or more FlexEnabled applications. Feature definition lines are identical in structure to those found in license files, except that certain values can be overridden by parameters defined at the fulfillment-record level. The fulfillment record also contains details about the activation transaction that added it to trusted storage.

Additionally, a fulfillment record installed on a license server contains details about how the license server can use the licenses. For example, these details can define the licenses as *concurrent* licenses, *activatable* licenses, or licenses that are *transferable* to other license servers. *Activatable* licenses are licenses that can be activated to a FlexEnabled client using an activation transaction. These types of licenses are described in more detail in the section, Transactions to a License Server, found later in this chapter.

Trusted storage is a secure storage area controlled by FlexNet Publisher Licensing Toolkit components and used to store fulfillment records and other license-related information. The later section Trusted Storage describes this type of storage in more detail.

Description of the Trusted Storage–Based Transactions

A software publisher can use the FlexNet Publisher toolkit to build publisher-specific components (activation utility, FlexEnabled application, or license server) that perform transactions to manage license rights in trusted storage. These transactions include the following:

- **Activation**—Loads license rights into trusted storage. If trusted storage has not been initialized on the system, this transaction also performs the initialization.
- **Return**—Returns license rights from trusted storage to the originating system.
- **Repair**—Required when changes are detected that might indicate trusted-storage tampering. In this case, the license rights are considered untrusted and cannot be used until they are repaired.
- **Transfer**—Transfers license rights from one license server to another license server.

The later chapters, Transactions with a FlexEnabled Client and Transactions to a License Server, provide more information about these transactions

Systems Involved in an Activation Transaction

To perform an activation transaction, you need an activation client (a FlexEnabled client or FlexNet license server) that communicates with a publisher's activation server or a license server or accesses a locally loaded Activation Specification Record (ASR). The transaction is enacted using the following possible system combinations:

- **License Server to FlexEnabled client**—The transaction to activate licenses from a license server to a FlexEnabled client is performed across the *local network*.
- **Activation server to FlexEnabled client**—The transaction to activate licenses from a *FlexNet Operations server* (or a publisher's custom activation server) to a FlexEnabled client is performed across the Internet.
- **ASR file to license server or FlexEnabled client**—The transaction to activate licenses on a FlexNet license server or a FlexEnabled client is performed by processing an *activation specification record* (ASR) directly on the license server or FlexEnabled client machine. This process is called *local activation*.
- **Activation server to license server**—The transaction to activate licenses from a *FlexNet Operations server* (or a publisher's custom activation server) to a license server is performed across the Internet. These license rights can be either *concurrent* (checked out by a FlexEnabled client) or *activatable* (activated to a FlexEnabled client).
- **License server to license server**—The transaction to activate licenses from one *license server* to another license server is performed across the *local network*. This is called a *transfer* of license rights between license servers. License rights are deactivated on one and then activated on the other.

License rights can be returned to the system that supplied them using a *return* transaction. A *repair* transaction is used to repair a fulfillment record when the trusted storage security system detects a change possibly due to tampering with trusted storage. Because of the nature of the configuration, you cannot return license rights to an ASR.

Choosing an Activation Mechanism

You can choose from different transaction mechanisms to activate license rights to trusted storage. These mechanisms include the following:

- **Programmatic activation**—This process sends request and response XML messages as a data stream between the two systems during the course of the transaction. The process generally does not require user involvement to complete the transaction.
- **Manual activation**—A method is similar to programmatic activation, but the request and response XML messages are saved to files, rather than sent seamlessly as data streams across a network connection. This method generally requires user involvement to complete the transaction. You can use the Manual Response Generator to process the content of these XML files. However, you should reserve the Manual Response Generator for testing purposes only; do not use it in a production environment.
- **Short-code activation**—This mechanism allows you to send request and response messages as numeric codes (called request codes and response codes), rather than XML messages. It is useful when an internet connection is not available to the user or when the XML files are too large to send. This mechanism generally requires user involvement to complete the transaction. You can use the Manual Response Generator to generate and process request codes and response codes. However, you should reserve the Manual Response Generator for testing purposes only; do not use it in a production environment.
- **Local activation**—This mechanism allows you to load license rights from a local ASR file. It is used for trials or for performing activations when an internet connection is not available to the user, XML files are too large to send, or activations are required in high-security environments. Depending on how the business process is implemented, this mechanism might require user intervention to complete the transaction.

Activation Transaction Types

FlexNet Publisher supports two transaction types:

- **Composite Transactions**—From FlexNet Publisher 11.14.0, composite transactions are fully supported for all servers. From this version onwards, it is best practice to use composite transactions for all interactions with servers.
- **Single-Action Transactions**—Single-action transactions perform one activation action per transaction. For example, using single-action transactions to activate two fulfillment records from a license server to a FlexEnabled client involves issuing two separate transactions.

  From FlexNet Publisher version 11.14.0, single-action transactions have been deprecated.

Composite Transactions

Composite transactions are for use between an activation client (such as a enterprise license server or FlexEnabled client) and an activation server (such as FlexNet Operations or a customized activation server), or between the FlexEnabled client and the enterprise license server.

A composite transaction uses one request from the activation client and one response from the activation server to perform multiple activations, repairs, and returns. This type of transaction gives the activation server flexibility in managing trusted-storage license rights. For example, a single composite transaction can do any combination of the following actions:

- Delete (return) one or more fulfillment records
- Create (activate) one or more fulfillment records
- Modify one or more fulfillment records (for example to change the license count)
- Combinations of these actions to perform partial or full returns or upgrades

Composite Transaction Features

Composite transactions provide the following features:

- Support complex licensing operations, such as upgrades (return one fulfillment, activate another) or multiple activations (more than one fulfillment) in a single request-response exchange with the server.
- Provide additional information in the request, such as:
  - Details of fulfillments that already exist on the requesting machine
  - Error information that enables recovery if a response cannot be processed successfully by the requesting machine (potentially avoiding license loss)
  - The trusted storage serial number and request timestamp (potentially useful for fraud detection)
- Provide more secure activations, returns, and repairs on FlexNet license servers or FlexEnabled clients that run on virtual machines.

Composite-Transaction Mechanisms and APIs

The mechanisms of composite transactions are as follows:

- The activation application uses API calls to create a request
- The request is sent to the activation server, either online or offline
- The activation server analyzes the request, applies its business rules, and creates a response
- The response is sent back to the activation client, which uses API calls to process and apply the trusted storage update

APIs and Attributes Specific to Composite Transactions

Specific API objects and attributes are required to support multiple actions within the transaction, to store details about existing fulfillments within trusted storage, and to support recovery should response-processing errors occur. Also, the API must support comprehensive dictionaries that provide additional vendor information and information specific to FlexNet Operations.

For this reason, and to clearly distinguish between the deprecated single-action transactions and composite transactions, a separate set of API calls are defined for activation utilities that support composite transactions. These have the function name prefix `fnpTran-` and are defined in the *C/C++ Function Reference*, as well as in the header file `flxActTransaction.h`.

Sample Utilities and Files to Demonstrate Composite Transactions

Specific sample activation utilities are provided to demonstrate the use of composite transactions. The source code for these utilities can be found in the folder `<install_dir>/examples/activation/comptranutil`. The FlexNet Publisher toolkit file `makefile.act` uses this single source to build two executables:

- `appcomptranutil` (application activation utility for FlexNet clients)
- `servercomptranutil` (server activation utility for license servers)

Additionally, the toolkit contains examples of response parameter files that the Manual Response Generator can use to generate responses. These response parameter files, located in `<install_dir>/<platform_dir>/publisher/xml/Version6/W7_*`, illustrate use cases for composite transactions.

Composite Transaction Example

This example demonstrates how a composite transaction uses a single request and response to perform multiple activations on a license server.

Note • *This example automatically deploys the sample prepped Trusted Configuration (*`DemoTrustedConfigOut.xml`*) that ships with certain toolkits. The sample activation utility* `appcomptranutil` *invoked in the example uses this Trusted Configuration to set up a trusted section in trusted storage for a first-time activation. For more information about the prepped Trusted Configuration, see the "First Build of the FlexNet Publisher Toolkit" chapter in the Development Environment Guide.*

Task ***To run a sample composite transaction that activates two fulfillment records***

1. At a command prompt from the toolkit's *`<install_dir>/<platform_dir>`* directory, enter the following command to remove any server-side trusted storage files that might have been created during previous demonstration or trial-licensing activities:

   ```
   tsreset_svr -delete all
   ```

2. Run the command to create a request for two activations:

   ```
   servercomptranutil -new request.xml -activate ENTL-EZCALC-S1 -activate ENTL-EZCALC-NS1
   ```

   Here, ENTL-EZCALC-S1 and ENTL-EZCALC-NS1 are the two Rights IDs provided by the publisher's activation server to authorize the two activations. The request is written to `request.xml`.

3. Enter the following command to have the sample Manual Response Generator create the response. (In practice, the activation server performs this step).

   ```
   ./publisher/responsegen -i request.xml
   -o response.xml
   -k ./publisher/xml/DemoKeyout.xml
   -m ./publisher/xml/Version6/W7_EZCalcSvrSingleStepMultipleActivate.xml
   ```

   Here, `W7_EZCalcSvrSingleStepMultipleActivate.xml` is the response parameter file. It contains the actions needed for the activation (in this case, one action for each activation). The response is written to `response.xml`.

4. Enter this command to process the response:

   ```
   servercomptranutil -process response.xml
   ```

   The utility displays the details of the action taken.

5. To view license rights in trusted storage, run the following command:

   ```
   servercomptrantuil -v format=long
   ```

Components Used With Trusted Storage–Based Licensing

The following diagram shows the components required to implement trusted storage–based licensing.

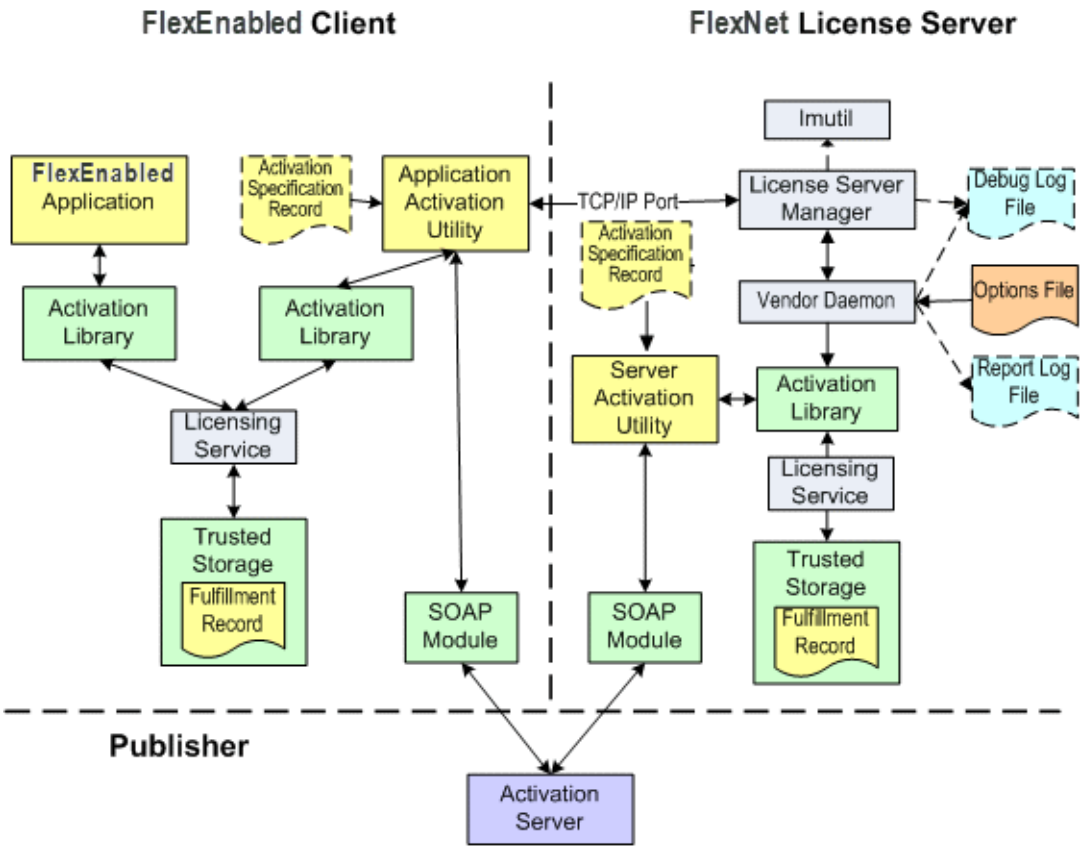


Figure 1-1: Components Used in Trusted Storage–Based Licensing

This diagram includes two instances of the activation libraries on the FlexEnabled client. Similarly, there are two activation libraries on the license server but these are not shown on the diagram. For `notr` toolkits, there are single activation libraries on the FlexEnabled client and the license server.

The toolkit also includes the Manual Response Generator (not shown in this diagram) that generates responses to activation, return, and repair requests and is used for testing only. For details, see The Manual Response Generator.

Application Activation Utility

The software publisher is responsible for providing the executable for the activation utility that runs on the FlexNet client (that is, the machine on which the FlexNet application is installed). This client-side, or *application*, activation utility uses the API in the activation library to initiate and manage activation transactions that install or manage license rights for the FlexEnabled application from the publisher's activation server or the enterprise license server.

The FlexNet Publisher Licensing Toolkit includes sample source code that you can use to create the application activation utility that processes local activations, composite transactions, and short-code transactions. The source code is found in files `compTranUtil*.c`, located in `<install_dir>\examples\activation\comptranutil`.

Additionally, the toolkit contains a sample working executable and its associated activation library, called `appcomptranutil.*` and `appcomptranutil_libFNP.*`, for issuing and processing composite transactions.

Although an application activation utility is shown as a separate component on the diagram in Figure 1-1 (and is supplied as stand-alone sample code in the toolkit), the utility can be built into the FlexEnabled application. Building the utility into the FlexEnabled application has advantages. For example, before the end user checks out a feature, the utility can issue the Flexible API function **lc_expire_days** to retrieve the remaining length of a license. The utility can then send a message to the end user when a license is about to expire, giving the user the option of initiating an activation before the license expires.

Utility Supporting Single-Action Transactions (Legacy)

Although activation can still be performed using single-action transactions, best practice is to use composite transactions, which offer a more secure activation process.

The toolkit contains sample source code, located in `<install_dir>\examples\activation\appActUtil.c`, which can be used to create application activation utilities that process local activations and single-action transactions.

The toolkit also contains the following sample working application activation utilities and their associated activation libraries:

- `appactutil.*` and `appactutil_libFNP.*` for issuing and processing ASRs and single-action transactions (standard toolkits)
- `appactutil.*` and `demo_client_libFNP_notr.*` (`notr` toolkits)

Server Activation Utility

The software publisher is responsible for providing the executable for the activation utility that runs on the enterprise license server. This *server* activation utility uses the API in the activation library to initiate and process activation transactions that install or manage license rights on a license server from the publisher's activation server or another enterprise license server.

The FlexNet Publisher Licensing Toolkit includes sample source code that you can use to create the server activation utility that processes local activations, composite transactions, and short-code transactions. The source code is found in files `compTranUtil*.c`, located in `<install_dir>\examples\activation\comptranutil`.

Additionally, the toolkit contains a sample working activation utility and its associated activation library, called `servercomptranutil.*` and `servercomptranutil_libFNP.*`, for issuing and generating composite transactions.

For instructions on using the sample server activation utilities, read *Getting Started with Trusted Storage–Based Licensing*.

Utility Supporting Single-Action Transactions (Legacy)

Although activation can still be performed using single-action transactions, best practice is to use composite transactions, which offer a more secure activation process.

The toolkit contains sample source code, located in `<install_dir>\examples\activation\serverActUtil.c`, which can be used to create server activation utilities that process local activations and single-action activation transactions.

The toolkit also contains the following sample working executables and their associated activation libraries:

- `serveractutil.*` and `serveractutil_libFNP.*` for issuing and processing ASRs and single-action transactions (standard toolkits)

- `serveractutil.*` and `demo_server_libFNP_notr.*` (`notr` toolkits)

Activation Specification Record

This file is used with *local activation* and *short-code activation*. The information normally included in the activation response is held in the activation specification record (ASR) and written to trusted storage during the activation transaction. The activation specification record includes the following information:

- Fulfillment record to be installed
- (Optional) Trusted Configuration data required to create or identify the section of trusted storage in which the fulfillment record is to be stored

  Note • *If the platform supports prepped Trusted Configurations, best practice is not to include the Trusted Configuration in the ASR. Instead, implement the prepped Trusted Configuration that is automatically deployed by your activation utility to create the initial trusted section whenever needed. (ASRs created for activations on a license server must use prepped Trusted Configuration; an activation fails if the ASR contains an embedded Trusted Configuration.) For more information about setting up the prepped Trusted Configuration, see the "First Build of the FlexNet Publisher Toolkit" chapter in the Development Environment Guide.*

- (Optional) For local activations, restrictions on when a license can be created
- For Short-Code Transactions, the short-code alias, short-code scheme ID, and optional overrides

Multiple ASRs can exist on a license server or a FlexEnabled client.

FlexNet Licensing Service

The FlexNet Licensing Service is an administrative proxy that runs on behalf of the user to perform system tasks where elevated privilege is required.

For trusted storage–based licensing, the FlexNet Licensing Service must be installed to enable the FlexEnabled application, the activation utilities, and the vendor daemon to operate correctly.

In addition, the FlexNet Licensing Service is required to enable the FlexEnabled application and the vendor daemon to access virtualization data (on Windows and Linux platforms only). The FlexNet Licensing Service regularly polls the system to update virtualization data. Therefore, the FlexNet Licensing Service runs continuously. The virtualization poll interval is not configurable.

For installation information of the FlexNet Licensing Service, see Chapter 6, "Initializing Trusted Storage."

Activation Library

An activation library handles trusted-storage access and activation tasks for the specific component (activation utility, toolkit executable, or FlexEnabled application) for which the library was generated. It provides external activation API used to any of the following:

- Initiate and manage activations, including local activations.
- Access fulfillment records held in trusted storage.

The activation API functions are described in *C/C++ Function Reference*.

The activation library has two modes of operation:

- As a *client* when it accesses client-side trusted storage (that is, trusted storage located on the FlexEnabled client machine and used to store licenses for the for a FlexEnabled application). This library and its corresponding FlexNet Publisher component (utility, toolkit executable, or FlexEnabled application) must also reside on the client machine.
- As a *server* when it accesses server-side trusted storage (that is, trusted storage located on the FlexNet enterprise license server and used to store licenses served to FlexEnabled applications). This library and its corresponding FlexNet Publisher component (utility, toolkit executable, or FlexEnabled application) must also reside on the license server machine.

These two modes of operation create separate trusted storage files and require the use of specific functions.

For the standard version of the toolkit, you generate a product-specific activation library using the `preptool` utility provided within the toolkit. For further information, see "Prepping the Application" in *Development Environment Guide* and preptool.

Note • *Each FlexNet Publisher component that uses the standard version of the activation library is built with its own copy of the activation library. See Figure 1-1 that illustrates this for the FlexEnabled client.*

For the `notr` toolkit, you use the appropriate activation library that you created when building the toolkit:

- `demo_client_libFNP_notr.*` (for use on a FlexEnabled client)
- `demo_server_libFNP_notr.*` (for use on a license server to access server-side trusted storage)

Trusted Storage

Trusted storage is a secure storage area that is controlled by FlexNet Publisher Licensing Toolkit components and used to store licensing data. Trusted storage can be one of two types:

- **Server-side trusted storage**—Holds license rights that are checked out by FlexEnabled applications running on client systems or that are activated to client-side trusted storage residing on FlexEnabled clients.
- **Client-side trusted storage**—Holds license rights that are used by FlexEnabled applications running on the client system.

In the activation library, certain functions access client-side trusted storage only and others access server-side trusted storage only. See Activation Library for details.

The trusted storage data store can be segmented into trusted sections, each with different security settings. Security for a specific trusted section is configured using a Trusted Configuration File. Starting in version 11.10.1, a prepped Trusted Configuration is supported on certain platforms, enabling the initial trusted section in trusted storage to be established automatically before an activation request is actually sent.

A backup copy of trusted storage is maintained by FlexEnabled applications created with toolkits of version 11.6.1 or greater and that use trusted storage. The backup trusted storage file is used to restore damaged sections of trusted storage. The use of the trusted storage backup file to restore a damaged section is totally transparent.

Trusted Configuration File

The Trusted Configuration file defines the configuration for a publisher-specific segment of trusted storage, called a *trusted section*. The Trusted Configuration file defines the following for the section:

- Anchors
- Binding used to lock the trusted storage to the system
- Transaction keys used to secure the contents of activation transactions
- Wind-back tolerance used to identify whether the system clock time has been turned back

Each trusted section is identified by a Trusted ID. The transaction keys, used to secure the contents of activation transactions, are also generated per Trusted ID. Thus, you should use application-specific Trusted IDs for high-value products.

The security settings used for trusted storage and activation transactions define the security of the license rights. See Configuring Security for Trusted Storage for more information.

Note • *Trusted-storage security is not synonymous with the security of the FlexEnabled application, which also depends on how securely the components are bound to the application.*

Fulfillment Record

A fulfillment record is placed in trusted storage as a result of an activation transaction. The record contains the license rights as well as information about the transaction itself.

Feature Definition Line

A fulfillment record contains one or more feature definition lines. Each feature definition line identifies certain license rights and characteristics. The following is an example feature definition line in a fulfillment record. It uses the same syntax and structure as a feature definition line in a license file.

```
FEATURE add vendor 2.0 31-dec-2020 1 SIGN="003D B062 54D1 47E3 1FF6 145F \
59CE 0200 8002 FE69 0FBC F4FF AA10 FFB3 6EB3"
```

When used in trusted storage, the feature definition lines alone do not define license rights. Both the feature definition lines and the properties of the fulfillment record define the license rights for a FlexEnabled application. Certain attributes of the fulfillment record override keywords in a feature definition line. For example, an expiration date or field count defined at the fulfillment-record level overrides this same attribute defined in any feature definition line.

For a description of the feature definition line syntax, see Defining Feature Definition Lines for a Fulfillment Record.

SOAP Module

The connection to the FlexNet Operations server uses Simple Object Access Protocol (SOAP) over an HTTP or HTTPS connection. The SOAP module, `FNPCommsSoap.*`, is required when your activation utility initiates and manages activation transactions with an FlexNet Operations server or a publisher's activation server using this online connection. The SOAP module supports HTTP 1.0 and HTTP 1.1.

Activation Server

The activation server is not part of the FlexNet Publisher Licensing Toolkit, but the publisher needs this server to activate licenses in trusted storage on a customer's FlexEnabled client or license server. Generally, the activation server resides in the software publisher's network and responds to activation, repair, and return requests from activation utilities (on the license server or the FlexEnabled client). The activation server can be a FlexNet Operations server or a custom activation server.

When activation transactions with an activation client (a FlexNet enterprise license server or a FlexEnabled client) must be sent across a firewall, the publisher can configure a proxy server that sends client requests to the activation server.

Note • *The FlexNet Publisher Licensing Toolkit contains a Manual Response Generator (*`responsgen`*) that you can use to simulate a basic activation server for testing purposes only.*

License Server

FlexNet Publisher contains components that allow publishers to build and distribute an enterprise license server to their customers. The license server is used when the licensing strategy includes the concurrent license model. The license server manages license rights as FlexEnabled applications connect to the license server to check out and check in features.

A single license server can manage licenses from multiple software publishers. Each software publisher builds and ships certain components that are specific to their licensing strategy.

A license server set up for trusted storage–based licensing comprises the following components:

- License server manager (`lmadmin` or `lmgrd`)
- Vendor daemon
- Debug log
- Report log
- Options file
- License server utilities
- Server-side activation utility

Minimally, the license server must have at one license server manager (`lmadmin` or `lmgrd`), vendor daemon, and activation utility to function.

The following features are not supported with trusted storage–based licensing:

- Common Vendor Daemon
- Three-server redundancy

License Server Manager

The license server manager (`lmadmin` or `lmgrd`) manages vendor daemons and handles the initial request from FlexEnabled applications. After the license server manager identifies the appropriate vendor daemon for a FlexEnabled application request, it forwards the request to that vendor daemon. From then on, the vendor daemon and the FlexEnabled application communicate directly.

lmadmin

For information on `lmadmin` and its requirements for trusted storage–based licensing, see Chapter 2, "Transactions to a License Server." Detailed information about `lmadmin` is also available from its Online help system and from the *License Administration Guide*, which can be downloaded from the Flexera web site.

lmgrd

An older version of the license server manager, `lmgrd`, is still available. For information on `lmgrd`, see the *License Administration Guide*.

Vendor Daemon

The vendor daemon is an executable that is built by the software publisher. The daemon is responsible for managing the license rights for that publisher and handling check-out and check-in requests from FlexEnabled applications.

The vendor daemon communicates with FlexEnabled applications using the FlexNet communication protocol available in the client library (often called `FlexNet comms`). This communication protocol uses TCP/IP to transmit messages between the two systems.

When the FlexEnabled application checks out or checks in a feature, the vendor daemon records these actions to the report log. Additionally, the vendor daemon writes informational and error messages to the debug log.

For information about customizing the vendor daemon, see Chapter 7, Customizing the Vendor Daemon.

License Server Utilities (lmutil)

The FlexNet Publisher Licensing Toolkit includes a set of command-line utilities to help license administrators manage licenses and license servers. The utilities are packaged as a single executable called `lmutil`. You run these utilities using the following command:

```
lmutil <utility name> <options>
```

where

- `lmutil` is the name of the executable.
- `<utility name>` is the name of the specific utility you want to execute.
- `<option>` includes any command-line options to use with that utility. Each utility supports its own set of options.

Always use the newest version of the license server utilities. These utilities can be downloaded from `www.flexerasoftware.com`. See the *License Administration Guide* for more information about these utilities.

Debug Log

A debug log file contains status and error messages useful for debugging the license server. The debug log contains information about license server manager (`lmadmin` or `lmgrd`) events and vendor daemon events.

Report Log

The report log contains information about feature usage and is generated by the vendor daemon. The content of the report log is encrypted; only the FlexNet Manager product can read this file, enabling license administrators to produce license-usage reports.

Options File

The Options file allows the license administrator to define parameters that support a publisher's licensing policy. These parameters can do the following:

- Allow or deny features for specific users, hosts, or groups
- Reserve licenses for specific users, hosts, or groups
- Restrict the number of licenses available to the enterprise
- Control the amount of information logged about license usage
- Enable logging to a report log

Note • *If there are no feature lines in the license file (as expected for a vendor daemon serving licenses only from trusted storage) and an options file is used, then the path to both the vendor daemon and the options file must be separately specified on the VENDOR line in the license file.*

FlexEnabled Applications

A FlexEnabled application is a software application that implements the licensing strategy using the FlexNet Publisher Licensing Toolkit components. A FlexEnabled application uses the Flexible API (from the client library) to locate and access license rights for the application. The application developer uses the checkout function to request a feature. If the feature is available to the application (based on the license model), the checkout succeeds and the application continues normally. If the feature is unavailable, the application developer should implement certain user-friendly behavior in the application to notify the user.

Note • *When FlexNet Operations FlexEnabled Client is activated ensure that the **Max Transfer** option is set to a non-zero count. The MAX Transfer option of FlexNet Operations On Premises is covered in the following topic `Packaging Products > License Models > Creating a License Model` of the help library.*

License Activation in a Virtual Environment

FlexNet Publisher provides the following support for trusted-storage licensing in a virtual environment:

- The ability to allow or deny a local activation (via an ASR) on FlexEnabled clients running in virtual environments
- Activations, repairs, and returns performed from the publisher's activation server to FlexEnabled clients or FlexNet license servers running in virtual environments

FlexNet Publisher uses either composite or single-action transactions to perform activation, repair, and return operations from the publisher's activation server to license servers or clients running in virtual environments. However, best practice is to use composite transactions, which offer a more secure activation processes.

For more information about composite and single-action transactions, see Activation Transaction Types. For information about the hypervisors that FlexNet Publisher supports, see the current *FlexNet Publisher Release Notes*.

Features Added or Updated to Support Virtualization

The following provides an overview of the functionality that has been added or updated to support trusted-storage licensing in a virtual environment:

Table 1-1 • Features Supporting Trusted-Storage Licensing in Virtual Environments

Feature to Support Virtualization	Where to Go for More Information
New activation-utility function called to detect a virtual environment	See the description of **flxActCommonVirtualStatusGet** in the *C/C++ Function Reference*.
New ASR data dictionary element to support virtualization policies	See the section Support for Trials in Virtual and Non-Virtual Environments in Chapter 4, "Implementing Trial Licenses and Trial Packs."
New data items in the request to identify the requester's virtual environment	See the section Additional Composite Request Data Items for Virtual Environments in Chapter 12, "Building an Activation Utility for Composite Transactions." ***Note*** • *FlexNet Operations 12.8 or a compatible license generator is needed to support composite-transaction requests generated by FlexNet Publisher 11.10 for virtual machines.*
VMID-only binding in a virtual environment	See the section Binding in Appendix A, "XML Schemas."
UMN3 added for requester verification	See the section Information Used to Identify the System Initiating the Request in Chapter 1, "Overview of Trusted Storage–Based Licensing."
New or updated sample files	See Appendix A, "XML Schemas."

2

Transactions to a License Server

This chapter expands the information given in the previous chapter for license rights held in trusted storage on a FlexNet enterprise license server. This chapter provides the following:

- An introduction to activation transactions available to license servers and how license rights held in trusted storage on a license server can be used
- Outline descriptions of activation transactions that can be used on license servers
- A description of how license rights are represented in fulfillment records on license servers

Introduction

The license rights held in trusted storage on a license server are obtained from FlexNet Operations server, a publisher's custom activation server, or from another license server. These license rights are used in the following ways:

- To serve *concurrent* licenses to FlexEnabled clients via the enterprise network. These licenses operate in the same way as *floating counted* licenses from a license file.
- To activate licenses to FlexEnabled clients using activation transactions across the enterprise network. The FlexEnabled client requests a license using an activation transaction. These licenses are referred to as *activatable* licenses.
- To *transfer* to other license servers that are using trusted storage. The license server that requires the license rights uses an activation transaction to request the licenses. Both concurrent and activatable licenses can be transferred in this way.

License rights held on the license server can be returned to the system that supplied them using a *return* transaction. Therefore, license rights that a license server obtained from the publisher's activation server can be returned to the publisher. Likewise, the license server also accepts the return of license rights that it has originally supplied to a FlexEnabled client or another license server. License rights that are returned to a license server are held in trusted storage and are available for use by another FlexEnabled client or license server. Thus, using activation and return transactions, license rights can be transferred around an enterprise.

The license server both initiates repair transactions when its own trusted storage requires repair and handles requests for repair of fulfillment records that it has supplied. When the license server is responding to a repair request, it uses repair rules contained within the original fulfillment record.

Components Required on the License Server

The following components are required on the license server to support activation. For descriptions of these components, see Components Used With Trusted Storage–Based Licensing in Chapter 1, "Overview of Trusted Storage–Based Licensing." Make sure you also read the *Development Environment Guide* for information about setting up the development environment, building license server executables, and packaging all related components.

- A publisher-specific vendor daemon, built using the FlexNet Publisher Licensing Toolkit. See *Development Environment Guide*.
- A license server manager, either `lmadmin` or `lmgrd`. `lmadmin` reports the status of all licenses held in trusted storage—*concurrent*, *activatable*, and *hybrid*. `lmgrd` reports the status of concurrent licenses only.
- A server activation utility to initiate and manage the activation of licenses in its own trusted storage or in trusted storage on a FlexEnabled client.

  For information about building a server activation utility, see Chapter 11, "Building a Server Activation Utility for Single-Action Transactions," or Chapter 12, "Building an Activation Utility for Composite Transactions." Additionally, the toolkit provides a sample of each server activation-utility type.
- For standard toolkits, the vendor daemon is processed using `preptool` and the resulting activation library component (called `<vendor name>_libFNP.dll`) should be added to the product package. This step is included in the `makefile.act` processing instructions. For more information about the `preptool`, see Chapter 13, "Working with the Software Publisher Utilities.".
- For standard toolkits, the server activation utility should be processed using `preptool` and the resulting activation library component added to the product package. For more information about `preptool`, see Chapter 13, "Working with the Software Publisher Utilities.".
- For `notr` toolkits, the activation library component, `demo_server_libFNP_notr.*`, generated when the toolkit is built, is added to the product package. See *Development Environment Guide*.
- Trusted storage must have been configured on the customer's machine.
  - On Windows, configuration for trusted storage involves initializing the trusted storage directory and installing the FlexNet Licensing Service using `FNP_Act_Installer.dll`.
  - On UNIX, configuration for trusted storage involves initializing the trusted storage directory using `install_fnp.sh`. Additionally, on OS X and Linux, the FlexNet Licensing Service is installed. For additional information, see Trusted Storage Directory Initialization (UNIX Platforms).
- Optionally, if the internet is used for communications to an activation server, the SOAP module, `FnpCommsSoap.*`, is added to the product package.

Summary of the Activation Process

The following basic steps are used in an activation transaction where the license server is the activation client. That is, the license server is the system requesting licenses from the activation server. The activation server can be either the publisher's activation server (FlexNet Operations or a custom activation server) or another license server.

1. The server activation utility uses the activation API to generate a request (XML message) for certain license rights. As part of this process, the API does the following to handle the publisher-specific section of trusted storage affected by the activation:
   - If necessary, creates the trusted storage file on the license server and then either creates the initial trusted section using a prepped Trusted Configuration or specifies the need for this trusted section in the request.
   - If an additional section is required in existing trusted storage, specifies the need for this section in the request.
   - Inserts details of all the current trusted sections of *server-side* trusted storage for this software publisher in the activation request.

   Note • *The prepped Trusted Configuration feature was introduced in FlexNet Publisher 11.10.1 for certain platforms. With this feature enabled, the server activation utility automatically creates the initial trusted section when it generates the first activation request. In this way, the trusted section is already in place when the transaction is sent. For versions prior to 11.10.1, the initial trusted section configuration is a separate transaction step.*

2. The server activation utility sends the request to the activation server, using one of these methods:
   - Sends the activation request as a data stream across the network connection to the activation server. (This method requires a SOAP connection.)
   - Writes the activation request to an XML file, which is then manually transported to the activation server (for example, by email).
3. The activation server validates the activation request against certain business rules and determines whether the activation request can be granted.
4. If the activation request complies with the activation server's business rules, the activation server generates an activation response in the form of an XML message. This message is either transmitted as a data stream across a network connection or written to a file and transported to the license server.
5. The server activation utility loads the license rights into trusted storage on the license server.

   Note • *Even though the prepped Trusted Configuration might be in place when sending an activation request, the activation server can choose to update the configuration of a trusted section in its response. FlexNet Operations responses, for example, normally update the configuration of the initial trusted section even though the section was previously created using the prepped Trusted Configuration.*

Activation Using the Publisher's Activation Server

The server activation utility initiates a composite activation transaction using the functions within the activation API. Either of these methods are used to transfer the transaction data between the license server and the publisher's activation server (such as a FlexNet Operations server or a custom activation server):

- XML messages sent as data streams across an internet connection using the SOAP protocol
- XML messages sent as files between the activation server and the license server (for example, using email)

Best Practice

Basically, for licensing life-cycle operations (for example, activations, returns, and repairs) that involve the publisher's activation server, best practice is to use composite transactions, not single-action transactions. Composite transactions can perform multiple actions in a single transaction and provide a higher level of security than single-action transactions. Additionally, unlike single-action transactions, composite transactions support UMN3, the preferred machine identifier in virtual environments. For descriptions of API functions you can use for composite transactions and for more information about UMN3, see Chapter 12, "Building an Activation Utility for Composite Transactions."

Transfer of Licenses Between License Servers

Licenses held in trusted storage on a license server can be transferred to other license servers. An activation transaction is used to transfer licenses between two license servers in the local network. Best practice is to use composite transactions which enable you to perform multiple actions in a single transaction. If legacy single-action transactions are used, and if the prepped Trusted Configuration feature is not enabled, two transactions are needed to configure a trusted section in trusted storage and then to perform the actual activation. (For more information about the prepped Trusted Configuration, see the chapter "First Build of the FlexNet Publisher Toolkit" in FlexNet Publisher's *Development Environment Guide*. For information about creating a server activation utility, see Chapter 12, "Building an Activation Utility for Composite Transactions.")

When a license server acts as an activation server, the operation of the activation server is completely automatic. It is controlled by data held in the relevant fulfillment records in server-side trusted storage and logic built into the vendor daemon.

Transfers are initiated by the server activation utility on the license server requesting the licenses (that is, the activation client). This utility uses the FlexNet communication protocol (`FlexNet comms`) to communicate with the license server issuing the license rights (that is, acting as the activation server). The utility generates and sends the activation request to this license server. The request includes the following information:

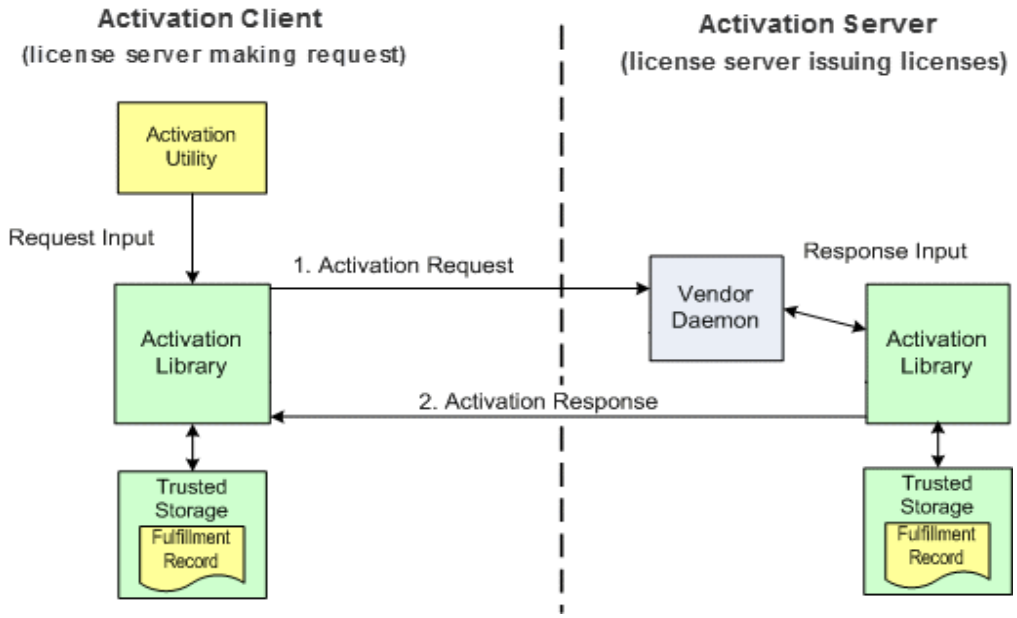
- EntitlementID value
- Expiry date
- Requested counts for the each of the three license groups held in trusted storage - activatable, concurrent and hybrid

The license server acting as activation server, in turn, sends the appropriate fulfillment record back to the requesting license server. The fulfillment record is sent only when all the following conditions are true:

- A fulfillment record exists with a matching entitlementID and an expiry date equal to or later than the requested expiry date.
- The hops element in the fulfillment record is greater than zero. See the next section Limiting the Number of Transfers for more information about hops.
- Sufficient licenses available; concurrent and hybrid licenses that are in use cannot be transferred. For more information about license counts, see the section FulfillmentDictionary Data in Server-side Trusted Storage.

The following diagram illustrates the activation transaction process between two license servers. The picture shows how, when a license server acts as an activation server, it uses fulfillment records within its own trusted storage to respond to activation requests from another license server.

Figure 2-1: Activation Transaction Process Between License Servers



Limiting the Number of Transfers

Limitations on the mobility of license rights are required for the following reasons:

- Enable easy tracing of the location of license rights.
- Reduce possibility of errors due to repair of trusted storage.

The transfer of license rights as fulfillment records can be limited by the use of the *hops* element in a fulfillment record. The hops element specifies the number of times a fulfillment record can be transferred. When license rights are transferred, the fulfillment record sent to the destination license server has a hops element value of one less than the value in the source fulfillment record. So, as license rights are successively transferred, the hops element is reduced until it is zero. A fulfillment record with zero hops cannot be transferred.

When license rights are returned, the hops element is restored to its original value.

For FNO users, the hops element is set using the **Max Transfers** attribute of the license model. Set the attribute to a non zero count.

A fulfillment record activated on a FlexEnabled client effectively has zero hops; it cannot be transferred. It can, however, be returned. See the Return Transaction section.

Local Activation Using an ASR

This type of activation does not use an activation server to install license rights on the license server. All the information required for the activation is held in a signed file, called an *activation specification record* (ASR), on the local FlexNet license-server machine.

Local activation can be used to implement license models for *trial* or *emergency* purposes. Trial license rights have limited duration and allow an end-user to test a product for a short period of time prior to purchasing it. Emergency license rights are provided via an ASR when the normal method of supplying license rights is not available (for example, when a network connection is unavailable or an activation server is temporarily out of service). Local activation also provides a solution for activating license rights on license servers in high-security environments.

For details of the API functions required for local activation, see the section Local Activation in Chapter 10, "Local Activation."

Repair Transaction

A repair is required when changes in trusted storage are detected that indicate trusted storage might no longer be intact. Any fulfillment records that are affected, and are *untrusted*, cannot be used until they are repaired. Thus, the license server cannot use the license rights contained in an untrusted fulfillment record until a repair transaction is completed. The activation API used in the server activation utility provides for the repair of fulfillment records.

Trusted-Storage Changes Due to Illegal Tampering

A trusted-storage change might be the result of an attempt to obtain license rights illegally. These attempts might include the following:

- Copying files from one system to another
- Restoring a backup that was made when trusted storage contained more licenses
- Setting the system clock to an earlier time

Trusted-Storage Changes Due to Valid Actions

A trusted-storage change might also be the result of a valid user action, such as one of the following:

- Changing a system hardware component that is used to bind trusted storage to the system
- Performing a system restore to correct system problems or roll-back changes.
- Correcting the system clock after it was set to the future, either inadvertently or while testing.

Return Transaction

A return transaction returns a fulfillment record to the source from which it was activated:

- The license server
- The FlexNet Operations server that supplied the original fulfillment record
- The publisher's custom activation server

The server activation utility initiates the return transaction. The activation server that receives the return transaction applies business rules to its processing of the return request:

- **A return to a license server**—The return is accepted if the fulfillment record was originally supplied by this license server and if incrementing the existing counts does not violate the maximum count values. The number of hops in the

fulfillment record is updated to what it was originally when the fulfillment record was originally written to trusted storage on the license server. Then the fulfillment record is written to trusted storage. For general information on hops, see Limiting the Number of Transfers.

- **A return to a FlexNet Operations server**—The business rules for accepting or rejecting a return request are configurable. See the FlexNet Operations server online help for details.
- **A return to a custom activation server**—The business rules for accepting or rejecting a return request are defined completely by the software publisher when building this application.

Note • *Composite transactions allow partial returns, but the return must be made to a publisher's activation server, not an enterprise license server.*

Description of License Rights in Server-Side Trusted Storage

This section describes license rights held in server-side trusted storage only.

Types of License Rights in Server-Side Trusted Storage

The license rights held in server-side trusted storage can be of three types:

- **Activatable**—Activated to a FlexEnabled client or another license server
- **Concurrent**—Served as concurrent licenses to a FlexEnabled client
- **Hybrid**—Used as either activatable or concurrent licenses

These license rights are not available for use by FlexEnabled applications on the license server. They are supplied to FlexEnabled applications on FlexEnabled clients or transferred to other license servers.

All *served activated* licenses are effectively locked to the license server as the fulfillment records that contain them are held in trusted storage. This statement assumes that the default settings for trusted storage configuration are used (that is, anchoring and binding are not disabled). If anchoring and binding are disabled, then trusted storage is not bound and consequently the licenses are not locked.

Served activated licenses are supplied to the FlexEnabled applications as these license types:

- **Activatable licenses**—The license server is the activation server and responds to an activation request from the application activation utility on the FlexEnabled client.
- **Concurrent licenses**—The license server holds licenses that can be served concurrently to FlexEnabled applications. Concurrent licenses are time-shared between users; once a license has been checked in by one user, it can be used by another user. Further details of its implementation within the fulfillment record are given in this section.

Feature Definition Lines for Licenses in Server-Side Trusted Storage

The same feature definition lines are used for both activatable licenses and concurrent licenses:

```
FEATURE add vendor 1.0 31-dec-2020 2 DUP_GROUP=U SIGN="<...>"
```

Items, such as DUP_GROUP, that only apply to concurrent licenses are disregarded when this feature definition line is used as an activatable license.

License Counts

Fulfillment records for served activated licenses can contain activatable licenses, concurrent licenses, and a third type, called *hybrid licenses*. A hybrid license can be used as either an activatable license or a concurrent license. The fulfillment record maintains counts for each of these license types.

Optionally, the fulfillment record also includes counts for the number of overdraft licenses that are available for each license type. In summary, the following licenses groups are potentially represented in a fulfillment record:

Table 2-1 • License Groups Represented in Fulfillment Records in Server-side Trusted Storage

License Group	Definition
Activatable	Licenses that are available for activation on FlexEnabled clients or other license servers.
Activatable Overdraft	Overdraft licenses that are available for activation on FlexEnabled clients or other license servers.
Concurrent	Available licenses that the license server can serve to FlexEnabled clients concurrently. This count is multiplied by the count within the feature definition line to produce the total number of available concurrent licenses.
Concurrent Overdraft	Overdraft licenses available as concurrent licenses. This count is multiplied by the value of the OVERDRAFT key word within the feature definition line to produce the total overdraft count.
Hybrid	Licenses that the license server can either serve to FlexEnabled clients concurrently or activate on FlexEnabled clients. Depending on how the license is used determines the number of available licenses. For activatable licenses, the specified count is the total number of available licenses. For concurrent licenses, the count is multiplied by the count within the feature definition line to produce the total available licenses.
Hybrid Overdraft	Overdraft licenses that are available either as concurrent overdraft licenses or activatable overdraft licenses. Depending on how the license is used determines the number of available licenses. For activatable licenses, the specified count is the total number of available overdrafts. For concurrent licenses, this count is multiplied by the value of the OVERDRAFT key word within the feature definition line to produce the total available overdrafts.

Important • *These license groups are used to describe license rights held in server-side trusted storage. Once a license has been activated to a FlexEnabled client (and is outside of the context of the license server), these groups have little meaning.*

The concurrent and hybrid counts are read into memory when the vendor daemon is run and used to serve concurrent licenses. They are included in the standard license pools held by the vendor daemon. Thus, the total concurrent pool for a feature is computed by adding the following items together:

- For a specific feature, count value + `OVERDRAFT`
- For fulfillment record with a concurrent license count, *`count value`*`*availConc + OVERDRAFT*availConcOD`
- Fulfillment record with a hybrid license count, *`count value`*`*AvailHyb + OVERDRAFT*availHybOD`

where:

- `Count value` is the value for the count field of the feature definition line.
- `OVERDRAFT` is the value set for `OVERDRAFT` in the feature line or feature definition line.
- `availConc` is the number of currently available concurrent licenses for the fulfillment record.
- `availHyb` is the number of currently available hybrid licenses for the fulfillment record.
- `availConcOD` or `availHybOD` is the number of currently available overdraft licenses available for concurrent or hybrid licenses, respectively, for the fulfillment record.

Note • *When the* `OVERDRAFT` *key word is not present in a FEATURE line, a default of zero is used. Thus, overdraft counts within a fulfillment record are added to the concurrent pool for feature definition lines only when* `OVERDRAFT` *has a positive value. It is recommended that feature definition lines include OVERDRAFT=1. The FlexNet Operations server sets OVERDRAFT=1 for all fulfillment records that it generates that contain positive overdraft counts.*

Managing the License Pool Counts

When the vendor daemon receives a request to activate a license, it searches through the license rights available in server-side trusted storage for a fulfillment record that can satisfy the request. The license groups are searched in the following order until a license is found that can satisfy the activation request:

1. Activatable
2. Hybrid
3. Activatable overdraft
4. Hybrid overdraft

To use a hybrid license as an activatable license, a free license must exist in the concurrent license pool held by the vendor daemon. The requested license is then deducted from this concurrent license pool and issued as an activatable license. The count of hybrid licenses is reduced by one.

If an activation is successful, a deduction record is added to the fulfillment record in server-side trusted storage. Each day at midnight, the vendor daemon reviews the deduction records and deletes those that have expired, unless they are activatable overdraft licenses. Deduction records for these license types are not deleted after they have expired and will persist. For more information about deduction records, see section titled Deduction Records.

If the user returns an activatable overdraft license before the expiration date, the license is added back to the available license pool and the deduction record is deleted. If the activatable overdraft license expires, it is not added back to the available license pool. It will always count against the activatable overdraft pool for that fulfillment ID. This behavior is not the same for other license types.

For more information about the license duration of activatable overdraft licenses, see Limiting License Duration of Activated Overdraft Licenses.

Limiting License Duration of Activated Overdraft Licenses

Overdraft licenses can be provided using activation both from the activatable license count and the hybrid license count. The duration of an overdraft license is restricted by adding a key-value pair to the fulfillment record, as shown in the following example:

```
maxOdDuration=86400
```

This value is used when calculating what expiry date to designate in the fulfillment record in an activation response that uses an overdraft license. The expiration value of the overdraft license is calculated as follows:

expiration day `=` *current time* `+ maxOdDuration`

This calculation, performed in seconds, is rounded up to the nearest whole day. Thus, if the current time is 12:00 a.m. on the January 24, and `maxOdDuration` is set to 86,400 seconds (24 hours), the overdraft license operates until end of day on January 25.

This calculated value is designated in the fulfillment record in response to an activation request except under these conditions:

- The user has requested an expiration date or duration that is shorter than the calculated value. In this case, the requested expiration date is used.
- The expiration date in the source fulfillment record is earlier than the calculated value. In this case, the source fulfillment record expiration date is used.

If the user returns an activatable overdraft license before the expiration date, the license is added back to the available activatable overdraft pool. If the activatable overdraft license expires, it will not be added back to the available license pool. It will always count against the activatable overdraft pool for that fulfillment ID. This behavior does not affect hybrid overdraft licenses.

Fulfillment Record in an Activation Response

When the license server responds to a request from a FlexEnabled client for an activatable license, the response can contain a fulfillment record, known as a local activation. For the most part, the local activation is a copy of the fulfillment record held on the license server. Any attributes of the record that are relevant to only served licenses, such as concurrent or hybrid counts, are ignored.

The fulfillment record contains the feature definition lines. The following shows a feature definition line that might be included in a fulfillment record within an activation response:

```
FEATURE add vendor 2.0 21-dec-2020 2 DUP_GROUP=U SIGN="2F5023468DD2"
```

This feature definition line is interpreted on the FlexEnabled client as a license having the following characteristics:

- For the **add** feature
- For a FlexEnabled application using *vendor* as the vendor daemon name
- For version 2.0 or earlier of the application
- With an expiration date of 21-Dec-2020, which might be overridden by the fulfillment record date, if it exists

Note • *The expiration date must be the same or earlier than the date in the source fulfillment on the license server.*

- With a count 2, which is ignored on the client
- With a `DUP_GROUP` value, which is ignored because it does not apply to local licenses

3

Configuring Security for Trusted Storage

The following chapter describes the components that enable secure access and communication to trusted storage. This section also describes how software publisher can customize the security configuration for their segment of trusted storage, called a *trusted section*.

Overview of Trusted Storage Security

Trusted storage contains license rights for FlexEnabled applications. To secure these license rights, the following security is provided for trusted storage:

- **Encryption**—The contents of trusted storage are encrypted to protected against external modification.
- **Restricted access**—Access to trusted storage is restricted. Only the activation library provides the access. FlexEnabled applications and activation utilities must use the activation library to access trusted storage.
- **Prepped Trusted Configuration**—A prepped Trusted Configuration is deployed through the activation utility at the time that a request is generated to ensure that at least one trusted section always exists within trusted storage. This prevents the copying of trusted storage between the time it is created and the time the response is processed. It also ensures that at least one identifier is sent in the request to enable the publisher's activation server to identify the requesting machine.
- **Protected communication with trusted storage**—Security is added to components that use the activation library to prevent spoofing of the activation library. This security is added when the components are prepared with the preptool (see Chapter 13, Working with the Software Publisher Utilities) and a version of the activation library is produced that uses unique keys. Slightly reduced security is provided when the `notr` toolkit is used and a single publisher specific activation library component is used.
- **System clock windback detection**—Time stamps are placed in trusted storage each time it is accessed. When windback detection is required, the current system time is checked against the time stamp in the fulfillment record.
- **System binding**—License rights are locked to a system to prevent them being transferred illegally to another system. This is referred to as *binding* and the system characteristics used for the binding are referred to as *binding identities*.
- **System anchoring**—Anchors are links to one or more system identifiers. They are used to identify if trusted storage has been deleted or overwritten. This is referred to as *anchoring*, and implies that one or more anchors are in use.

The majority of the security is automatically provided and cannot be configured or changed. The last three items (windback detection, binding, and anchoring) are configured using a *Trusted Configuration*. The following sections describe each and how they are configured.

Defining the Trusted Configuration

Windback detection, binding, and anchoring are configured using a *Trusted Configuration*. A Trusted Configuration defines a publisher-specific section of trusted storage and is used to initialize that area. Each trusted section can have its own windback detection, binding, and anchoring configuration.

The *Trusted ID* uniquely identifies the section of trusted storage and is used in both *client*-side and *server*-side trusted storage. Client-side trusted storage is created on a FlexEnabled client and is used to store license rights used directly by a FlexEnabled application running on that system. Server-side trusted storage is created on a license server and is used to store license rights that are either served to FlexEnabled applications or activated to FlexEnabled clients.

Note • *Potentially both client and server-side trusted storage can exist on a license server. This occurs when a FlexEnabled application is installed on a license server.*

The Trusted Configuration includes separate settings for client- and server-side trusted storage. You can use these separate settings to apply higher security to the server-side trusted storage to reflect the higher value placed on license rights held there. (License rights held in client-side trusted storage apply to a single FlexEnabled application running on the client machine, whereas license rights held in server-side trusted storage are served to multiple FlexEnabled clients.)

Within a Trusted Configuration, the default behavior for both binding and anchoring is to use all the available binding identities or anchors. However, you can optionally disable a binding identity or an anchor for all platforms or for a specific platform. You should only have to do this when the binding or anchor has proved to be unreliable and results in repeated breaks in trusted storage.

For a description of how to create a Trusted Configuration file, see the section O_ConfigPhaseOne.xsd in Appendix A, "XML Schemas." Additionally, best practice is to deploy a prepped Trusted Configuration to create an initial trusted section before a request is sent or an ASR is installed. For more information on how to set up for prepped Trusted Configuration deployment, see Deploying Trusted Configuration.

Windback Detection

Windback detection can be set on or off using the functions provided within the Flexible API for the FlexEnabled application. For the vendor daemon windback, detection is set when the vendor daemon is built.

Windback detection, for license rights held in trusted storage, is based on the time stamp placed in trusted storage when trusted storage was last accessed. The default is that when the current time is more than 24 hours earlier than this time stamp, windback is reported. The default *windback tolerance* for license rights held in trusted storage is thus the same as that for license rights held in license files.

When windback is detected, fulfillment records that have a limited duration (that is, duration is not permanent) are marked as untrusted. These untrusted fulfillment records cannot be used until they are repaired.

The windback tolerance can be configured when the Trusted Configuration is defined. Windback tolerance can be separately defined for any combination of the following:

- Trusted section of trusted storage

- Platform
- FlexEnabled client or license server (client-side or server-side trusted storage)

For detailed information on configuring windback tolerance, see Timing in Appendix A, "XML Schemas."

Windback tolerance is normally set so that valid reasons for resetting the clock do not trigger windback detection. Windback tolerance can be set to allow for:

- Automatic changes in the system clock for daylight saving. This applies to both FlexEnabled clients and license servers.
- Changes made to the system clock on a laptop when travelling between time zones. This is likely to be applicable to FlexEnabled clients only.

Binding

Binding is used to lock license rights to a system and prevent them being transferred illegally to another system. Binding is implemented by storing system information (called binding identities) in trusted storage. Each time trusted storage is accessed, these *binding identities* are checked and compared with the values held in trusted storage. A decision is then made, based on the relative security of the binding identities, whether a request can proceed, or trusted storage security has been compromised and needs repair.

Binding Default

The binding default is to use all available binding identities. This default ensures the following:

- **Increased binding security**—The more binding identities used, the greater the security. Because license servers generally manage more license rights than FlexEnabled clients, increased binding security is usually more important for server-side trusted storage. License servers have a lower tolerance to changes in the binding identities than do FlexEnabled clients.
- **Resilience against legitimate binding breaks**—Resilience means that legitimate system changes made by an end user (for example, a new hard-disk installation) do not require a repair to be performed as long as other binding identities remain the same.

Note • *Resilience against binding breaks is generally a bigger issue for FlexEnabled clients than it is for license servers. Servers are not often updated; and, if they are, the update is carefully planned.*

Possible Binding Identities

The possible binding identities are operating-system dependent, as shown in the following table.

Table 3-1 • Binding Identities

Identity Description	Windows	Mac	Solaris	Linux	Identity
Machine	X	X	X	X	SYSTEM
Hard disk or file system	X	X	X	X	HARDDISK
Display or graphics card	X	X	X	X	DISPLAY
Bios serial number and version	X				BIOS
CPU information: OS specific	X	X	X	X	CPU
RAM size	X	X	X	X	MEMORY
Network card address	X	X	X	X	ETHERNET
Virtual machine image identifier	X			X	VMID
GenerationID	X				GENID
TPM hostid	X				TPMID

Special Conditions Affecting Binding Identities

The following conditions affect binding identities:

- The appropriate binding identities for the environment (physical or virtual) are automatically selected at run time, for more details refer to Binding section in XML Schemas Chapter.
- If one or more binding identities change and trust is not lost (due to the availability of sufficient identities that do not change) then the values held in trusted storage are replaced with the new ones. This allows the same or other items to change again in the future without causing loss of trust.
- If a “Dealbreaker” binding identity changes, trust will be lost regardless of the number that do not change.

Anchoring

An anchor is a reference to a certain system identifier. There are two types of anchoring mechanisms available with the toolkit:

- **Publisher Anchor**—Used to anchor trusted storage to the system and to prevent the FlexEnabled application from executing if trusted storage has been deleted or overwritten. This type of anchor is enabled by default without any additional configuration and is used with all types of activation. This is the standard anchoring mechanism.

- **Trial Anchor**—Used to anchor trial licenses only. This type of anchor is not enabled by default and must be configured separately. A trial anchor ensures that users have access to a FlexEnabled application for the trial period only.

Both of these anchoring mechanisms use the same anchoring technologies. Each type of anchoring technology is supported on only certain operating systems and provides a different level of security. On some operating systems, multiple anchoring technologies are available. The default configuration is to use all anchoring technologies available for the operating system.

Note • *On Windows, changing the configuration to exclude registry anchors will result in the software being more resilient to system restore exploits.*

Table 3-2 lists the available anchoring technologies and supported operating systems. For more information about changing the anchor configuration, see O_ConfigPhaseOne.xsd in Appendix A, "XML Schemas."

Table 3-2 • Anchor Technologies

Anchoring Technology	Windows	Mac	UNIX
File-based		X	X
Registry-based	X		
Hardware-based (track zero)	X		
OS X file system based		X	

Space Limitations for Hardware-based Anchors

A Windows system can store approximately 160 anchors on the track zero sector. This limited space must be shared by both trial anchors and publisher anchors.

There is a small possibility that the track zero space will become full. When this happens, either the oldest trial anchor or the publisher anchor with the oldest 'last updated' stamp will be overwritten. This will require that the license be repaired the next time the FlexEnabled application is run.

Publisher Anchors

Publisher anchors are used to determine whether trusted storage has been deleted or overwritten. A publisher anchor is linked to the Trusted Configuration (identified by a single Trusted ID). The anchors are defined for all FlexEnabled applications with license rights assigned to that Trusted ID.

The default configuration for both publisher anchors is that it is enabled and uses all anchoring technologies available for the operating system. We do not recommend changing this behavior because it makes the software vulnerable to a hack caused by a system restore.

To change the configuration from the default, use the Trusted Configuration file. See O_ConfigPhaseOne.xsd in Chapter A, XML Schemas for more information.

Note • *Comparison with anchoring in SafeCast/ FlexNet Activation—repair only required when all anchors are broken. Thus when track zero and registry anchor are both in use, a break on track zero due to a legitimate user action (such as using a dual partitioned disk) does not result in a repair.*

Trial Anchors

Trial anchors link a trial license to the system and is intended to be used when you want to ensure that access to a FlexEnabled application continues for the trial period only. Each trial anchor is defined using a specific Trial ID. This type of anchor is available when using local activation only.

Each publisher can define up to 255 Trial IDs per Trusted Configuration (identified by a single Trusted ID) and values can range from 1 to 255. Each publisher must manage and assign Trial IDs independently and should choose a numbering convention that meets their specific business requirements. For information about creating trial licenses, see Chapter 4, "Implementing Trial Licenses and Trial Packs."

Deploying Trusted Configuration

When an activation utility is running and detects that no trusted storage exists, it automatically creates the trusted-storage file. However, to create the initial section within trusted storage in order to store fulfillment records requires a Trusted Configuration. The publisher can use one of three methods to deploy the Trusted Configuration:

- **Include it in an ASR**—When the ASR is processed, the initial trusted section is created.
- **Include it in the activation response**—When the response is processed, the initial trusted section is created.
- **Include a prepped Trusted Configuration in the activation utility itself**—If the utility detects that no section exists in trusted storage, it automatically creates the initial section before an ASR is processed or a request is sent.

Flexera recommends that publishers use this last method as standard practice so that an initial trusted section is automatically created as needed using the prepped Trusted Configuration. Publishers should use the other two methods only when creating *additional* sections in the trusted storage or when supporting platforms that do not support prepped Trusted Configuration. (See the current *FlexNet Publisher Release Notes* for the list of platforms that support prepped trusted storage.)

Note • *You cannot use the prepped Trusted Configuration method to update an existing trusted section. Such an update is performed by processing an activation response only. Also note that, if a trusted section created from a prepped Trusted Configuration is in place, FlexNet Operations still normally includes a* `Config` *action in its activation response, which updates the existing trusted section configuration.*

The Prepped Trusted Configuration

The deployment of a prepped Trusted Configuration is the expected method for activations. To implement this method, you define the configuration and specify its location when preparing the activation utility. When the activation utility is built, the configuration is embedded in the utility's code. For more information, see preptool in Chapter 13, "Working with the Software Publisher Utilities." Also see Chapter 10, Building an Application Activation Utility for Single-Action Transactions and Chapter 11, "Building a Server Activation Utility for Single-Action Transactions."

How Prepped Trusted Configuration Works

When the activation utility starts to perform a local activation or a transaction and detects no trusted-storage file on the machine, it automatically creates the file (just as it also does for the alternative methods described previously). However, using the embedded configuration, the utility also creates the trusted section within the file. This method has the following advantages:

- **Security**—Upon creation, trusted storage is immediately protected from unauthorized copying. The alternative methods of including Trusted Configuration in activation responses leave a gap of time (before the response is processed) during which trusted storage can be copied.
- **Additional machine identification**—Machines are normally identified in activation requests by one or more unique machine identifiers, such UMNs and the MID. UMNs (Unique Machines Numbers) are included in requests whether or not a trusted section yet exists. However, the MID (the Machine Identifier) is based on a composite of all entities used to bind a trusted section. Consequently, the trusted section must exist before the MID is sent in requests. Security issues arise when, in some environments, one or more UMNs might not be available. Therefore, if a trusted section exists before an initial activation request is created, the MID is guaranteed to be included in the request. Although the MID is less stable than a UMN (it can change when binding entities change), it is useful in most cases and always serves as a backup identifier if one or more of the UMNs are unavailable.

Note • *It is not necessary to specify a prepped Trusted Configuration for FlexEnabled applications that do not perform activations.*

4

Implementing Trial Licenses and Trial Packs

You can use local activation using ASRs to implement trial licenses on a FlexEnabled client machine or on a FlexNet license server. The activation transaction obtains the license rights using an Activation Specification Record (ASR) that is either packaged with the product or delivered separately to the user. In addition to defining the license model, the ASR should also define a trial anchor. A trial anchor persists beyond the trial period and provides enhanced security by preventing the trial license from being extended when trusted storage has been deleted and recreated.

Important • *Do not configure trial license capabilities in a short-code ASR. Flexera does not support configuring these two product features together in the same ASR file. For more information about short-code ASRs, see the section titled B_ActivationSpecificationRecord.xsd in Appendix A, "XML Schemas."*

If you need to issue more than the standard allocation of 255 trials, contact your Flexera account manager to request one or more FlexNet Publisher trial packs. Each trial pack provides 256 additional trial IDs. See Using Trial Packs for more information.

Creating a Trial License

To configure trial anchors in the ASR file:

1. Optionally, change the anchoring technologies used in the Trusted Configuration file. If you don't change this, the default anchoring technologies are used. For information about creating a Trusted Configuration file, see O_ConfigPhaseOne.xsd in Appendix A, "XML Schemas."
2. Configure the activation specification record (ASR) as required to define the license model.
3. To configure trial anchors, create an `<ENTRIES>` element under `<DataDictionary>` element in the ASR.
4. Within the `<ENTRIES>` element add the following `<ENTRY>` elements:
   - Set the `PUBLISHER_DEFINED_CRITERION`, so that license rights in the ASR are activated only once.

     ```
     <Entry>
        <Key>PUBLISHER_DEFINED_CRITERION</Key>
        <Value>LOCAL_ONCE</Value>
     ```

```
</Entry>
```

If you define this element as **LOCAL_ALWAYS** and also set a PUBLISHER_TRIAL_ID, then the Trial ID is ignored and publisher anchors are created.

- Define a unique identifier for the trial anchor.

```
<Entry> <!-- Trial ID = 1 -->
   <Key>PUBLISHER_TRIAL_ID</Key>
   <Value>1</Value>
</Entry>
```

The Trial ID must be a number between 1 and 255 that uniquely identifies the trial. You can also define a Trial ID using asrgen. See asrgen in Chapter 13, Working with the Software Publisher Utilities for more information. You can create no more than 255 Trial IDs per publisher regardless of the number of trusted sections.

- Define a trial reset value. If this value is zero (**0**), you cannot delete the trial anchors later. If you intend to delete the trial anchors for this trial license, set this to a number other than zero.

```
<Entry>
   <Key>PUBLISHER_TRIAL_RESET</Key>
   <Value>0</Value>
</Entry>
```

- Define the duration of the trial license.

```
<Entry> <!-- 30 day trial -->
   <Key>PUBLISHER_DEFINED_DURATION</Key>
   <Value>2592000</Value>
</Entry>
```

This number that specifies the duration of the trial license in seconds. For example, **2592000** represents the number of seconds for a 30 day trial.

For more information about these elements, see DataDictionary Element in the Appendix A, “XML Schemas.”

See Also
Local Activation Using an ASR
asrgen
Creating an Activation Specification Record

Example Trial License Definition in an ASR

The following XML code is an example of this configuration.

```
<DataDictionary>
   <Entries>
      <Entry> <!-- Activate once only -->
         <Key>PUBLISHER_DEFINED_CRITERION</Key>
         <Value>LOCAL_ONCE</Value>
      </Entry>
      <Entry> <!-- Trial ID = 1 -->
         <Key>PUBLISHER_TRIAL_ID</Key>
         <Value>1</Value>
      </Entry>
      <Entry>
         <Key>PUBLISHER_TRIAL_RESET</Key>
         <Value>0</Value>
      </Entry>
```

```
        <Entry> <!-- 30 day trial -->
           <Key>PUBLISHER_DEFINED_DURATION</Key>
           <Value>2592000</Value>
        </Entry>
     </Entries>
  </DataDictionary>
```

Deleting Trial Anchors

There are certain situations where you may need to delete trial anchors from a system, for example if the system is a test system and you are testing the trial license capability. You can build a utility that deletes trial anchors from a system so that the trial license can be reloaded. The following functions allow you to accomplish this:

- **FlxActBool flxActCommonResetTrialASRs(FlxActLicSpc licSpc, const char * filePath);**
- **FlxActBool flxActCommonResetTrialASRFromBuffer(FlxActLicSpc licSpc, const char * pszBuffer);**

See the *C/C++ Function Reference* for more information about these functions. They remove the fulfillment record from trusted storage and reset the trial anchor that the ASR describes.

Note • *Ensure that the* `PUBLISHER_TRIAL_RESET` *property in the ASR is set to a value other than zero (0). If this value is set to zero (0) or if this property is not defined in the ASR, these functions will fail. Also, as long as the* `PUBLISHER_DEFINED_CRITERION` *is set to* `LOCAL_ALWAYS`*, no limit exists to the number of times the ASR file can be loaded.*

Important • *The trial reset functionality is not intended for use in a generally available product. You incur security risks if you make resettable ASRs (that is, set* `PUBLISHER_TRIAL_RESET` *to a value other than 0) generally available or if you leave calls to* **flxActCommonResetTrialASRs** *or* **flxActCommonResetTrialASRFromBuffer** *in production code. Best practice is to have a separate utility -- available only to QA -- that embeds the resettable ASR and calls* **flxActCommonResetTrialASRFromBuffer**.

Using Trial Packs

If you want to distribute more than the standard 255 trials that FlexNet Publisher initially provides, you can obtain one or more FlexNet Publisher trial packs. Each trial pack contains 256 additional trial IDs. These trial packs are available to publishers who use trusted storage–based licensing only.

Note • *The use of trial packs is transparent to the end user.*

The following outlines the basic process for requesting and using trial packs:

1. Contact your Flexera account manager and request a new trial pack.
2. Flexera provides you with the trial pack ID XML snippet, which contains the trial pack ID and trial pack signature.
3. Use the `asrgen` utility to add the trial pack XML snippet file as part of ASR creation. For example:

   ```
   asrgen [-t trial_id] [-x XML_Snippet_file] -c license_template -i config_template -o asr_output_file
   ```

Important • *Ensure that you use the most recent trial pack ID XML snippet received from Flexera, when creating new ASRs.*

The following shows the `<DataDictionary>` element of a sample trial ASR:

```
<DataDictionary>
<Entries>
<Entry>
<Key>PUBLISHER_DEFINED_CRITERION</Key>
<Value>LOCAL_ONCE</Value>
</Entry>
<Entry>
<Key>TRIAL_PACK_IDS</Key>
<Value>11184640</Value>
</Entry>
<Entry>
<Key>TRIAL_PACK_SIGNATURE</Key>
<Value>BF03C5B7CC529866ABA02Z5832B04544324D874DA834EFC7F77AEE51009A84414073B2B5C3AAD507A1C2C5994D60
0561D6273C3BF31AF157BABC82CE</Value>
</Entry>
<Entry>
<Key>PUBLISHER_TRIAL_ID</Key>
<Value>300</Value>
</Entry>
<Entry>
<Key>PUBLISHER_DEFINED_DURATION</Key>
<Value>2592000</Value>
</Entry>
</Entries>
</DataDictionary>
```

4. Specify a `PUBLISHER_TRIAL_ID` value that falls within the range of values made available to you by the trial pack ID XML snippet.

   Because each trial pack provides an additional 256 trials, specify a number that falls within the range of trials supported by the number of trial pack IDs that you have. For example, if you have one trial pack (indicated by the single value for `TRIAL_PACK_IDS`, you have an additional 256 trial IDs. When combined with the original 255 trial IDs that are provided in the standard allocation, this provides a range of 511 trial IDs. Each additional trial pack augments the range by 256 trial IDs. If you have two trial packs, your range is extended to 767.

5. Sign the ASR file as usual, using XMLSign.

6. Distribute the trial ASRs to your customers, along with your trial applications.

When the ASR is decoded, checks are made for the presence of a `TRIAL_PACK_IDS` entry. If the entry is not found in the ASR, the `PUBLISHER_TRIAL_ID` value cannot exceed 255. Each trial pack ID provides an additional 256 trials. If the ASR contains a `TRIAL_PACK_IDS` entry, the trial ID value is matched to its corresponding trial pack ID, to ensure that it is valid. For example, if a trial ID of 600 is specified in the ASR, the `TRIAL_PACK_IDS` entry must contain at least two values.

Trial Pack ID XML Snippet

The trial pack information required to implement additional trials is included in a text file containing XML entries—this is called a trial pack ID XML snippet. When you request a trial pack, Flexera generates a snippet containing your publisher-specific trial ID information and provides the snippet to you.

If you request an additional trial pack in the future, replace your existing XML snippet with the new snippet provided to you by Flexera.

The format of the XML snippet is similar to the following:

```
<Entry>
<Key>TRIAL_PACK_IDS</Key>
<Value>11184640</Value>
</Entry>
<Entry>
<Key>TRIAL_PACK_SIGNATURE</Key>
<Value>BF03C5B7CC529866ABA02A8532B04544324D874DA834EFC7F77AEE51009A40192563B2B5C3AAD507A1C2C5994D89
1771D6273C3BF31AF157BABC82CE</Value>
</Entry>
```

TRIAL_PACK_IDS Entry

The `TRIAL_PACK_IDS` Value element can contain one or more values, which allows for the issue of several trial packs at one time. All trial pack IDs for a publisher will be listed in the Value field, *in the order in which they are issued* by Flexera. This allocation sequence of `TRIAL_PACK_IDS` must be maintained.

TRIAL_PACK_SIGNATURE Entry

The content of the trial packs XML snippet is signed when created, which prevents other publishers from using the trial pack ID. This signature is verified upon ASR load by the client software using the public key. The ASR load will fail if an invalid signature is detected.

Support for Trials in Virtual and Non-Virtual Environments

A new data dictionary element, VIRTUALIZATION_POLICY, allows you to control whether an ASR can be loaded on physical or virtual environments:

- For trials allowed only on physical machines, set `VIRTUALIZATION_POLICY=PHYSICAL`.
- For trials allowed only on virtual machines, set `VIRTUALIZATION_POLICY=VM_ONLY`.

The following three scenarios cover the most likely trial requirements for a publisher:

- **Allow trials on both physical and virtual environments**--If you do not want to distinguish between trials on virtual and non-virtual machines, simply omit the `VIRTUALIZATION_POLICY` option.
- **Allow trials only on physical environments**--Set `VIRTUALIZATION_POLICY=PHYSICAL`.
- **Allow trials only on virtual environments**--Set `VIRTUALIZATION_POLICY=VM_ONLY`.

5

Adding Function Calls to a FlexEnabled Application

To incorporate FlexNet Publisher Licensing Toolkit into your application, you add function calls into your application code. There are two APIs available to the developer:

- **Flexible API**—provides a C interface
- **Java API**—provides a Java library.

This chapter gives an overview of the Flexible API along with a coding example.

For a Java API overview and examples, please refer to the *Licensing for Java Programming Reference Guide*. When you are ready to build and test your application, see "Building Your FlexEnabled Application" in the *Development Environment Guide*.

Using the Flexible API

The FlexEnabled application uses a set of functions that request (checkout) and release (checkin) licenses of selected features. The Flexible API is appropriate for applications that can invoke C functions. This section gives a brief overview of how the Flexible API is used in conjunction with license rights held in trusted storage on a FlexEnabled client.

When your application uses license rights held in trusted storage on the FlexEnabled client, you modify your application code to use functions provided by the Flexible API.

To provide access to the license rights held in trusted storage and use the functions in conjunction with these license rights the activation library must be loaded. See Adding Function Calls to Your Application for additional details.

The Flexible API provides two basic operations: *checkout* and *checkin*:

- **Checkout**— This enables you to link the operation of some feature of your application to the availability of a license for this feature.

  A checkout call is inserted in your code to obtain a license for a feature before the function is run. So if you wanted to license an add function, for example, you would insert a checkout call that identified the add feature before the block of code that provided the add function. FlexNet Publisher Licensing Toolkit indicates if a valid license for that feature is available. If a valid license is not available, then normally the operation of the associated product functions will be terminated.

When the FlexEnabled application uses local trusted storage, the default behavior is to use a license available in trusted storage on the FlexEnabled client. Only if the requested license is not available in local trusted storage are licenses in local license files, or from a license server, used.

- **Checkin**—When the feature is no longer in use, the license can be released using the checkin function.The checkin function may not be required because there is no sharing of licenses, but it is recommended that it is included so that the product code can be used with other functions without modification.

The Flexible API is provided in both C and Java. Some of the functionality in the Flexible API is not applicable to a license held locally (either in client-side trusted storage or in a license file on the FlexEnabled client). You can include the functionality that is not applicable in fulfillment records held in client-side trusted storage and it is ignored by the client library.

The following summarizes the functionality that is not applicable to FlexEnabled applications using a license held in client-side trusted storage:

- Server heartbeat management and communication
- Debug log file messages
- License usage information
- Vendor daemon messages and information

Adding Function Calls to Your Application

This section describes how to modify your application code to call the Flexible API functions. The `platform` directory in the toolkit contains a sample implementation using the Flexible API. This implementation serves as a basis for the example in this section and produces a minimal, well-behaved FlexEnabled application.

Figure 5-1 schematically shows a complete example using the minimum set of Flexible API functions to produce a minimal, well-behaved FlexEnabled application. This example is derived from `machind/lmflex.c` in the toolkit.

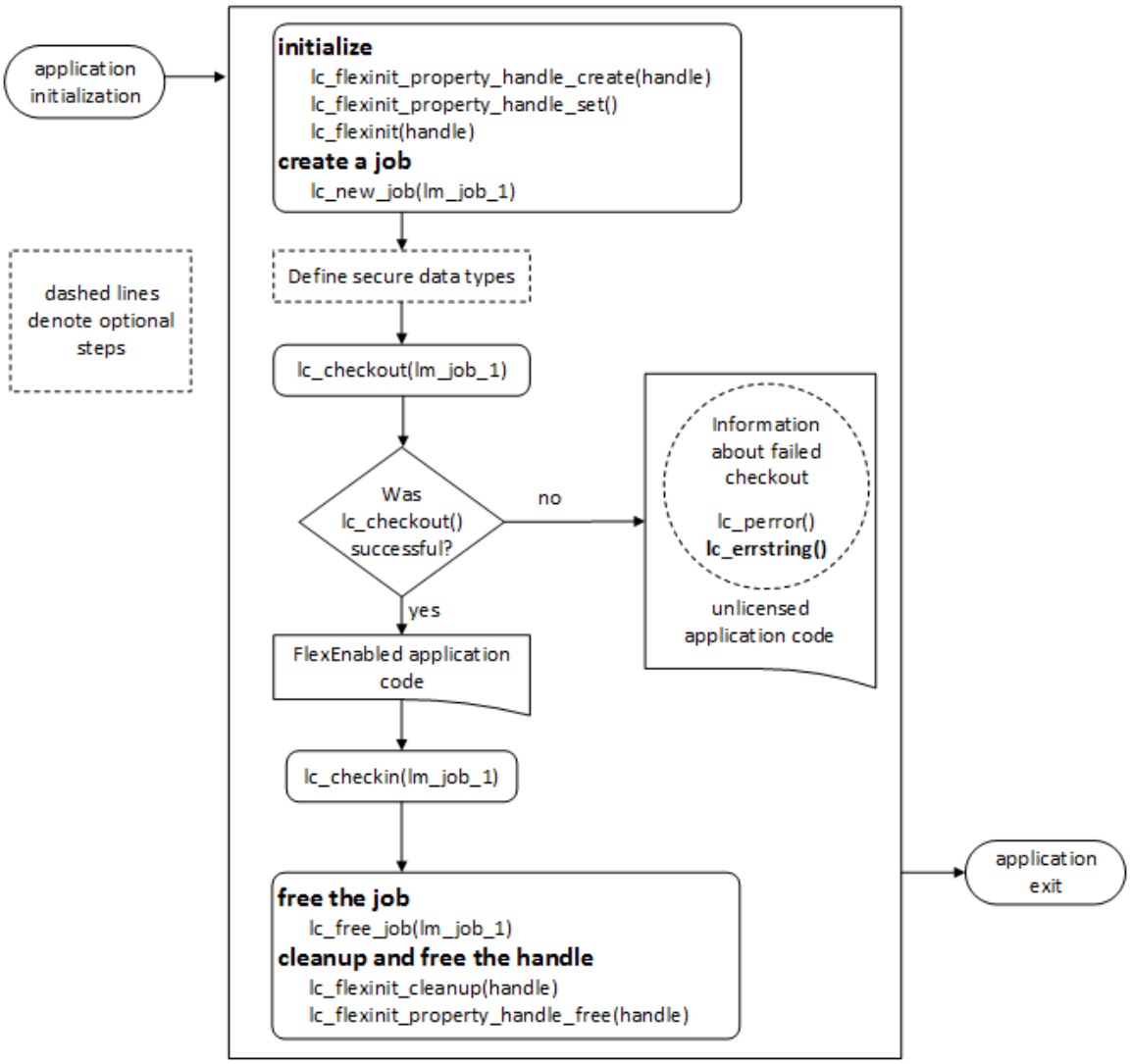


Figure 5-1: Flexible API Implementation

The functions referenced in this implementation are divided into the following categories:

- **Initialization and cleanup**— These initialization functions are required for activation support, the tamper resistant runtime, Secure Data Type (SDT) implementation, and anti-debugger technology (ADT). For an outline description of these functions see Initialization and for detailed information see the *C/C++ Function Reference*.
- **Job management and license checkin and checkout**—for details, see the descriptions of the following functions in *C/C++ Function Reference*:
  - **lc_new_job**
  - **lc_checkout**
  - **lc_checkin**
  - **lc_free_job**
  - **lc_set_attr**

- **lc_perror**
- **lc_errstring**

Add calls like the following to your application code.

```
#include "lmclient.h"
#include "lm_attr.h"
       /*...*/
       VENDORCODE code;
       LM_HANDLE *lm_job;
       struct flexinit_property_handle *initHandle;
       int stat;

/* Open and initialize the property handle. See
   Initialization for further details.*/
       if (stat = lc_flexinit_property_handle_create(&initHandle))
       {
             /* error processing */
             exit(1);
       }
       if (stat = lc_flexinit_property_handle_set(initHandle,
             (FLEXINIT_PROPERTY_TYPE)FLEXINIT_PROPERTY_USE_TRUSTED_STORAGE,
             (FLEXINIT_VALUE_TYPE)1))
       {
             /* error processing */
             exit(1);
       }
       if (stat = lc_flexinit(initHandle))
       {
             /* error processing */
             exit(1);
       }

/* Create a job */
       if (lc_new_job(0, lc_new_job_arg2, &code, &lm_job))
       {
             lc_perror(lm_job, "lc_new_job failed");
             exit(lc_get_errno(lm_job));
       }

/*
Optionally, define secure data types here.
*/

/* Check out a license. */
       if (lc_checkout(lm_job, "myfeature", "1.0", 1,
           LM_CO_NOWAIT, &code, LM_DUP_NONE))
       {
             /* Error processing. See Handling Checkout Request Denials for
                further details. */
       }

/*
```

```
Checkout succeeded. Actual application code is here.
*/

{
       /*...*/
}

/* Check in all licenses and free the job. */
       lc_checkin(lm_job, "myfeature", 0);
       lc_free_job(lm_job);

/* Clean up and release the property handle. */
       if (stat = lc_flexinit_cleanup(initHandle))
       {
             /* Error processing */
       }
       if (stat = lc_flexinit_property_handle_free(initHandle))
       {
            /* Error processing */
       }
/* Application exit code*/
       /*...*/
```

The header file `lmclient.h` contains all the symbolic definitions required for most calls to the API functions. `lm_attr.h` contains the attribute definitions used in calls to **lc_set_attr** and **lc_get_attr**. See the *C/C++ Function Reference* for a complete list of attribute definitions and for a full description of the **lc_checkout** function parameters.

lc_set_attr Attributes for Trusted Storage

The following table lists the **lc_set_attr** attributes that are directly relevant to FlexEnabled applications that use trusted storage. For full details of these attributes see the relevant sections in the *C/C++ Function Reference*.

Table 5-1 • lc_set_attr Attributes for Trusted Storage

Attribute	Description
LM_A_TS_CHECK_BADDATE	If set checks for windback detection in trusted storage.
LM_A_SORT_TS_FIRST	By default is set to use licenses from local trusted storage first. Use this attribute to turn off this default.
LM_A_TS_ONLY	To use server and client based trusted storage licenses only. ***Note*** • *To use this attribute, set the LM_A_SORT_TS_FIRST attribute.*

Initialization

Initializing the FlexNet Publisher Licensing Toolkit requires that you create and initialize a property handle and load the activation library. This is necessary in the following instances.

- Your application accesses license rights in trusted storage on the system local to the application.
- You wish to take advantage of antidebugger technology.
- The FlexEnabled application uses Secure Data Types (SDTs).

Table 5-2 lists the initialization routines.

Table 5-2 • Initialization Functions

Function Name	Description
lc_flexinit	Loads the activation library and initializes the FlexNet Publisher Licensing Toolkit.
lc_flexinit_cleanup	Unloads the activation library and performs any cleanup functions required.
lc_flexinit_property_handle_create	Creates a property handle.
lc_flexinit_property_handle_free	Frees an initialization property handle.
lc_flexinit_property_handle_set	Sets a type and value into the initialization property handle.
lc_flexinit_property_handle_get	Returns the value for a specified property.

Coordinate the use of these routines with the following:

- Secure Data Types (SDTs)—Call **lc_flexinit** before any SDTs are used in the code. See the *Implementing Secure Data Types (SDTs)* chapter in the *Development Environment Guide* for details.
- Job management—Call **lc_flexinit** *before* calling **lc_new_job**; call **lc_flexinit_cleanup** *after* calling **lc_free_job**.

In multi-threaded environments, ensure synchronization between threads. Specific care is required in the following areas:

- Executable exit logic—Your code must explicitly check in all licenses, using **lc_checkin**, and free all jobs, using **lm_free_job**, before calling **lc_flexinit_cleanup** and exiting. Do not rely on the automatic checkin mechanism to perform proper cleanup upon exit.
- Lost connection to the license server—When a concurrent license is in use and the connection to the license server breaks, the code must explicitly check in all licenses before calling **lc_flexinit_cleanup**.

Note • *For Windows platforms, initialize and uninitialize FlexNet Publisher Licensing Toolkit outside* `DllMain`.

Handling Checkout Request Denials

In order for the checkout request to succeed, the FlexEnabled application must be able to locate valid license rights. License rights can be located either in a license file or in trusted storage. The application first tries to find a license file, using the traditional searching algorithm, containing the license rights. Failing to find a license file, it accesses trusted storage on the local system in an attempt to find a valid feature line.

If the application cannot find a valid feature line, control is returned back to the application and the checkout request is denied. If the checkout request is denied, analyze the error code received from the call to **lc_checkout**, as documented in the *Programming Reference for License File–Based Licensing*. In addition, check for the error code `LM_REPAIR_NEEDED` to determine if trusted storage needs repairing.

You must include logic in your application to handle denied checkout requests, based on the design of your license lifecycle. Some approaches include, but are not limited to:

- Activate a license via an external activation utility, thereby making it available in the local system's trusted storage. See Chapter 10, Building an Application Activation Utility for Single-Action Transactions for further details.
- Repair trusted storage as indicated by the return code from **lc_checkout**. See Repair from a License Server in Chapter 10, Building an Application Activation Utility for Single-Action Transactions for further details.
- Allow partial or full access to the application.
- Deny access to the application.

6

Initializing Trusted Storage

On all platforms, before trusted storage is accessed on a machine, the trusted storage directory must be initialized. On Windows, Linux, and OS X, the FlexNet Licensing Service also must be installed. (Note that the installation of the FlexNet Licensing Service initializes the trusted storage directory; hence, there is no additional procedure required.)

Depending on your platform, go to the appropriate section:

- Windows, Linux, and OS X platforms: Install the FlexNet Licensing Service
- All other platforms: Trusted Storage Directory Initialization (UNIX Platforms)

FlexNet Licensing Service

The FlexNet Licensing Service is an administrative proxy that runs on behalf of the user to perform system tasks where elevated privilege is required.

For trusted storage–based licensing, the FlexNet Licensing Service must be installed to enable the FlexEnabled application, the activation utilities, and the vendor daemon to operate correctly.

In addition, the FlexNet Licensing Service is required to enable the FlexEnabled application and the vendor daemon to access virtualization data (on Windows and Linux platforms only). The FlexNet Licensing Service regularly polls the system to update virtualization data. Therefore, the FlexNet Licensing Service runs continuously. The virtualization poll interval is not configurable.

Producers can perform a pre-emptive check for the FlexNet Licensing Service in their own applications, via the new **lc_fnpservice_present** API, supported on Linux and Windows, which returns a positive integer if the FlexNet Licensing Service is installed. For more information on **lc_fnpservice_present**, see the *C/C++ Function Reference*, chapter *Flexible API*.

Installing the FlexNet Licensing Service on Windows

On Windows platforms, the FlexNet Licensing Service API is used to install and uninstall the service and register that an application requires the service. It is strongly recommended that the API is used within your product installer. The Windows operating system requires that a user have administrator privileges to install and uninstall services. Additionally, the installer must be run with elevated privileges.

Two types of functions are provided:

- **MSI-compatible Installer**—Use **fnpActSvcInstallForMSI** and **fnpActSvcUninstallForMSI**.
- **C/C++ Applications**—Use **fnpActSvcInstallWin** and **fnpActSvcUninstallWin** within an application.

There are example files included in the licensing toolkit that demonstrate the use of these functions.

The recommended method of installing the FlexNet Licensing Service on Windows is from the MSI-compatible installer used to install the FlexEnabled application.

Task ***To install/uninstall the FlexNet Licensing Service using an MSI-based installer***

1. Use **fnpActSvcInstallForMSI** in the MSI to install the FlexNet Licensing Service and **fnpActSvcUninstallForMSI** to uninstall the service.
2. Set the required execution level to Administrator in the MSI.
3. Package the MSI as an executable.

   This is required because Windows will not allow an MSI to call out to a DLL (in this case `FNP_Act_Installer.dll`) that executes code requiring Administrative privileges.
4. Sign the MSI executable package.

   Windows requires executables that call out to DLLs that execute administrative-privilege code to be signed. Note that InstallShield 12 or later has settings in the Release Wizard that enable you to sign the generated installer executable.

Task ***To install/uninstall the FlexNet Licensing Service using an application or C-based installer***

1. Use **fnpActSvcInstallWin**, **fnpActSvcUninstallWin**, and **fnpActSvcGetLastErrorWin** in your application.
2. Run the application/installer with elevated privileges.

   This requires the user to log on with administrator privilege and then use the ‘run-as’ option set to administrator. Alternatively, signing the executable with an Authenticode license file will ensure that elevated privileges are used.

Installing the FlexNet Licensing Service on Linux

On Linux, the FlexNet Licensing Service comprises the FlexNet Licensing Service executable and the FlexNet Licensing Service daemon. The daemon is launched by the FlexNet Licensing Service executable.

The following command-line options are available when running `install_fnp.sh`.

Table 6-1 • `install_fnp.sh` options

Option	Description
`--nolsb`	Creates a symlink to the native loader to enable the FlexNet Publisher binaries to run without full lsb support on the target platform.
`--nodaemon`	Prevents the installation script from automatically starting the FlexNet Licensing Service daemon. This can be useful when using systemd.

Task ***To install the FlexNet Licensing Service***

1. Run `install_fnp.sh`.

   The `install_fnp.sh` shell script must be run with root privilege. The script configures the FlexNet Licensing Service executable to run as a root-privilege setuid process.

2. The FlexNet Licensing Service daemon needs to run continuously. Therefore, a further installation step is required to ensure it is started a boot time. Because the FlexNet Licensing Service daemon does not need to run with root privilege, it can be started by adding the following line to a nominated user's crontab:

   ```
   @reboot /usr/local/share/FNP/service64/<version>/FNPLicensingService -r 2>&1 >/tmp/fnpd.log
   ```

Notes

- For the i86_lsb FlexNet Licensing Service daemon, the path is `/usr/local/share/FNP/service/<version>/FNPLicensingService`.
- The machine's administrator is free to redirect output as required.
- The machine's administrator may want to add an additional line to the crontab to run the above command at regular intervals to ensure that if the FlexNet Licensing Service daemon gets stopped for some unexpected reason, it will be restarted by the *crond*. The FlexNet Licensing Service daemon itself ensures there is never more than one instance running.
- Typical messages output on running the startup command are `Licensing Service daemon activated` and `Licensing Service daemon already active`.
- Rather than using cron, administrators may want to add a boot-script for the FlexNet Licensing Service daemon or optionally use systemd. The following commands are required:

  To start the FlexNet Licensing Service daemon

  ```
  /usr/local/share/FNP/service64/<version>/FNPLicensingService -r
  ```

  Note • *The process ID of the daemon will be written to `/var/run/FNPLicensingService64.pid` (`/var/run/FNPLicensingService.pid` on 32-bit).*

  To stop the FlexNet Licensing Service daemon

```
/usr/local/share/FNP/service64/<version>/FNPLicensingService -k
```

- When the Linux FlexNet Licensing Service daemon is not installed and an operation occurs that requires it, a descriptive error is displayed. For example:

  ```
  ./lmhostid -ptype VM -uuid
  ```

  results in

  ```
  lmhostid: The VM Host ID is not available. (-215,14704) - The FlexNet Licensing Service does not appear to be running.
  ```

Installing the FlexNet Licensing Service on OS X

On OS X, a shell script is used to configure the FlexNet Licensing Service executable to run as a root-privilege setuid process.

Task ***To install the FlexNet Licensing Service***

Run `install_fnp.sh`.

Sample Applications

The Windows toolkit contains two sample programs that install and uninstall the FlexNet Licensing Service using **fnpActSvcInstallWin** and **fnpActSvcUninstallWin** functions of the FlexNet Licensing Service API:

- `installanchorservice.exe`
- `uninstallanchorservice.exe`

These files are located in the `<platform_dir>` directory of the toolkit. The source code is available in `\examples\anchor_service`. These executables must be run as an Administrative user.

Note • *The* `installanchorservice` *and* `uninstallanchorservice` *executables are intended for testing only.*

The Windows toolkit also contains example InstallShield project files that you can use to create your own installer for the FlexNet Licensing Service. These files are located in the `<platform_dir>\examples\serviceinstall directory`. See the `Readme` file for information on how to use the example project files.

installanchorservice.exe

The sample executable `installanchorservice.exe` installs the FlexNet Licensing Service and registers the FlexEnabled application as requiring the service. To run this executable, ensure that the FlexNet Licensing Service installer component, `FNP_Act_Installer.dll`, and the executable are in the same folder.

Usage

`installanchorservice` *`publisher_name product_name`*

Where:

Table 6-2 • installanchorservice and uninstallanchorservice Inputs

Input	Description
`publisher_name`	The name you assign to yourself as a publisher. This string can be anything meaningful to you and does not need to match your vendor name as registered with Flexera and cannot contain the comma character. This value and the *product_name* value must uniquely identify the FlexEnabled application.
`product_name`	The name of the application. This name is used to identify that the application is using activation functionality. This string cannot contain the comma character. This value and the *publisher_name* value must uniquely identify the FlexEnabled application.

uninstallanchorservice.exe

The sample executable `uninstallanchorservice.exe` removes the specified FlexEnabled application from the registry of all applications requiring the FlexNet Licensing Service. If no other application is registered to the FlexNet Licensing Service, the executable removes the FlexNet Licensing Service from the system. To run this executable, ensure that the FlexNet Licensing Service installer component, `FNP_Act_Installer.dll`, and the executable are in the same folder.

Usage

`uninstallanchorservice` *`publisher_name product_name`*

The command line arguments are described in Table 6-2.

Trusted Storage Directory Initialization (UNIX Platforms)

If `install_fnp.sh` has not been run to install the FlexNet Licensing Service (see xref here), then it will need to be run to explicitly initialize the trusted storage directory.

For UNIX platforms on which trusted storage is accessed, you must provide a means to initialize the trusted storage directory structure. You must include logic similar to that provided in `<platform_dir>/publisher/install_fnp.sh`. Make sure your end user is aware they need to perform this initialization as `root`.

Task ***To initialize the trusted storage directory***

Run `install_fnp.sh`.

This shell script configures the FlexNet Licensing Service executable to run as a root-privilege setuid process.

At the end of the process, you should receive the following confirmation message:

```
Configuration completed successfully.
```

Note • *On most UNIX platforms, the trusted storage directory is typically* `/usr/local/share/macrovision/storage/FlexNet/`*. FlexNet Publisher does not support custom locations for the trusted storage directory.*

7

Customizing the Vendor Daemon

When you build the production toolkit, the publisher-specific vendor daemon (*vendor* or `vendor.exe`) is built also. You can customize the publisher-specific vendor daemon using vendor variables in `machind/lsvendor.c`, but changes to this file are normally neither suggested nor required.

Task ***To customize the vendor daemon using vendor variables***

1. Configure the licensing toolkit as described in the *Development Environment Guide*.
2. Edit `lsvendor.c`; ensure that you are using the latest version from the current toolkit.
3. Build the production toolkit as described in the *Development Environment Guide*.

The build process generates the intermediate file, `lsrvend.c`, from `lsvendor.c`. Any errors in `lsvendor.c` show up as compile-time errors in `lsrvend.c`.

Basic Vendor Variables

If you need to customize your vendor daemon, you can edit the vendor variables in `lsvendor.c.` Usually, this file should be left as is. Most of the variables in this file appear for historic and compatibility reasons and should not be used except where required for compatibility. This chapter provides information for the most popular vendor variables.

Table 7-1 • Basic Vendor Variables

Variable	Description
ls_allow_tz_override	Enables overriding of the license server time zone and time settings.
ls_allow_vm	Specifies whether the vendor daemon is restricted to run only in a virtual environment or is restricted to run only on a physical machine.

Table 7-1 • Basic Vendor Variables

Variable	Description
ls_compare_vendor_on_increment and ls_compare_vendor_on_upgrade	Controls whether an `INCREMENT` line will require the vendor string to match in order to pool its licenses. Controls whether an UPGRADE line will require the vendor string to match in order to upgrade another license.
ls_daemon_periodic	Ensures that a specific function will be called periodically in the vendor daemon's main processing loop.
ls_incallback	Installs a vendor-defined checkin callback routine.
ls_infilter	Installs a vendor-defined checkin filtering routine.
ls_min_lmremove	Makes the `lmremove` utility ineffective for a specified period of time.
ls_minimum_user_timeout	Specifies the minimum value allowed for a feature's TIMEOUT value.
ls_outfilter	Installs a vendor-defined checkout filtering routine.
ls_show_vendor_def	Enables display of vendor-defined checkout string using `lmstat`.
ls_support_custom_daemon_select_timeout	Controls the use of the `DAEMON_SELECT_TIMEOUT` keyword in the vendor-daemon options file.
ls_user_init1	Installs an initialization routine that runs before normal vendor daemon initialization.
ls_user_init2	Installs an initialization routine that runs after normal vendor daemon initialization.
ls_user_init3	Installs an initialization routine that runs after the license file is read and after each `lmreread`.
ls_user_init_attributes	Installs an initialization callback routine that can be used to set attributes in your vendor daemon.
ls_vd_shutdown	Enables you to define a callback routine that is invoked when the license server is shut down via the `lmdown` utility.
ls_vendor_msg	Enables you to add support for sending messages from your FlexEnabled application to the daemon.

ls_allow_tz_override

```
unsigned int ls_allow_tz_override = 0;
```

To enable override of the license server time zone and time settings, set `ls_allow_tz_override` to 1. When the variable is set to 1, the vendor daemon reads the values of the environment variables `LM_SET_SERVER_TIMEZONE` and `LM_SET_SERVER_TIME`. Use these environment variables to test scenarios that simulate vendor daemons running in different time zones.

Note • *It is recommended that you set the `ls_allow_tz_override` variable to 0 and rebuild your toolkit before distributing to your customers, if you have set the variable to 1 for testing.*

See Also
Testing Time Zone Licensing in the *Programming Reference for License File–Based Licensing*

ls_allow_vm

```
FLEX_VM_TYPE ls_allow_vm = PHYSICAL | VM_ONLY | VM_ALL
```

Specifies whether the vendor daemon is restricted to run only in a virtual environment or is restricted to run only on a physical machine. By default, the vendor daemon can operate in a virtual environment or on a physical machine.

To use the non-default values for the `ls_allow_vm` variable, you must have a set of vendor keys that provides for virtualization support.

The following values are supported. Specify one and only one value.

Table 7-2 • ls_allow_vm Values

Value	Description
PHYSICAL	Restricts the license server to run only on a physical machine; the license server cannot be run in a virtual environment.
VM_ONLY	Restricts the license server to run only in a supported virtual environment; the license server cannot be run on a physical machine. See the current FlexNet Publisher *Release Notes* for information about supported VM platforms.
VM_ALL	Allows the license server to run on both physical and virtual environments.

ls_compare_vendor_on_increment and ls_compare_vendor_on_upgrade

```
int ls_compare_vendor_on_increment = 0;
int ls_compare_vendor_on_upgrade = 0;
```

If VENDOR_STRING is used in your license files, then the value of these two variables may need to be modified. You cannot use only one variable: If you set one, you must set both.

ls_compare_vendor_on_increment gives you control over whether an INCREMENT line will require the vendor string to match in order to pool its licenses. If set to a non-zero value, then the string is included in the pooling decision; if set to 0, then the string is not involved in pooling.

ls_compare_vendor_on_upgrade gives you control over whether an UPGRADE line will require the vendor string to match in order to upgrade another license. If set to a non-zero value, then the string is included in UPGRADE line matching decision; if set to 0, then it is not included.

INCREMENT lines add licenses to a prior FEATURE or INCREMENT line and are combined into one pool if all of the following are true:

- Feature names match
- Feature versions match
- Any node-locked hostid, if present, matches
- USER_BASED and HOST_BASED fields match
- PLATFORM fields match
- DUP_GROUP fields match
- FLOAT_OK fields match
- VENDOR_STRING fields match

See Also
License Pools
VENDOR_STRING

ls_daemon_periodic

```
void (*ls_daemon_periodic)() = 0;
```

If you set the function pointer ls_daemon_periodic in lsvendor.c to one of your functions, this function will be called approximately once per minute in the vendor daemon’s main processing loop. You must ensure that the object file (.o/.obj) for this routine is linked into your vendor daemon.

ls_get_status_is_master

```
int stat = 0;
stat = ls_get_status_is_master();
```

This function, declared in ls_attr.h, can be used from the vendor daemon's ls_daemon_periodic callback (and only from that callback) within lsvendor.c to determine if that vendor daemon is master. ls_get_status_is_master returns a non-zero integer if the vendor daemon is master. This function cannot be called from ls_user_initx callbacks because this state is unknown at that point.

A limitation is that ls_get_status_is_master works only from vendor daemons started with lmgrd. All of a triad's vendor daemons started with lmadmin will report a non-zero value from ls_get_status_is_master.

ls_incallback

```
int (*ls_incallback)() = 0;
```

To install a vendor-defined checkin callback routine, initialize `ls_incallback` with a pointer to your routine. The checkin callback is called with no parameters, and the return value is unused. The checkin callback routine is called after the checkin is performed.

To obtain the parameters of the current checkin call, use the **ls_get_attr** call described in ls_outfilter.

ls_infilter

```
int (*ls_infilter)() = 0;
```

To install a vendor-defined checkin filtering routine, initialize `ls_infilter` with a pointer to your routine. The checkin filter is called with no parameters. If it returns 0, the current checkin is aborted; a return of 1 allows the current checkin to continue. If the filter aborts the operation (returns 0), then it should set the error code, using **lc_perror**, appropriately.

To obtain the parameters of the current checkin call, use the **ls_get_attr** call described in ls_outfilter.

ls_min_lmremove

```
int ls_min_lmremove = 120;
```

The `lmremove` utility could be used to bypass the license count for a feature if a license administrator were to run `lmremove` on each user as soon as they had checked out a license. `ls_min_lmremove` makes the `lmremove` utility ineffective for a specified period of time after a user connects to the daemon (120 seconds by default).

ls_minimum_user_timeout

```
int ls_minimum_user_timeout = 900;
```

This is the minimum value (in seconds) allowed for a feature's `TIMEOUT` value. An attempt to set a timeout value less than `ls_minimum_user_timeout` will result in the maximum value being set. If `ls_minimum_user_timeout` is set to <=0, then the user `TIMEOUT` option is disabled.

ls_outfilter

```
int (*ls_outfilter)() = 0;
```

Note • *Contact Flexera Technical Support for assistance before using `ls_outfilter`. Callbacks in this area are rarely needed.*

To install a vendor-defined checkout filtering routine, initialize `ls_outfilter` with a pointer to your routine. The checkout filter is called with no parameters. If it returns 0, your routine has either checked out the feature, or rejected the checkout request. If it returns 1, then the normal server checkout occurs.

If 0 is returned and the checkout fails, set the error appropriately using **lc_perror**.

ls_get_attr

To obtain the parameters of the current checkout call, use the **ls_get_attr** call. This is for use only in the `ls_outfilter` callback.

`ls_get_attr(attr, &value)`

where:

Table 7-3 • ls_get_attr Parameters

Parameter	Description
attr	An attribute specified in `ls_attr.h`. See `LS_ATTR_LAST_REPORT_LOG_ROTATION` and `LS_ATTR_REPORT_LOG_FILENAME` for a description of attributes only used with ls_outfilter.
value	Value of the attribute.

ls_get_attr operates in the same manner as **lc_get_attr**. **ls_get_attr** allows you to retrieve the values of the feature name, user, host, display, and so on for use in your filtering function.

LS_ATTR_LAST_REPORT_LOG_ROTATION

Type

`time_t`

Default

`NULL`

This attribute is used with **ls_get_attr** to get the last date and time that the report log was rotated. **ls_get_attr** is used with a server-side checkout filter to add custom business rules that determine whether a license checkout request is granted.

See Also
LS_ATTR_REPORT_LOG_FILENAME

LS_ATTR_REPORT_LOG_FILENAME

Type

`const char *`

Default

`NULL`

LS_ATTR_REPORT_LOG_FILENAME is used to get the report log file name and to checkout filter to add custom business rules that determine whether a license checkout request is granted.

ls_set_attr

To set the parameters of the current checkout call, use the `ls_set_attr` call. This is for use only in the `ls_outfilter` callback.

`ls_set_attr(attr, &value)`

where:

Table 7-4 • ls_set_attr Parameters

Parameter	Description
attr	An attribute specified in `ls_attr.h`. See LS_ATTR_CHECKOUT_DATA for more information.
value	Value of the attribute.

ls_set_attr operates in the same manner as **ls_get_attr**. **ls_get_attr** allows you to retrieve the values of the feature name, user, host, display, and so on for use in your filtering function.

LS_ATTR_CHECKOUT_DATA

Type

`char *`

Default

`NULL`

LS_ATTR_CHECKOUT_DATA is used to set and get the checkout data.

ls_checkout

The **ls_checkout** vendor daemon routine is only for use in `ls_outfilter` callbacks:

```
ls_checkout(feature, num_lic, wait, who, version, dup_sel, linger, sign, 0,1);
```

Parameters

The **ls_checkout** function uses the following parameters:

Table 7-5 • ls_checkout Parameters

Parameter	Description
feature	Feature desired.
num_lic	Number of licenses.
wait	“Wait until available” flag. If set to `1`, the request is queued if a license is not available.
who	The user.
version	Version number of feature.
dup_sel	Duplicate license selection criteria.

Table 7-5 • ls_checkout Parameters

Parameter	Description
linger	How long the license is to linger.
sign	Signature in the feature definition line.

Note • **ls_get_attr** *can be used to retrieve all the parameters that* **ls_checkout** *requires.*

Return

The **ls_checkout** function returns the following:

Table 7-6 • ls_checkout Return Values

Return Value	Description
0	Success, license checked out
<> 0	Error.

ls_show_vendor_def

```
int ls_show_vendor_def = 0;
```

Your FlexEnabled application can send a vendor-defined checkout string to the daemon on each checkout request. If `ls_show_vendor_def` is non-zero, this data will appear in **lc_userlist** calls, and hence, in `lmstat` output. If you use this vendor-defined checkout data and want your users to be able to view it with `lmstat`, then set `ls_show_vendor_def` to 1.

ls_support_custom_daemon_select_timeout

```
unsigned int ls_support_custom_daemon_select_timeout = 0;
```

This variable controls the use of the `DAEMON_SELECT_TIMEOUT` keyword in the vendor-daemon options file. `DAEMON_SELECT_TIMEOUT` allows a license administrator to define a threshold for the amount of time that can elapse before a vendor-daemon connection experiences a timeout.

- If this variable is set to **0** (the default), the `DAEMON_SELECT_TIMEOUT` keyword is disabled, prohibiting customization of the timeout threshold.
- If this variable is set to **1**, the license administrator can use `DAEMON_SELECT_TIMEOUT` to customize the timeout threshold.

When `DAEMON_SELECT_TIMEOUT` is disabled (or not defined) in the options file, a vendor-daemon connection will time out after 1 second, by default.

Important • *Best practice is to set this variable to* **1** *only for environments known to have limited bandwidth.*

ls_user_init1

```
void (*ls_user_init1)() = 0;
```

To install an initialization routine that runs before normal vendor daemon initialization, initialize `ls_user_init1` with a pointer to your routine and make sure an object file with this function is linked with your vendor daemon.

ls_user_init2

```
void (*ls_user_init2)() = 0;
```

To install an initialization routine that runs after normal vendor daemon initialization, initialize `ls_user_init2` with a pointer to your routine and make sure an object file with this function is linked with your vendor daemon.

ls_user_init3

```
void (*ls_user_init3)() = 0;
```

To install an initialization routine that runs after the license file is read and after each `lmreread`, initialize `ls_user_init3` with a pointer to your routine and make sure an object file with this function is linked with your vendor daemon.

ls_user_init_attributes

```
void (*ls_user_init_attributes)() = 0;
```

To install an initialization callback routine that can be used to set attributes in your vendor daemon, initialize `ls_user_init_attributes` with a pointer to your routine. An object file that includes this initialization function must be linked to your vendor daemon. The callback routine executes after the vendor daemon is initialized but before any license files are read.

The following functions can be used within your intialization function to customize your vendor daemon:

- **lc_set_attr**—Sets attributes in a single vendor daemon or for all vendors defined in a Common Vendor Daemon.
- **lc_set_attr_vendor**—Sets attributes for a single vendor in a Common Vendor Daemon.
- **ls_set_log_vendor_enable**—Defines which debug log messages are output by the vendor daemon.

See the *C/C++ Function Reference* for a description of **lc_set_attr** and **lc_set_attr_vendor,** and the attributes that can be set using them.

ls_set_log_vendor_enable

The **ls_set_log_vendor_enable** function is for use only in `ls_user_init_attributes` callbacks.

```
int ls_set_log_vendor_enable(
    char * string,
    int enable);
```

Parameters

The **ls_set_log_vendor_enable** function uses the following parameters.

Table 7-7 • ls_set_log_vendor_enable Parameters

Parameter	Description
string	Specifies the type of debug log message whose output is to be controlled. The following are valid values: ● "IN" = details of checked-in licenses ● "OUT" = details of checked-out licenses ● "QUEUED" = details of license requests that have been queued ● "DENIED" = details of license requests that were denied ● "UNSUPPORTED" = details of license requests that were for unsupported features
enable	Specifies whether debug log messages of the specified type are output. The following are valid values: ● 1 = enable logging (the default) ● 0 = disable logging

Return

This function returns the following values.

Table 7-8 • ls_set_log_vendor_enable Return Values

Return Value	Description
0	Success.
1	Error, an invalid parameter has been input.

Discussion

The default behavior of the vendor daemon is to output all debug log messages. This function disables the output of the specified type of debug log message.

The output of debug log messages is also controlled by the setting of the `NOLOG` option in the options file. This setting is overriden by the `NOLOG` parameter in the options file. See the *License Administration Guide* for details about the options file.

ls_vd_shutdown

```
void (*ls_vd_shutdown)() = 0;
```

You can define a callback routine that is invoked when the license server is shut down via the `lmdown` utility. (For example, you might need a routine that provides customize cleanup operations.) To implement this callback, initialize `ls_vd_shutdown` with a pointer to the routine.

Note • *The additional time needed for callback processing might cause the vendor daemon to time out and terminate before the callback can complete.*

ls_vendor_msg

```
char * (*ls_vendor_msg)() = 0;
```

To add support for sending messages from your FlexEnabled application to the daemon (with **lc_vsend**), initialize `ls_vendor_msg` with a pointer to your routine which processes the message and creates the reply for the FlexEnabled application. `ls_vendor_msg` is called with a single parameter, the character string sent by the FlexEnabled application. It should create a reply message and return a pointer to it. The message string will be unused the next time that `ls_vendor_msg` is called, so the use of a single static `char` array in `ls_vendor_msg` is appropriate. Make sure an object file with this routine is linked with your vendor daemon.

See Also
`lc_vsend` in the *C/C++ Function Reference*

Advanced Vendor Variables

The vendor variables in this section provide advanced capabilities beyond what is provided by the basic variables in Customizing the Vendor Daemon.

Table 7-9 • Advanced Vendor Variables

Variable	Description
ls_a_license_case_sensitive	Controls license file case sensitivity in the vendor daemon.
ls_do_checkroot (UNIX Only)	Controls requirement for the vendor daemon's residence on a real root file system.
ls_dump_send_data	Controls the output of transmitted daemon data.
ls_entitlement_based_pooling	Enables license pooling based on the entitlement ID.
ls_single_xver_signature	customize X-version usage for legacy clients.
ls_ts_borrow_reclaim_percentage	Controls the activation borrow reclaim operation
ls_ts_update_poll_interval	controls the poll interval to detect server trusted storage license rights update for trusted storage.
ls_use_exclusive_commRev4	Authenticates and communicates CommRev4 messages.
ls_user_lockfile	Renames the vendor daemon locking mechanism.
ls_user_lock (Windows only)	Renames the vendor daemon locking mechanism.

ls_a_license_case_sensitive

```
int ls_a_license_case_sensitive = 0;
```

This variable controls case sensitivity for feature definition lines in the license file in the context of the vendor daemon. If set to 1, licenses are case sensitive. Default is 0, not case sensitive.

If this variable is set to 1, then the corresponding case-sensitivity attribute, LM_A_LICENSE_CASE_SENSITIVE, must be set in the FlexEnabled application and license generator. ls_a_license_case_sensitive has no impact while checking out a borrowed license from the system registry as this flag is applicable only during check-out from license server. Check-out from borrow registry is always case-sensitive.

See Also
LM_A_LICENSE_CASE_SENSITIVE in the *C/C++ Function Reference*
ls_hud_hostid_case_sensitive in the *Programming Reference for License File–Based Licensing*

ls_do_checkroot (UNIX Only)

```
int ls_do_checkroot = 0;
```

To require that your vendor daemon be running on a file system that has its root directory as the "real" root directory of the disk, set this option. This prevents a user from cloning part of the UNIX file hierarchy and executing the daemon with a chroot command. If this were done, the vendor daemon locking would be bypassed and the user could run as many copies of your vendor daemon as desired.

Note • *Theft by using chroot is an obscure, difficult kind of theft. This is because the user must have root permission, and setting up a phony root directory is a non-trivial task. It requires that the necessary parts of the OS from /etc, /dev, /bin, and so on be copied into this phony root directory and is an ongoing administrative hassle.*

The check performed by **ls_do_checkroot** will fail on a diskless system. This prevents diskless systems from acting as vendor daemon. Running license daemons on diskless nodes is not recommended, but if you choose to support this, you need to set **ls_do_checkroot** to 0.

For improved security, set **ls_do_checkroot** to 1. To minimize confusion and support calls when your users are running on diskless nodes, set **ls_do_checkroot** to 0.

ls_dump_send_data

```
int ls_dump_send_data = 0;
```

This variable controls the debug output of transmitted daemon data. It should normally be left set to 0.

ls_entitlement_based_pooling

```
int ls_entitlement_based_pooling = 0;
```

This variable enables license pooling based on the entitlement ID.

To enable pooling based on the entitlement ID, set this variable to 1. If it is set to 0, the entitlement ID is not considered as a pooling component. The default setting is 1 (pooling is enabled).

This variable is useful in scenarios where a license administrator wants to differentiate between features that have identical names but belong to different products. `ls_entitlement_based_pooling` must be set to 1 when using the `INCLUDE` or `EXCLUDE` option in combination with the `ENTITLEMENT` modifier in the vendor daemon options file. For more information, see the *License Administration Guide*, chapter *Managing the Options File*.

When `ls_entitlement_based_pooling` is set to 1, ensure that the `INCREMENT` lines are unique. Otherwise, the license administration utilities might not work as expected.

ls_single_xver_signature

```
int ls_single_xver_signature = 0;
```

The X-version signature is used to customize X-version usage for legacy clients by serving all the checkout requests from legacy clients. The behavior of the vendor variable is described in the following:

- When the variable is set to 0, multiple versions of the FlexEnabled application, each based on a different version of FlexNet Publisher, take a designated version mentioned in the feature line and can be supported with one license file. The default value is 0.

  Example: To support four separate version signatures (current, v7.1, v8.1, and v11.4), the feature definition line looks something like this:

  ```
  FEATURE f1 demo 1.0 permanent 5 TZ=SERVERTZ SIGN=0 V7.1_SIGN=0 V8.1_SIGN=0 V11.4_SIGN=0.
  ```

- When the variable is set to 1, the cross-version mentioned in the `FEATURE` line serves all the checkout requests from legacy clients irrespective of the client version.

  Example: To support four separate version signatures (current, v7.1, v8.1, and v11.4), the feature definition line looks something like this:

  ```
  FEATURE f1 demo 1.0 permanent 5 TZ=SERVERTZ SIGN=0 V11.4_SIGN=0.
  ```

ls_ts_borrow_reclaim_percentage

```
unsigned int ls_ts_borrow_reclaim_percentage = 0;
```

The `lmremove` command enables license administrator to perform a borrow reclaim action by removing the deduction records from server-side trusted storage for the specified client host. This vendor variable enables publishers to enable the reclaim feature by setting an appropriate value for the borrow reclaim percentage. By default, the reclaim percentage is 0, the borrow reclaim operation by license administrator is not allowed.

The vendor variable behaves as follows:

- When this vendor variable is set to 0, activation borrow reclaim operation is not allowed. The default value is 0.
- When this vendor variable is set to X (X > 0), X% of total deduction records is allowed to be reclaimed.

This feature is intended primarily for high-license-count, high-borrow-activate enterprise scenarios. In such cases ls_ts_borrow_reclaim_percentage = 10 is recommended.

ls_ts_update_poll_interval

```
unsigned int ls_ts_update_poll_interval = 600;
```

When a server-side trusted storage license right is updated, the vendor daemon detects it in approximately 600 seconds and performs an automatic reread to update its license rights. This vendor variable enables publishers to change this poll interval, if needed.

The value range of this vendor variable is 30 <= X <= 86400, where X takes 30 seconds as minimum and 24 hours as maximum. If the value of `ls_ts_update_poll_interval` is not within the allowed range, the TS poll interval is set to the default value of 600 seconds.

The vendor variable behaves as follows:

- When this vendor variable is set to 0, poll is disabled and the server trusted storage license rights update is detected on vendor daemon by midnight reread only. The default value is 600 seconds.
- When this vendor variable is set to X, poll is done at every X seconds. If server-side trusted storage update is detected at poll, the vendor daemon reread is triggered.

Note • *lmadmin users need to set cacheTimeout to a matching value in order for the lmadmin webportal to be automatically updated.*

The options file setting AUTOMATIC_REREAD OFF disables this vendor variable.

ls_use_all_feature_lines

```
int ls_use_all_feature_lines = 0;
```

The variable causes your vendor daemon to process every `FEATURE` line in the license file as an `INCREMENT` line.

With **ls_use_all_feature_lines** set to a non-zero value, any old `FEATURE` lines which you may have shipped will now be "legal," so, for example, if you had shipped a `FEATURE` line with a count of 5, then upgraded it with a new line with a count of 7, your users would now be able to use 12 licenses.

Note • *License borrowing depends on the `INCREMENT` line, so if you use* **ls_use_all_feature_lines***, then license borrowing will not be available to you.*

See Also
Feature Definition Line Syntax

ls_use_exclusive_commRev4

```
int ls_use_exclusive_commRev4 = 0;
```

This variable can be set to 1 for vendor daemon to authenticate and communicate with commRev4 messages only.

ls_user_lockfile

```
char * ls_user_lockfile = (char *)NULL;
```

By default, only one instance of a vendor daemon with the same name can run on the same system at one time; a lock file is used to prevent multiple copies from executing at the same time.

The default lock file names are—

- **UNIX**—`/var/tmp/lock`*vendor*. On some systems, including DEC Alpha, the location is `/var/tmp/.flexlm/.lock`*vendor*.

  Note • *It is not recommended to have /var/tmp on a shared network drive for FlexNet Publisher licensing.*

- **Windows**—`C:\flexlm\`*vendor*

—where *vendor* is the vendor daemon name, as on the `VENDOR` line in the license file.

To change the location of the lock file, set **ls_user_lockfile** to the new location. If **ls_user_lockfile** is `NULL`, the default lock file will be used. Express the lock file name as the file's full path name.

Usage Guidelines for Vendor Daemon Synchronization

Windows Platforms

- If one or more vendor daemons involved in the synchronization are earlier than version 8.2, use this mechanism for all daemons in the group.
- If all vendor daemons involved in the synchronization are version 8.2 or later, see ls_user_lock (Windows only).

UNIX Platforms

- Use this mechanism regardless of vendor daemon version.

ls_user_lock (Windows only)

```
char *(*ls_user_lock)() = NULL;
```

By default, only one instance of a vendor daemon with the same name can run on the same system at one time. To prevent multiple vendor daemons that have different vendor names from running simultaneously, use this callback mechanism.

Initialize **ls_user_lock** with a pointer to your routine that returns a string containing either one lock name or a list of space delimited lock names. The lock names are used in a lock-out mechanism to prevent multiple copies of the vendor daemon from running on the same system. Lock names have the following restrictions:

- The maximum number of lock names supported per vendor daemon is 30.
- Each lock name can contain a maximum of 179 characters and any ASCII character except "`/`", "`\`", or space.
- Lock names are case sensitive.

Any other vendor daemon using one or more of these lock names is prevented from running on the same system at the same time. The default value includes one lock name: the name of the vendor daemon. If **ls_user_lock** is `NULL`, then this default lock name is used.

This mechanism is provided as a callback so that the set of lock names can be constructed at run time, rather than appearing as a literal string in the binary for a hacker to find and change.

Example

The following code is an example of a vendor-defined callback function. This example returns a string of three lock names: `TOM`, `DICK`, and `HARRY`.

```
char* MyCallBackFunction()
{
/* Return a string constructed dynamically at runtime.
   For the purposes of this example, a static string is returned.
*/
    return ("TOM DICK HARRY");
}
```

This callback is registered using the **ls_user_lock** identifier:

```
char *(*ls_user_lock) () = MyCallBackFunction;
```

The callback function can be included directly in `machind\lsvendor.c` or can be linked in from another object file during the vendor daemon build process.

Usage Guidelines for Vendor Daemon Synchronization

Windows Platforms

- If all vendor daemons involved in the synchronization are version 8.2 or later, use this mechanism for all daemons in the group.
- If one or more vendor daemons involved in the synchronization are earlier than version 8.2, see ls_user_lockfile.

UNIX Platforms

This mechanism is not applicable; see ls_user_lockfile.

8

Transactions with a FlexEnabled Client

The following chapter describes the activation transactions that create license rights in client-side trusted storage on a FlexEnabled client.

Introduction

The license rights held in trusted storage on a FlexEnabled client can be obtained from the publisher's activation server, such as FlexNet Operations or a custom activation server, or from a license server. These license rights are intended for FlexEnabled applications installed on the system and remain there unless they are returned to the originating system.

License rights held on a FlexEnabled client are returned to the system that supplied them using a *return* transaction. The FlexEnabled application can be also designed to initiate a repair transactions when trusted storage on that system must be repaired.

Components Required on a FlexEnabled Client

The following components are required on the FlexEnabled client to support trusted storage–based licensing. Make sure you also read the document, *Development Environment Guide* for information about setting up the development environment, building a FlexEnabled application, and packaging all related components.

- The publisher's application that is modified using the client libraries to create a FlexEnabled application. See Chapter 5, Adding Function Calls to a FlexEnabled Application for additional information.
- An application activation utility that initiates activation transactions. (This functionality can also embedded in the FlexEnabled application code.)

  For information about building an application activation utility, see Chapter 10, Building an Application Activation Utility for Single-Action Transactions or Chapter 12, "Building an Activation Utility for Composite Transactions." Additionally, the toolkit provides a sample of each application activation-utility type.
- For standard toolkits, the activation library component, generated when the application activation utility is processed with `preptool`. See the section Chapter 13, Working with the Software Publisher Utilities for additional information.

- For `notr` toolkits, the activation library component, `demo_client_libFNP_notr.*`, generated when the toolkit is built. For additional information, see the *Development Environment Guide*.
- Trusted storage must have been configured on the customer's machine.
  - On Windows, configuration for trusted storage involves initializing the trusted storage directory and installing the FlexNet Licensing Service using `FNP_Act_Installer.dll`.
  - On UNIX, configuration for trusted storage involves initializing the trusted storage directory using `install_fnp.sh`. Additionally, on OS X and Linux, the FlexNet Licensing Service is installed. For additional information, see the *Development Environment Guide*.
- If local activation is required, an activation specification record (ASR). See Chapter 13, Working with the Software Publisher Utilities, especially the section about `asrgen`, for more information.
- If short-code transactions are required, an activation specification record for short codes. See Appendix A, "XML Schemas," specifically the section ShortCodeData Element, for more information.
- If programmatic activation is used with FlexNet Operations server or a custom activation server, the SOAP module (`FnpCommsSoap.*`)

Summary of the Activation Process

The following basic steps are used in an activation transaction where the FlexEnabled client is the activation client. That is, the customer machine on which the FlexEnabled application is installed is the system requesting licenses from the activation server. The activation server can be either the publisher's activation server (FlexNet Operations or a custom activation server) or a FlexNet enterprise license server.

1. The application activation utility uses the activation API to generate a request (XML message) for certain license rights. As part of this process, the API does the following to handle the publisher-specific section of trusted storage affected by the activation:
   - If necessary, creates the trusted storage file on the FlexEnabled client and then either creates the initial trusted section using a prepped Trusted Configuration or specifies the need for this trusted section in the request.
   - If an additional section is required in existing trusted storage, specifies the need for this section in the request.
   - Inserts details of all the current trusted sections of *client-side* trusted storage for this software publisher in the activation request.

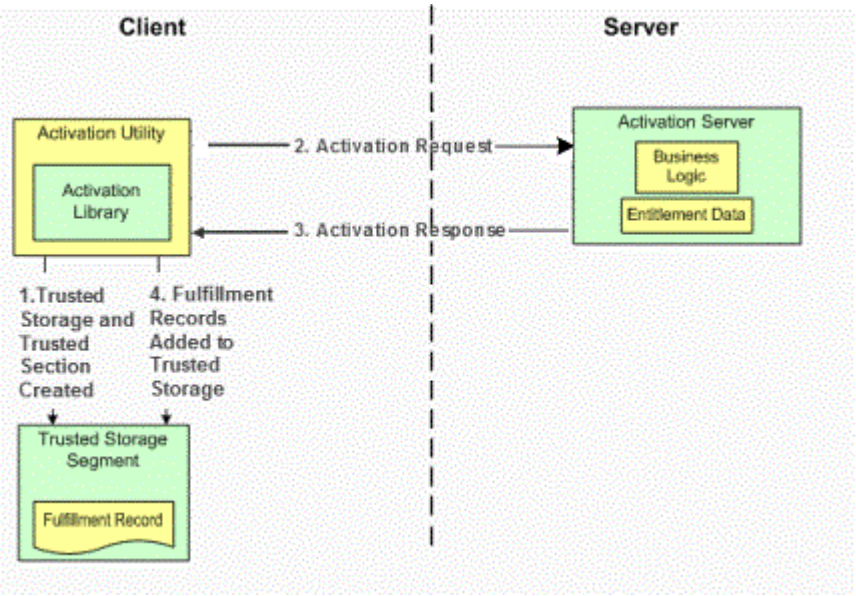
   Note • *The prepped Trusted Configuration feature was introduced in FlexNet Publisher 11.10.1 for certain platforms. With this feature enabled, the application activation utility automatically creates the initial trusted section when it generates the first activation request. In this way, the trusted section is already in place when the transaction is sent. For versions prior to 11.10.1, the initial trusted section configuration is a separate transaction step.*

2. The application activation utility sends the request to the activation server, using one of these methods:
   - Sends the activation request as a data stream across the network connection to the activation server. (This method requires a SOAP connection.)
   - Writes the activation request to an XML file, which is then manually transported to the activation server (for example, by email).

3. The activation server validates the activation request against certain business rules and determines whether the activation request can be granted.
4. If the activation request complies with the activation server's business rules, the activation server generates an activation response in the form of an XML message. This message is either transmitted as a data stream across a network connection or written to a file and transported to the license server.
5. The application activation utility loads the license rights into trusted storage on the FlexEnabled client.

The following diagram depicts this process:

Figure 8-1: Activation Transaction using XML Messages



When Using Short-Code Activation

With short-code transactions, messages are formatted as encoded characters rather than XML. This activation mechanism is intended to be used in very specific business scenarios.

1. The activation utility on the client reads an ASR file, and then generates a code containing the encoded activation request message.
2. This request is transferred to the activation server.

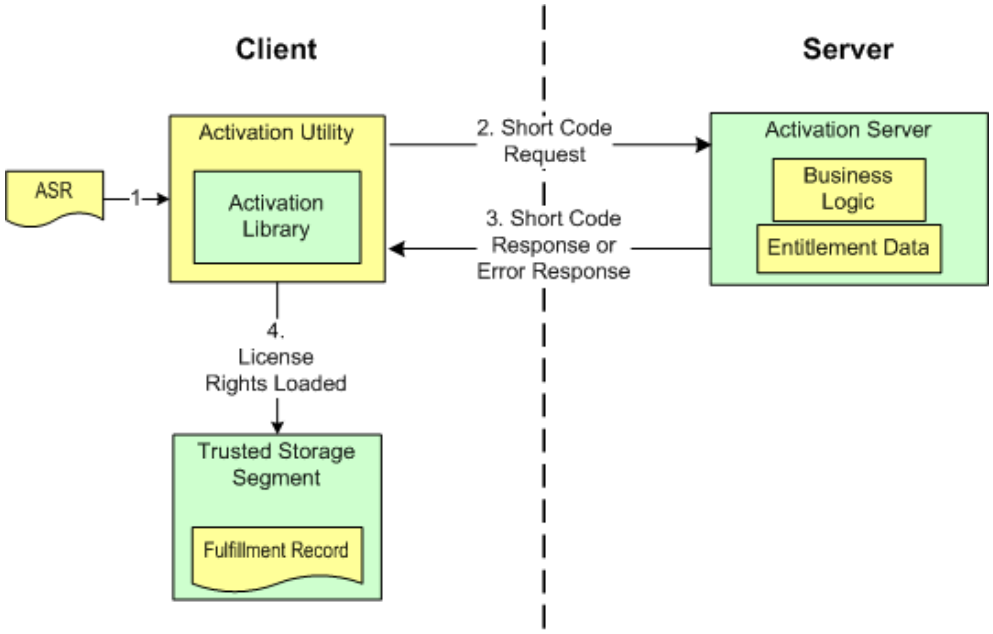
For example:

- The user reads the code to a customer support agent, who then enters the code for the activation server to process.
- The client connects to the activation server using an automated telephony system.

3. The activation server processes the short-code request and then generates either a short-code response indicating that the request was valid or an error response indicating that the request was rejected.
4. If a short-code response is returned, the activation utility uses the contents of both the short-code response and the ASR to load license rights into trusted storage.

The following diagram depicts this process:

Figure 8-2: Activation Transaction using Short Codes



For more information about using this type of activation, see the following sections in this guide:

- The section Transactions Using Request and Response Codes in Chapter 12, Building an Activation Utility for Composite Transactions for information about building this functionality into an application activation utility.
- The section B_ActivationSpecificationRecord.xsd in Appendix A, B_ActivationSpecificationRecord.xsd for information about creating an ASR file for short-code activation.

Activation Using the Publisher's Activation Server

The application activation utility initiates a composite activation transaction using the functions within the activation API to transfer the transaction data between the FlexEnabled client and the publisher's activation server (such as a FlexNet Operations server or a custom activation server):

- XML messages are sent as a data stream across an internet connection using the SOAP protocol (for FlexNet Operations).
- XML messages are sent manually as files between the activation server and the activation client (for example, by using email).

Another type of transfer mechanism is the short-code method. For examples of the API required, see Chapter 10, "Building an Application Activation Utility for Single-Action Transactions."

Best Practice

Basically, for licensing life-cycle operations (for example, activations, returns, and repairs) that involve the publisher's activation server (but do not use short codes), best practice is to use composite transactions, not single-action transactions. Composite transactions can perform multiple actions in a single transaction and provide a higher level of security than single-action transactions do. Additionally, unlike single-action transactions, composite transactions

support UMN3, the preferred machine identifier in virtual environments. For descriptions of API functions you can use for composite transactions and for more information about UMN3, see Chapter 12, "Building an Activation Utility for Composite Transactions."

Activation Using a License Server

The application activation utility initiates the transaction using the functions within the activation API. These transactions use FlexNet Publisher Licensing Toolkit built-in connections. The location of the license server that the request is sent to is obtained by the application activation utility and set using the activation API. When this location is not specifically set, the default is `@localhost`.

The API functions include the processing of the trusted storage configuration, when required, and the processing of the activation response. The application activation utility does not have to specifically handle these processes.

Local Activation Using an ASR

This type of activation does not use an activation server. All the information required for the activation is held in a signed file, called an *activation specification record* (ASR), on the FlexEnabled client machine.

Local activation can be used to implement license models for *trial* or *emergency* purposes. Trial license rights have limited duration and allow an end-user to test a product for a short period of time prior to purchasing it. Emergency license rights are provided via an ASR when the normal method of supplying license rights is not available (for example, when a network connection is unavailable or an activation server is temporarily out of service). Local activation also provides a solution for activating license rights in high-security environments.

For details of the API functions required for local activation, see the section Local Activation in Chapter 10, "Local Activation."

Repair Transactions

A repair is required when changes in trusted storage are detected that indicate trusted storage might no longer be intact. Any fulfillment records that are affected, and are *untrusted*, cannot be used until they are repaired. Thus FlexEnabled applications requiring a license contained within an untrusted fulfillment record cannot run until a repair transaction is completed. The activation API provides for the repair of fulfillment records.

Trusted-Storage Changes Due to Illegal Tampering

A trusted-storage change might be the result of an attempt to obtain license rights illegally. These attempts might include the following:

- Copying files from one system to another
- Restoring a backup that was made when trusted storage contained more licenses
- Setting the system clock to an earlier time

Trusted-Storage Changes Due to Valid Actions

A trusted-storage change might also be the result of a valid user action, such as one of the following:

- Changing a system hardware component that is used to bind trusted storage to the system
- Performing a system restore to correct system problems or roll-back changes.
- Correcting the system clock after it was set to the future, either inadvertently or while testing.

Return Transaction

A return transaction returns a fulfillment record to the source from which it was activated:

- The license server
- The FlexNet Operations server that supplied the original fulfillment record
- The publisher's custom activation server

The application activation utility initiates the return transaction. The activation server that receives the return transaction applies business rules to its processing of the return request:

- **A return to a license server**—The return is accepted if the fulfillment record was originally supplied by this license server and if incrementing the existing counts does not violate the maximum count values. The number of hops in the fulfillment record is updated to what it was originally when the fulfillment record was originally written to trusted storage on the license server. Then the fulfillment record is written to trusted storage. For general information on hops, see Limiting the Number of Transfers.
- **A return to a FlexNet Operations server**—The business rules for accepting or rejecting a return request are configurable. See the FlexNet Operations server online help for details.
- **A return to a custom activation server**—The business rules for accepting or rejecting a return request are defined completely by the software publisher when building this application.

Note • *Composite transactions allow partial returns, but the return must be made to a publisher's activation server, not an enterprise license server.*

Short-Code Transactions

Normal transactions use XML formatted messages. These are transmitted either as a data stream (without end-user intervention) or as a files (with end-user intervention). When a network connection is unavailable or the XML documents are too large, you can transmit the messages using request and response codes.

These transactions are referred to as *short-code transactions* because the request and response message is encoded as a numeric value. They are available for transactions only between a FlexEnabled client and an activation server. Typically, this type of transaction is used for telephone transactions either with a customer service representative or using automated touch tone telephony.

The limitations imposed on the length of the request and response codes require that detailed information, normally included in the activation transactions, is already present on the FlexEnabled client. The following information is provided for short-code transactions in an ASR:

- **Short-code alias**—Identifies transactions using this ASR
- **Short-code scheme ID**—Identifies the default format of the request and response codes
- **Trusted Configuration file**—Contains the data required to create or identify the trusted section in trusted storage in which the fulfillment record is to be stored
- **Fulfillment record**—Contains a set of license rights for the FlexEnabled application
- **Overrides**—Modify the default format of the request and response codes

For information on ASRs for short-code transactions, see the section ShortCodeData Element in Appendix A, "ShortCodeData Element." For information about how to implement short-code transactions, see Appendix 10, "Building an Application Activation Utility for Single-Action Transactions," specifically the section entitled Transactions Using Request and Response Codes.

Fulfillment Record on a FlexEnabled Client

Fulfillment records in trusted storage on a FlexEnabled client contain licenses for products that are installed and run on that machine only. The basic license information is held in both feature definition lines and fulfillment record properties. Thus certain attributes defined at the fulfillment-record level can override these same attributes in the feature definition lines. When a check-out function within the FlexEnabled application is run, the FlexNet code built into the application combines the two sources of information to produce a complete license line. If the license is valid for the feature, the application code continues to run.

The following is an example feature definition line from a fulfillment record for a feature called *add*:

```
FEATURE add vendor 2.0 21-dec-2020 5 SIGN="<...>"
```

This line provides the following license:

- For the *add* feature
- For a FlexEnabled application using *vendor* as the vendor daemon name
- For version 2.0 or earlier of the application
- For rights that expire on 21-dec-2020

  Note • *The expiration date defined at the fulfillment-record level might override this date.*

- For uncounted usage, even though, as in this example, a number suggests that the license is counted

9

Using the Activation Borrow API

The Activation Borrow API uses trusted storage to provide a more secure way to use licenses off-line, for example on laptops. The Activation Borrow API is intended to replace the BORROW keyword capability. For an overview of other mobile licensing capabilities, see the chapter on Mobile Licensing in *Programming Reference for License File–Based Licensing* document.

Note • *Activation Borrow is implemented with single-action transactions only, not composite transactions.*

The Activation Borrow API can be used either from a FlexEnabled application or from a stand-alone utility. The expectation is that borrowing and returning a license will generally be based in the FlexEnabled application. It is expected that the functions that display details of licenses held in trusted storage will generally be based in a separate utility.

To borrow a license, the client system must be connected to the network and be able to communicate with the license server. For example, a FlexEnabled application can be run once so that the activation process writes the license right locally into client-side trusted storage. Once this occurs, the client can disconnect from the network and continue to use the application until the expiration date. When the application is run on a client that is disconnected from the network, the license checkout process will look to the client-side trusted storage for validation.

There are also functions available that allow the application to return the borrowed license back to the license server. If the license is not returned by the expiration date, it is automatically returned to the license pool. After this date, the FlexEnabled application will no longer run. The publisher should include functionality to notify the user that their license has expired.

If the borrowed license was activated from the overdraft license pool, it will not be returned to the pool when it expires and will no longer be available as an overdraft license. See Summary of the Activation Process in Chapter 2, “Transactions to a License Server,” for more information about Activatable Overdraft licenses.

To display borrowed licenses that are stored in client-side trusted storage, you must create an activation context and then get this information using the **flxActBorrowTSViewFRAttributeGet** function. This capability can be developed in an activation utility similar to `appactutil`. The ‘lmutil’ utility cannot display borrowed licenses available in client-side trusted storage. Before calling this function call flxActBorrowTSViewCreate to create the required context object and determine the number of fulfillment records in client-side trusted storage.

When the **lc_checkout** function is called, the default order in which licenses are used is the following: client-side trusted storage, then license files, and then license servers. When you activate a license using the Activation Borrow APIs, you will want to make sure that this default behavior is in operation.

The default behavior can be overridden by:

- Using **lc_set_attr** with LM_A_SORT_TS_FIRST set to zero.
- Setting an optional *sort* field in a feature definition line.

For details of these settings, see the *Programming Reference for License File–Based Licensing document.*

License Borrowing with Activation Borrow API

When users normally obtain licenses for a product from a license server but occasionally require a mobile license, one solution is to use *activation* to provide a temporary license. Typically this functionality is required so that licensed software on a lap-top can be used when the lap-top is disconnected from the enterprise system.

The activation borrow API provides a simple activate function call that requests a temporary *product license* from the license server. The license server takes a concurrent product license from *trusted storage* on the license server and supplies it to the requestor. The borrowed product license is stored in trusted storage on the requesting system. Then it is available when the lap-top is disconnected from the enterprise system and the software is run.

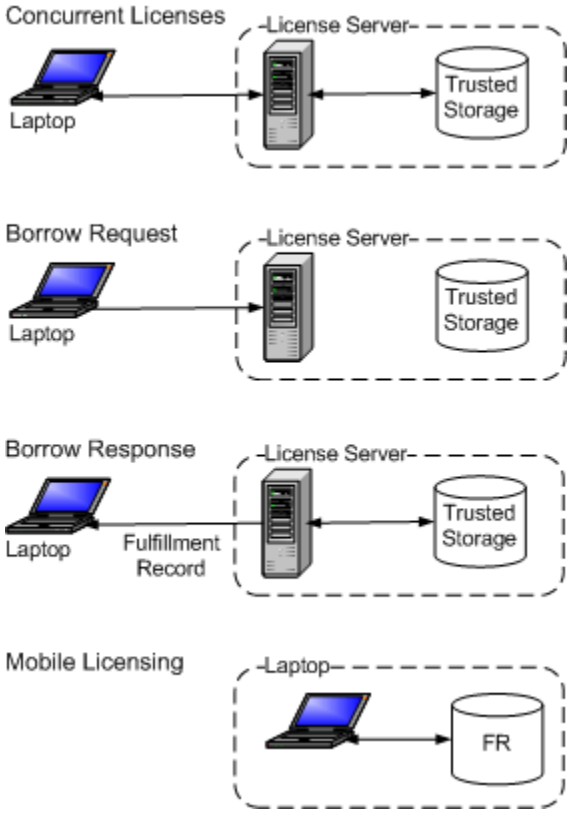
The product license is represented in trusted storage as a *fulfillment record*. A fulfillment record may contain several FEATURE and INCREMENT lines and contains a Product ID. Thus it can be used instead of a PACKAGE line to define a set of licensed features for a product.

License borrowing using the activation borrow API provides:

- Secure storage of the borrowed license—as a fulfillment record in trusted storage on the local system.
- One request can provide all the feature licenses required for a product—a fulfillment record can contain multiple FEATURE lines and a Product Id.
- Early return of a borrowed license—use the return function to return the product license to the license server that supplied it.
- If the borrowed license is an activatable overdraft type and the user does not return the license before the expiration date, it will not be added back to the available license pool. It will always count against the activatable overdraft pool for that fulfillment ID. This behavior is not the same for activatable, hybrid, and hybrid overdraft licenses. For these license types, at the end of the license period the fulfillment record on the lap-top expires and the license server adds the license back into the concurrent license pool.

The following diagram illustrates the use of activation to borrow a license:

Figure 9-1: Mobile Licensing Using Activation Borrow API



When a user wants to return a borrowed license early, before the license has expired, the activation borrow API provides a single function call. This call uses the fulfillment ID returned in the activation borrow call to specify the fulfillment record that is to be returned. Alternatively, the activation borrow API includes simple functions to examine the contents of the local trusted storage to enable the user to choose which fulfillment record to return.

Using the Activation Borrow API

To use the activation borrow API both the license server and the FlexEnabled application must support activation. The following sections describe in outline what is required.

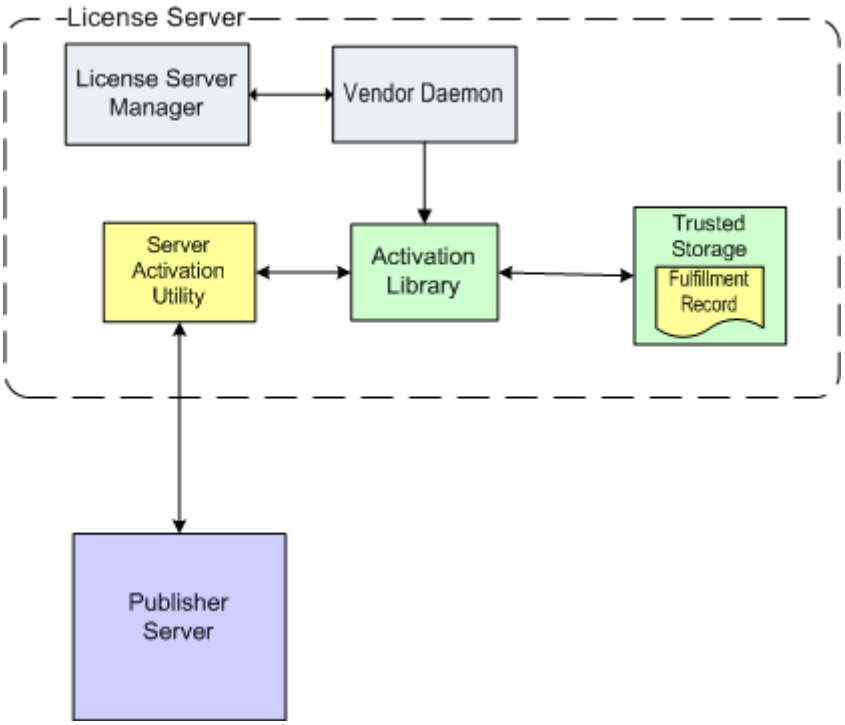
Adding Activation Support to the License Server

To support activation, the license server requires an additional module: a *server activation utility* that controls transactions that acquire license rights, fulfillment records, from a publisher server. The server activation utility is a publisher-specific utility that you write to match your product requirements.

The server activation utility and the license server access trusted storage. Access to trusted storage is provided by the *activation library*. This provides publisher-specific security and it is linked to the license server and server activation utility.

The following diagram illustrates the modules required on the license server to support activation borrow:

Figure 9-2: License Server Activation Modules



The activation library and server activation utility are installed in the same directory as the existing license server components, license server manager (`lmadmin` or `lmgrd`), and the vendor daemon. Trusted storage is created and managed by the activation library.

For detailed information on activating licenses in trusted storage on the license server, see Chapter 2, "Transactions to a License Server."

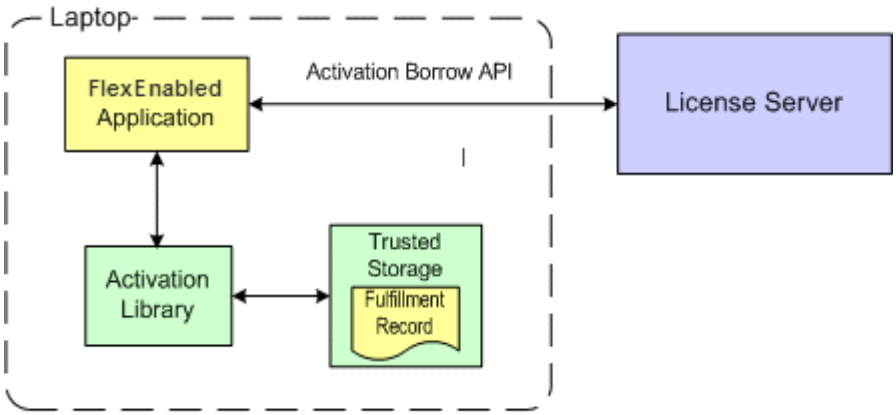
Using the Activation Borrow API in a FlexEnabled Application

To use the activation borrow API in a FlexEnabled application:

1. Add activation borrow API function calls to the FlexEnabled application and link it to the flxActBorrow header file.
2. Add activation initialization functions to the FlexEnabled application. These functions are in the Flexible API and load the activation library.
3. Create a publisher and product-specific activation library and add it to the product package. On some platforms additional components are required to install the FlexNet Licensing Service.

The following diagram illustrates the modules required locally to the FlexEnabled application to support activation borrow:

Figure 9-3: Modules Required Locally to the FlexEnabled Application for Activation Borrow



The activation library is installed in the same directory as the FlexEnabled application. Trusted storage is created and managed by the activation library.

In addition to borrowing a license, the activation borrow API can be used to view fulfillment records held in trusted storage local to the FlexEnabled application.

Using Fulfillment Records for Borrow

To use fulfillment records on a license server for borrowing licenses:

1. Set a Product ID in the fulfillment record and ensure that all the licensed features required for the product are included in *feature definition lines* within the fulfillment record. Feature definition lines that begin with the FEATURE or INCREMENT within a fulfillment record.
2. Set the count within each feature definition line within the fulfillment record to one. This ensures that there is a one to one match when using these licenses as concurrent or borrowed licenses.
3. Set the *hybrid* count—this sets the number of borrowable product licenses.
4. Set the *activatable* count to zero—to ensure that licenses are borrowed from the pool of concurrent licenses.

Hybrid licenses are included in the concurrent license pool until they are borrowed and removed from the concurrent license pool. If all the concurrent licenses are in use, then an activation request for a license will be refused. Note that fulfillment records on a license server can also contain *concurrent* and activatable counts. Concurrent licenses can only be supplied concurrently, that is they are not available for borrowing. Activatable licenses are used in preference to hybrid licenses, so set this count to zero when you want the activation borrow API to pull licenses from the concurrent license pool.

Feature definition lines do not have to contain the BORROW keyword for these feature licenses to be available for borrowing using the activation borrow API.

Comparison with BORROW Keyword

The activation borrow API can be used to provide license borrowing functionality that is equivalent to BORROW. The previous sections describe how to use it, so that it matches as closely as possible the functionality provided by BORROW. However, it does not operate in the same way, so these two methods *cannot be combined directly.*

For example, details of the licenses issued using the BORROW keyword can be displayed using lmborrow. When using the activation borrow API, the publisher must write their own utility to output details of licenses held in trusted storage. Details of all borrowed licenses are not available because trusted storage is publisher-specific and its contents are only available to publisher-supplied software. Thus, lmborrow cannot be used to display licenses borrowed using the activation borrow API.

Similarly, the Options file keywords that can be used with BORROW are not available for use with the activation borrow capability. The only options file keywords that limit activation borrow are those that apply specifically to activation, that is EXCLUDE_ENTITLEMENT, INCLUDE_ENTITLEMENT, ACTIVATION_LOWWATER and ACTIVATION_EXPIRY_DAYS.

The number of licenses available for borrowing is defined when using BORROW by the count in each FEATURE or INCREMENT line that includes the BORROW keyword. The equivalent for activation borrow is the hybrid count in a fulfillment record.

Using the BORROW keyword, a maximum borrow period is defined. To limit the possible misuse of borrowed licenses, the default borrow period is set to one week. Licenses obtained using the activation borrow capability are secure: starting and stopping the license server does not allow for extra uses. There is no provision for defining a maximum borrow period with the activation borrow capability. Licenses borrowed using the activation borrow capability cannot have an expiration date later than the expiration date of the source fulfillment record on the license server.

Using the BORROW keyword, it may be difficult to ensure that all the feature licenses for a product are borrowed. Consider the case where the user uses the lmborrow utility to initiate borrow. The sequence of steps performed by the user is as follows:

1. Run lmborrow to initiate borrow.
2. On the same day and on the same system, run the FlexEnabled application that borrow licenses are required for.
3. Use each licensed feature within the application. When each feature is checked out, a borrowed license is installed.
4. Run lmborrow to clear the borrow setting. No more licenses are borrowed.

Using the activation borrow capability, the publisher can ensure that all license features are included in the borrowed license by including them all in a single fulfillment record.

The BORROW and activation borrow capabilities are not compatible with use of three server redundant configuration. This is because persistent data is held on a single license server: for the BORROW capability, this is the borrow data-cache and for the activation borrow capability this is the data required to add the license back in to the concurrent license pool when the borrowed license expires. Note that it is a general limitation of activation that it is not supported on a three redundant server configuration.

10

Building an Application Activation Utility for Single-Action Transactions

An application activation utility manages license rights in the FlexEnabled client's trusted storage. The utility uses the Activation API to initiate and manage activation transactions between the FlexEnabled client (local to the FlexEnabled application) and either the publisher's activation server or an enterprise license server.

About Single-Action Transactions

This chapter describes the steps for creating and deploying an application activation utility that uses *single-action transactions*. This type of transaction performs one activation action (for example, one activation, repair, or return for one fulfillment record) per transaction. Additionally, single-action transactions support activations between the publisher's activation server and the FlexEnabled client. However, best practice is to use composite transactions for activations with the activation server.

If your license model requires a application activation utility that supports single-action transactions, you should use the information in this chapter to create and provide a publisher-specific version of this utility. An example application activation utility is provided with the FlexNet Publisher Licensing Toolkit at the following location:

```
<platform_dir>/appactutil.exe
```

The corresponding source code can be found at:

```
<install_dir>/examples/activation/appActUtil.c
```

Activation Utilities for Single-Action Transactions

The appactutil utility retains short code transaction support using the legacy short code API; see the Composite Transaction section for short code transactions in this release, see Chapter 12, "Building an Activation Utility for Composite Transactions."

Path and Directory Conventions Used in This Chapter

References to directories and paths in this chapter use the following conventions:

- The term `<install_dir>` is used to represent the path and directory in which your FlexNet Publisher toolkit is installed.
- The term `<platform_dir>` is used to refer to the path and directory in which the platform-specific components of the FlexNet Publisher Licensing Toolkit are located. For example, for a Windows 32-bit system, `<platform_dir>` represents `<install_dir>\i86_n3`. For a Sun Microsystems 32-bit Sparc running Solaris 10.*x*, `<platform_dir>` represents `<install_dir>/sun4_u10`.
- File paths for locations common to both UNIX and Windows platforms are constructed using forward slashes.

Creating an Activation Utility

Use the application-specific and common API functions in the activation API when creating an application activation utility. These are the functions that can be used to create and access the application's trusted storage. The basic structure of the utility is described below.

Task ***To create the sequence of Activation API calls in an activation utility***

1. Install the FlexNet Licensing Service (if it is required by the platform and has not been previously installed on the machine).

   Note • *The FlexNet Licensing Service can be installed by the FlexEnabled application installer or by a separate utility that runs during the application initialization. Publishers must develop this capability within their application. See Chapter 6, "Initializing Trusted Storage," for more information.*

2. Call **flxActCommonLibraryInit** to initialize the library. This function loads the activation library and must be called before any other Activation API.

   Note • *In trusted storage–based applications,* **flxActCommonLibraryInit** *is the same as the Flexible API function* **lc_flexinit** *(described in Chapter 5, Adding Function Calls to a FlexEnabled Application). If you are using both Activation API and the Flexible API in this utility, ensure that you call either* **flxActCommonLibraryInit** *or* **lc_flexinit,** *but not both.*

3. Open an application activation handle by calling **flxActCommonHandleOpen** with the mode parameter set to `FLX_ACT_APP`. This sets up access to client-side trusted storage. Client-side trusted storage is used to store license rights used directly by a FlexEnabled application.
4. Design logic to initiate and manage one or more activation transactions and to access fulfillment records in trusted storage. See Typical Transactions.
5. Close the activation handle by calling **flxActCommonHandleClose**, which releases all resources associated with it.

6. Release the Activation API by calling **flxActCommonLibraryCleanup** to shutdown the activation library. No other Activation API routines can be called after this point.

Important • *Note the following about* **flxActCommonLibraryCleanup***:*

- *This function is also performed by the Flexible API function* **lc_flexinit_cleanup** *(described in Chapter 5, Adding Function Calls to a FlexEnabled Application). If your utility previously called* **lc_flex_init** *(see Step 2), use* **lc_flexnit_cleanup** *here, not* **flxActCommonLibraryCleanup***.*
- *On Windows platforms, do not call* **flxActCommonLibraryCleanup** *or* **flxActCommonLibraryInit** *in* `DllMain`*. These APIs result in calls to the Microsoft Win32 FreeLibrary and LoadLibrary APIs. Calls to these Win32 APIs in* `DllMain` *are widely known to be associated with issues.*

Typical Transactions

The following is a list of typical transactions, each of which is described in subsequent sections:

- Using the Vendor Dictionary to Send Custom Data
- Local Activation
- Activation from a License Server
- Return Fulfillment Record to a License Server
- Repair from a License Server
- Activation from an Operations Server
- Return to an Operations Server
- Repair from an Operations Server
- Transactions Using Request and Response Codes
- Repair Using Request and Response Codes
- Return Using Request and Response Codes
- Activation from a Manual Response Generator
- Return Fulfillment Record to Manual Response Generator
- Repair Using a Manual Response Generator

Fulfillment Record Manipulation Functions

The following sections describe functions for manipulating records in trusted storage:

- View the Fulfillment Record from an Activation Response
- Display License Rights in Client-Side Trusted Storage
- Display Server-Side License Rights Remotely On the Client
- Delete Multiple Fulfillment Records from Trusted Storage
- Delete Multiple Fulfillment Records from Trusted Storage
- Enable Fulfillment Record when No Response to Return Request

Building and Deploying Your Activation Utility

This section provides general guidance for building and deploying an activation utility.

For detailed information about building, packaging, and releasing all products in your licensing solution, see the *Development Environment Guide*.

Task ***To build and deploy an application activation utility***

1. Compile and link the application activation utility.

   A sample `makefile` is included in the toolkit (`.../<platform_dir>/makefile.act`). Use the build steps for `appactutil.exe` as an example. The sample build steps include compilation, linking, and the use of `preptool` (see step 2).

2. For standard toolkits, use `preptool` to prepare the application activation utility and to generate a publisher-activation library component for use by the application activation utility. See Chapter 13, "Working with the Software Publisher Utilities," especially the preptool section.

   Note • *If you have created your own publisher-specific version of* `preptool`*, use that version (which might have a different name).*

   For `notr` toolkits, use the activation library component, `demo_server_libFNP_notr.*`, which was created when you built the toolkit.

3. Set up your product installer to install the application activation utility and associated components in the same directory. The following components are required:

   - The prepped version of the application activation utility
   - The activation library created in step 2
   - For Windows, the FlexNet Licensing Service installer component (`FNP_Act_Installer.dll`) needed for installation of the FlexNet Licensing Service service, whether this is performed as part of the product installation or separately.

     Note • *Ensure that the correct version of the FlexNet Licensing Service installer component is added to the FlexEnabled application. For more details, see the section "The FlexNet Licensing Service" in the Development Environment Guide.*

   - To enable transactions with an FlexNet Operations server, the `FnpCommsSoap` module.

   Optionally, the product installer can install the FlexNet Licensing Service. (This is the recommended method for installing the FlexNet Licensing Service.) To install this service, the product installer must do the following:

   - For Windows platforms, use the API provided by the FlexNet Licensing Service installer component (`FNP_Act_Installer.dll`). For more instructions, see Chapter 6, "Initializing Trusted Storage."

Note • *If your product installer is InstallShield-based, you can find examples of installing the Licensing Service in* `<platform_dir>\examples\serviceinstall`.

- For OS X and UNIX platforms, run the `install_fnp.sh` script.

Sending a Reason with a Request Message

With activation requests and return requests sent as XML-formatted messages, the activation utility can send a numeric value indicating the reason for the transaction. Software publishers must define their own values and reasons, and then define business rules that map these items. By default, this data is not captured in a request, but must be programmed into the activation utility.

Use the following functions in the activation API allow you to manipulate this information in an activation object:

- **flxActAppActivationReasonGet**
- **flxActAppActivationReasonSet**

Use the following functions in the activation API allow you to manipulate this information in a return object:

- **flxActAppReturnReasonGet**
- **flxActAppReturnReasonSet**

See the FlexNet Publisher document titled *C/C++ Function Reference* for information about using these functions.

Using the Vendor Dictionary to Send Custom Data

This is an optional element in transaction requests and responses that contains custom data to be sent with the transaction. For example, it could be used to send registration information supplied by the user when he activates a license. During a repair transaction it can be used to send additional information about the user or the system.

Custom data can be added to both request and response messages. When vendor data is available in a response, it is stored in trusted storage along with any associated fulfillment record. Note that if vendor data is included in a return response, it will not be stored if the associated fulfillment record has been removed from trusted storage.

The vendor dictionary has a standard data dictionary format and contains a set of entries. The first element in the entry is the **Key** element, which identifies the type of data in this entry. The second element of the entry, the **Value** element, contains the data for the entry. This type of entry is referred to here and elsewhere as a *key-value pair*.

It is visible in XML requests as the VendorDictionary element. The vendor data contained within an request can be viewed using the GUI version of the manual response generator, `responsegenUI.exe`. If you are using the command-line version, `responsegen.*`, you will have to examine the request before using the manual response generator to generate the appropriate response. For details, see Chapter 14, “The Manual Response Generator.” See the FlexNet Operations documentation for information about how vendor data is used.

The following functions allow you to manipulate custom data in an activation object:

- **flxActAppActivationVendorDataGet**
- **flxActAppActivationVendorDataSet**

The following functions allow you to manipulate custom data in a return object:

- **flxActAppReturnVendorDataGet**
- **flxActAppReturnVendorDataSet**

The following functions allow you to manipulate custom data in a repair object:

- **flxActAppRepairVendorDataGet**
- **flxActAppRepairVendorDataSet**

See the *C/C++ Function Reference* for information about using these functions.

Local Activation

A local activation retrieves information for the activation from an activation specification record (ASR) on the FlexEnabled client. The following description gives the sequence of operations required for this type of activation.

Task ***To create the sequence of Activation API calls for local activation***

1. Call **flxActCommonRepairLocalTrustedStorage** to ensure that the ASR can be written to trusted storage. Alternatively, this step can be performed after an attempt to perform a local activation fails.
2. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container.
3. Add the fulfillment records from valid ASRs to trusted storage with **flxActCommonLicSpcAddASRs** or **flxActCommonLicSpcAddASRFromBuffer**. ASRs in the specified location are read in turn, and the following steps carried out:
   a. Entries in the DataDictionary Element are read (see Appendix A, DataDictionary Element).
   b. If publisher rules for local activation are defined, the rules are applied. For example, rules can determine the number of times a particular fulfillment record can be loaded into trusted storage or whether the ASR can be used in a virtual-only or a physical-only environment. When the ASR does not comply with the rules, the fulfillment record is not added to the license specification.
   c. If a duration is defined, the expiry date is calculated and the fulfillment record is modified.
   d. When all the ASRs have been processed, the fulfillment records in the license specification container are written to trusted storage.
4. Delete the license specification container with **flxActCommonLicSpcDelete**.

An example of code that implements local activation is in the example application activation utility, `appActUtil.c`. See the function **sHandleLocalActivation** that installs fulfillment records from the ASRs in a specified directory or buffer into trusted storage.

Note • *For a product that requires both local and short-code transactions, create separate ASRs for local and short-code transactions. A short-code ASR (that is, an ASR with a* `ShortCodeData` *element), cannot be used for local activation. Ensure that ASRs used for local activation do not include a* `ShortCodeData` *element.*

Fulfillment Record Manipulation

The following sections describe how to use the activation API to perform the following functions:

- View the Fulfillment Record from an Activation Response
- Display License Rights in Client-Side Trusted Storage
- Display Server-Side License Rights Remotely On the Client
- Delete Multiple Fulfillment Records from Trusted Storage
- Delete Multiple Fulfillment Records from Trusted Storage
- Enable Fulfillment Record when No Response to Return Request

View the Fulfillment Record from an Activation Response

When an activation request is processed successfully, a response is returned containing the fulfillment record.

If you want the activation utility to check what license rights were granted or to get the vendor data sent by the activation server, use the following function:

```
FlxActBool flxActAppActivationRespLicSpcGet(FlxActAppActivation act, FlxActProdLicSpc *pProdSpc)
```

This function takes the activation object and assigns a product-license specification object to a pointer identified by the second argument. You can then use other functions that operate on the product-license specification object to get information about the fulfillment record. Do not change the properties of the product-license specification object.

Call this function any time after you call the **flxActAppActivationRespProcess**, but before you call **flxActCommonHandleClose**.

Display License Rights in Client-Side Trusted Storage

The following description gives a possible sequence of operations using the Activation API to display the contents of the application's trusted storage, that is, details of activated licenses available on a FlexEnabled client.

Task ***To display license rights in trusted storage***

1. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.

2. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specification objects it contains. One product license specification object corresponds to one fulfillment record.
3. Iterate through all product license specifications in turn. For each specification:
   a. Get each product license specification object at the specified index using **flxActCommonLicSpcGet**.
   b. Extract required information from the fulfillment record. The following information in a fulfillment record is available. Use the appropriate method from the Activation API:
      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - License group counts—**flxActCommonLicSpcCountGet**
      - Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
      - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
      - Expiry date—**flxActCommonProdLicSpcExpDateGet**
      - Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
      - Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
      - Trust flags—**flxActCommonProdLicSpcTrustFlagsGet**
      - Disabled because return in progress—**flxActCommonProdLicSpcIsDisabled**
      - Fulfillment type—**flxActCommonProdLicSpcFulfillmentTypeGet**
      - Server chain—**flxActCommonProdLicSpcActServerChainGet**
   c. Format the extracted information and output it.
4. Release the license specification object resources.

   Delete the license specification object with **flxActCommonProdLicSpcDelete**.

Note • *The setting of the mode parameter to* `FLX_ACT_APP` *with* **flxActCommonHandleOpen** *defines that this process outputs details of licenses held in the application's trusted storage (client-side trusted storage).*

The function **sDumpTS** in the example application activation utility, `appActUtil.c`, displays the license rights held in client-side trusted storage. This function is effectively called by the `-view` switch and controlled by two switches: `-valid`, which outputs details of all fulfillment records that are from trusted sections of trusted storage, and `-broken`, which outputs details of all fulfillment records that are from untrusted sections of trusted storage.

Display Server-Side License Rights Remotely On the Client

The following description gives a sequence of operations that use Activation API to read license rights held in the license server's trusted storage and display them remotely on the FlexEnabled client.

Task ***To display license rights in server-side trusted storage***

1. Remotely read fulfillment records from the license server's trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container on the client.
   b. Call either function to read fulfillment records from the license server's trusted storage and then add these records to the license specification container as product license specification objects:
      - **flxActCommonLicSpcPopulateFromServerTS** to read only the *trusted* fulfillment records
      - **flxActCommonLicSpcPopulateAllFromServerTS** to read *all* fulfillment records
2. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specification objects it contains. One product license specification object corresponds to one fulfillment record.
3. Iterate through all product license specifications in turn. For each specification:
   a. Get each product license specification object at the specified index using **flxActCommonLicSpcGet**.
   b. Call **flxActCommonProdLicSpcPopulateFRFromServerTS** to load the entire fulfillment record that matches the fulfillment ID present in the product license specification obtained in Step 1b.
   c. Extract required information from the fulfillment record. The following information in a fulfillment record is available. Use the appropriate method from the Activation API:
      - Trust flags—**flxActCommonProdLicSpcTrustFlagsGet**
      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
      - Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
      - Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
      - Expiry date—**flxActCommonProdLicSpcExpDateGet**
      - Disabled because return in progress—**flxActCommonProdLicSpcIsDisabled**
      - License group counts—**flxActCommonLicSpcCountGet** (`CONCURRENT`, `CONCURRENT_OD`, `HYBRID`, `HYBRID_OD`, `ACTIVATABLE`, `ACTIVATABLE_OD`, or `REPAIRS`)
      - Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
      - Fulfillment type—**flxActCommonProdLicSpcFulfillmentTypeGet**
      - Server chain—**flxActCommonProdLicSpcActServerChainGet**
   d. Format the extracted information and output it.
4. Release the license specification object resources.

   Delete the license specification object with **flxActCommonProdLicSpcDelete**.

The function **sDumpServerTS** in the example application activation utility, `appActUtil.c`, displays the license rights read from server-side trusted storage. The function is called by the `-serverview` switch. (Set the mode parameter to `FLX_ACT_APP` for **flxActCommonHandleOpen**.)

Important • *The remote license server whose trusted storage you want to view must be running.*

Delete a Single Fulfillment Record from Trusted Storage

The following description gives a possible sequence of operations using the Activation API to delete a fulfillment record from trusted storage. To delete a fulfillment record from trusted storage:

1. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
2. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specification objects it contains. One product license specification object corresponds to one fulfillment record.
3. Iterate through all product license specifications in turn until the required fulfillment record is found. For each specification:
   a. Get each product license specification object at the specified index using **flxActCommonLicSpcGet**.
   b. Determine if this is the fulfillment record to be deleted. The following information in a fulfillment record is available. Use the appropriate method from the Activation API:
      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - Entitlement ID—**flxActCommonProdLicSpcEntitlemetIdGet**
      - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
      - Expiry date—**flxActCommonProdLicSpcExpDateGet**
      - Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
      - Fulfillment Type—**flxActCommonProdLicSpcFulfillmentTypeGet**
4. Call **flxActCommonLicSpcProductDelete** to delete the selected fulfillment record.
5. Release the license specification object resources.
6. Delete the license specification object with **flxActCommonLicSpcDelete**.

Note • *The setting of the mode parameter to* `FLX_ACT_APP` *with* **flxActCommonHandleOpen** *defines that this process outputs details of licenses held in the application's trusted storage (client-side trusted storage).*

The function **sHandleDelete** in the sample application activation utility source, `appActUtil.c`, deletes the fulfillment record with the specified fulfillment ID. This function is effectively called by the `-delete` switch. You must specify the fulfillment ID of the record to be deleted when using this switch:

```
appactutil -delete fulfillment ID
```

Delete Multiple Fulfillment Records from Trusted Storage

The **flxActCommonHandleDeleteProduct** function deletes all fulfillment records for a specified product from trusted storage.

Note • *If you set the* `FLX_ACT_APP` *attribute using the* **flxActCommonHandleOpen** *function, the* **flxActCommonHandleDeleteProduct** *function deletes fulfillment records held in the client-side trusted storage.*

The function **sHandleDeleteProductFromTS**, in the example application activation utility, `appActUtil.c`, deletes fulfillment records with the specified product ID. This function is effectively called by the `-delproduct` switch. When using this switch, use the product ID to identify the product:

```
appactutil -delproduct product ID
```

Enable Fulfillment Record when No Response to Return Request

When a return request is sent, the fulfillment record that is being returned is disabled. The client then waits until a response is received from the activation server. In the case where no response or an invalid response is received, the fulfillment record remains disabled and the license rights it contains are unavailable. The **flxActAppReturnCancel** function enables the fulfillment record and makes the license rights available.

For example code that demonstrates the use of this function, see the sample activation utility, `appactutil`. The following `appactutil` command demonstrates the use of **flxActAppReturnCancel**.

```
appactutil -return <fulfillmentID> -cancel
```

Transactions with a License Server

The following sections describe how to use the activation API to implement transactions with a license server:

- Activation from a License Server
- Return Fulfillment Record to a License Server
- Repair from a License Server

Activation from a License Server

The following description gives the sequence of operations required to perform an activation transaction with a license server. A successful activation transaction results in license rights being installed in the application's trusted storage on the FlexEnabled client.

Task ***To perform an activation transaction from a license server***

1. Create an application activation object with **flxActAppActivationCreate**.
2. Add activation request parameters to the application activation object.
3. Set the following mandatory request parameters:
   - *Entitlement ID*—Use **flxActAppActivationEntitlementIdSet** to set the entitlement ID. This identifies the entitlement for which the license rights are being requested.
   - *Expiration date*—Use either **flxActAppActivationExpDateSet** to enter a specific expiration date for the license rights or **flxActAppActivationDurationSet** to enter the relative duration.

   Entitlement ID and license rights expiration are determined by external means.
4. Add the license server address to the application activation object.
   a. Use **flxActCommonHandleSetCommType** with `commType` of `flxCommsMvsnFlex` to set the communication link to a license server.
   b. Use **flxActCommonHandleSetRemoteServer** to set the address of the license server.
5. Send the activation request to a license server and process the response with **flxActAppActivationSend**. This function returns success when a fulfillment record is successfully added to trusted storage. It returns a non-success when the transaction fails for any reason.
6. Delete the application activation object with **flxActAppActivationDelete**.

In the example application activation utility, `appActUtil.c`, the function **sHandleActivation** includes the functionality to obtain the license rights specified from a license server.

Return Fulfillment Record to a License Server

The following description gives a typical sequence of operations required to return license rights to a license server.

Task ***To return license rights to a license server***

1. Create an application return object with **flxActAppReturnCreate**.
2. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
3. Examine fulfillment records and choose the one that is to be returned.
   a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.

   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to choose the one to be returned. The following elements in a fulfillment record can be examined:

   - Product ID—**flxActCommonProdLicSpcProductIdGet**
   - License group counts—**flxActCommonProdLicSpcCountGet**
   - Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
   - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
   - Expiry date—**flxActCommonProdLicSpcExpDateGet**
   - Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
   - Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
   - Fulfillment Type—**flxActCommonProdLicSpcFulfillmentTypeGet**

4. Set the return request parameters by adding the fulfillment record that is to be returned to the return object with **flxActAppReturnProdLicSpcSet**.

5. Add the license server address to the application return object.

   a. Use **flxActAppReturnProdLicSpcSet** with `commType` of `flxCommsMvsnFlex` to set the communication link to a license server.

   b. Use **flxActCommonHandleSetRemoteServer** to set the address of the license server.

6. Send the return request to a license server and process the response with **flxActAppReturnSend**. When a successful response is received, the fulfillment record is deleted from the application's trusted storage.

7. Release resources.

   a. Delete the license specification container with **flxActCommonLicSpcDelete**.

   b. Delete the application return object **flxActAppReturnDelete**.

In the example application activation utility, `appActUtil.c`, the function **sHandleReturn** includes the functionality to return the fulfillment record, which is specified using its fulfillment ID, to a license server.

Repair from a License Server

The need for a repair of client-side trusted storage is indicated by the FlexEnabled application. When a checkout of a license held in trusted storage is attempted and fails because a repair of trusted storage is required, the Flexible API returns a `LM_REPAIR_NEEDED` error. Code in the FlexEnabled application can then call the application activation utility directly or indicate that a repair of trusted storage is required.

Task ***To repair a single fulfillment record***

1. Create a repair object with **flxActAppRepairCreate**.

2. Read fulfillment records from trusted storage.

   a. Call **flxActCommonLicSpcCreate** to create a license specification container.

   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.

3. Examine fulfillment records and find the one that requires repair.

   a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.

   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to find the one that requires repair. Use **flxActCommonProdLicSpcTrustFlagsGet** to determine whether a fulfillment record requires repair. You can also use other **flxActCommonProdLicSpc*Get** functions to identify the specific fulfillment record that requires repair.

4. Set the repair request parameters by adding the fulfillment record that requires repair to the repair object with **flxActAppRepairProdLicSpcSet**.

5. Add the license server address to the repair object.

   a. Use **flxActCommonHandleSetCommType** with `commType` of `flxCommsMvsnFlex` to set the communication link to a license server.

   b. Use **flxActCommonHandleSetRemoteServer** to set the address of the license server.

6. Send the repair request to a license server and process the response with **flxActAppRepairSend**. If the license server allows the repair, the fulfillment record is repaired. If the license server does not allow the repair, then the fulfillment record is not repaired and **flxActAppRepairSend** returns nonsuccess.

7. Release resources.

   a. Delete the license specification container with **flxActCommonLicSpcDelete**.

   b. Delete the repair object **flxActAppRepairDelete**.

In the example application activation utility, `appActUtil.c`, the function **sHandleRepair** includes the functionality to repair the fulfillment record with the specified fulfillment ID if it requires repair.

Using the Manual Response Generator

The manual response generator can be used for testing. It provides an illustration of a manual transaction using XML files where request and response messages are stored as XML files. The files can be transmitted by any suitable means, for example email, however the entire file must be transmitted unaltered.

The following sections describe how to use the activation API to implement transactions using the manual response generator:

- Activation from a Manual Response Generator
- Return Fulfillment Record to Manual Response Generator
- Return Fulfillment Record to Manual Response Generator

Activation from a Manual Response Generator

The following description outlines the sequence of operations required to perform an activation from a manual response generator.

Task ***To perform an activation from a manual response generator***

1. Create an application activation object with **flxActAppActivationCreate**.
2. Add activation request parameters to the application activation object.

   Set the following mandatory request parameters:

   - **Entitlement ID**—Use **flxActAppActivationEntitlementIdSet** to set the entitlement ID. This identifies the entitlement for which the license rights are being requested.
   - **Expiration Date**—Use either **flxActAppActivationExpDateSet** to enter a specific expiration date or **flxActAppActivationDurationSet** to enter the relative duration.

   Entitlement ID and the expiration date are determined by external means.
3. Optionally, add vendor data to the activation object. Use **flxActAppActivationVendorDataSet** to specify key value pairs that are added to the activation request. For an overview, see Using the Vendor Dictionary to Send Custom Data.
4. Create an activation request using **flxActAppActivationReqSet**.
5. Output the activation request.
6. Clean up memory and delete temporary files. Delete application activation object using **flxActAppActivationDelete**.
7. Copy the activation request and paste it into the manual response generator.
8. Use the manual response generator to create the required response, either a config response or an activation response. For instructions on operating the manual response generator, see Chapter 14, "The Manual Response Generator."
9. Create an application activation object with **flxActAppActivationCreate**.
10. Input the response and process it using **flxActAppActivationRespProcess**. If the response contains a Trusted Configuration, it is used to create the requested trusted section in trusted storage. If the response contains a fulfillment record, it is placed in trusted storage. The parameter `bIsTrustedConfig` records whether the response contained a Trusted Configuration or a fulfillment record.
11. Clean up memory and delete temporary files. Delete the application activation object with **flxActAppActivationDelete**.
12. If the activation response contained a Trusted Configuration, repeat Step 1 through Step 11. The activation request created in Step 4 contains details of the newly created section of trusted storage. Then, at Step 8, an activation response that contains the fulfillment record requested can be generated (as the required trusted section is available on the requesting system).

In the sample application activation utility, `appActUtil.c`, the function **sHandleActivation** includes the functionality to generate an activation request as an XML file (Step 1 through Step 6). The function **sHandleManualProcessing** inputs the response from the manual response generator and processes it (Step 9 through Step 11). Step 12 is not demonstrated in the example activation utility.

Return Fulfillment Record to Manual Response Generator

The following description gives in outline the sequence of operations required to return a fulfillment record to a manual response generator.

Task ***To return a fulfillment record to a manual response generator***

1. Create an application return object with **flxActAppReturnCreate**.
2. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
3. Examine fulfillment records and choose the one that is to be returned.
   a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.
   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to choose the one to be returned. The following elements in a fulfillment record can be examined:
      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - License group counts—**flxActCommonProdLicSpcCountGet**
      - Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
      - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
      - Expiry date—**flxActCommonProdLicSpcExpDateGet**
      - Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
      - Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
      - Fulfillment type—**flxActCommonProdLicSpcFulfillmentTypeGet**
4. Set the return request parameters by adding the fulfillment record that is to be returned to the return object with **flxActAppReturnProdLicSpcSet**. Optionally, add vendor data to the return object. Use **flxActAppReturnVendorDataSet** to specify key value pairs that are added to the return request. For an overview, see Using the Vendor Dictionary to Send Custom Data.
5. Create a return request using **flxActAppReturnReqSet**.
6. Output the return request.
7. Clean up memory and delete temporary files. Delete return object using **flxActAppReturnDelete**.
8. Copy the return request and paste it into the manual response generator.
9. Use the manual response generator to create the required response, either a config response or a return response. For instructions on operating the manual response generator, see Chapter 14, “The Manual Response Generator.”
10. Create an application return object using **flxActAppActivationCreate**.

11. Input the response and process it using **flxActAppReturnRespProcess.**

    If the response contained a Trusted Configuration, it is used to do one of the following:

    a. Update the settings for an existing trusted section. This is the more probable scenario for a Trusted Configuration in a return response.

    b. Create a trusted section. If the return response reported success, then the fulfillment record is deleted from the application's trusted storage.

    The parameter `bIsTrustedConfig` records whether the response contained a Trusted Configuration or the result of the return request.

12. Clean up memory and delete temporary files. Delete the application activation object with **flxActAppReturnDelete**.

13. If the return response contained a Trusted Configuration, repeat Step 1 through Step 12.

In the sample application activation utility, `appActUtil.c`, the function **sHandleReturn** includes the functionality to generate a return request as an XML file (Step 1 through Step 5). The function **sHandleManualProcessing** inputs the response from the manual response generator and processes it (Step 10 through Step 12). Step 13 is not demonstrated in the example activation utility.

Repair Using a Manual Response Generator

The need for a repair of client-side trusted storage is indicated by the FlexEnabled application. When a checkout of a license held in trusted storage is attempted and fails because a repair of trusted storage is required, the Flexible API returns the error `LM_REPAIR_NEEDED`. Code in the FlexEnabled application can then call the application activation utility directly or indicate that a repair of trusted storage is required.

The following description gives in outline the sequence of operations required to repair a fulfillment record using a manual response generator.

Task ***To repair a single fulfillment record using the manual response generator***

1. Create a repair object with **flxActAppRepairCreate**.

2. Read fulfillment records from trusted storage.

    a. Call **flxActCommonLicSpcCreate** to create a license specification container.

    b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.

3. Examine fulfillment records and find the one that requires repair.

    a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.

    b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to find the one that requires repair. Use **flxActCommonProdLicSpcTrustFlagsGet** to determine whether a fulfillment record requires repair. You can also use other **flxActCommonProdLicSpc*Get** functions to identify the specific fulfillment record that requires repair.

4. Set the repair request parameters by adding the fulfillment record that requires repair to the repair object with **flxActAppRepairProdLicSpcSet**. Optionally, add vendor data to the repair object. Use **flxActAppRepairVendorDataSet** to specify key value pairs that are added to the repair request. For an overview, see Using the Vendor Dictionary to Send Custom Data.
5. Create a repair request using **flxActAppRepairReqSet**.
6. Output the repair request.
7. Clean up memory and delete temporary files. Delete the return object using **flxActAppRepairDelete**.
8. Copy the repair request and paste it into the manual response generator.
9. Use the manual response generator to create the required response: either a config response or a repair response. For instructions on operating the manual response generator, see Chapter 14, "The Manual Response Generator."
10. Create a repair object using **flxActAppRepairCreate**.
11. Input the response and process it using **flxActAppRepairRespProcess**.

    If the response contained a Trusted Configuration, it is used to do one of the following:

    a. Update the settings for an existing trusted section. This is the more probable scenario for a Trusted Configuration in a repair response.

    b. Create a trusted section.

    If the response is a repair response, then the fulfillment record is repaired.

    If the response reports an error, then the fulfillment record is not repaired and **flxActAppRepairRespProcess** returns non success. An error response is returned when the repair is not allowed and also to report an error in the transaction.

    The parameter `bIsTrustedConfig` records whether the response contained a Trusted Configuration or the result of the repair request.
12. Clean up memory and delete temporary files. Delete the repair object with **flxActAppRepairDelete**.
13. If the repair response contained a Trusted Configuration, repeat Step 1 through Step 12.

In the example application activation utility, `appActUtil.c`, the function **sHandleRepair** includes the functionality to generate a repair request as an XML file (Step 1 through Step 6). The function **sHandleManualProcessing** inputs the response from the manual response generator and processes it (Step 10 through Step 12). Step 13 is not demonstrated in the example activation utility.

Transactions with an Operations Server

The following sections describe how to use the activation API to implement transactions with a license server:

- Activation from an Operations Server
- Return to an Operations Server
- Repair from an Operations Server

Activation from an Operations Server

The following description gives the sequence of operations required to perform an activation transaction with a FlexNet Operations server. A successful activation transaction results in license rights being installed in the application's trusted storage on the FlexEnabled client.

Task ***To perform an activation transaction with a FlexNet Operations server***

1. Create an application activation object with **flxActAppActivationCreate**.
2. Add activation request parameters to the application activation object. The request parameters required depend on the processing logic in the Operations server. See the FlexNet Operations server documentation for details.

   Optionally, add vendor data to the activation object. Use **flxActAppActivationVendorDataSet** to specify key value pairs that are added to the activation request. For an overview, see Using the Vendor Dictionary to Send Custom Data.
3. Add the operations server address to the application activation object.
   a. Use **flxActAppCommonHandleSetCommType** with `commType` of `flxCommsMvsnSoap` to set the communications link to an Operations server.
   b. Use **flxActAppCommonHandleSetRemoteServer** to set the address of the Operations server.
   c. Optionally, provide settings and information for this communications link: **flxActCommonHandleSetTimeout**, **flxActAppCommonHandleSetProxyDetails**, or **flxActAppCommonHandleSetSSLDetails**.
   d. Optionally, set the activation reason using **flxActAppActivationReasonSet**.
   e. Optionally, add a callback, **flxActAppCommonHandleSetStatusCallback**, which reports status changes during a transaction. This also allows functionality to be added that enables the end user to cancel the current transaction.
4. Send the activation request to an Operations server and process the response with **flxActAppActivationSend**. This function returns success when a fulfillment record is successfully added to trusted storage. It returns a non-success when the transaction fails for any reason.
5. Delete the application activation object with **flxActAppActivationDelete**.

In the example application activation utility, `appActUtil.c`, the function **sHandleActivation** includes the functionality to obtain the license rights specified from an Operations server when the switch `-comm soap` is used.

Return to an Operations Server

The following description gives a typical sequence of operations required to return license rights to an Operations server.

Task ***To return license rights to an Operations server***

1. Create an application return object with **flxActAppReturnCreate**.
2. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.

b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.

3. Examine fulfillment records and choose the one that is to be returned.

   a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.

   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to choose the one to be returned. The following elements in a fulfillment record can be examined:

      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - License group counts—**flxActCommonProdLicSpcCountGet**
      - Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
      - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
      - Expiry date—**flxActCommonProdLicSpcExpDateGet**
      - Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
      - Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
      - Fulfillment type—**flxActCommonProdLicSpcFulfillmentTypeGet**

   c. Set the return request parameters by adding the fulfillment record that is to be returned to the return object with **flxActAppReturnProdLicSpcSet**.

   d. Optionally, add vendor data to the return object. Use **flxActAppReturnVendorDataSet** to specify key value pairs that are added to the return request. For an overview, see Using the Vendor Dictionary to Send Custom Data.

   e. Optionally, set the return reason using **flxActAppReturnReasonSet**.

4. Add the Operations server address to the application return object.

   a. Use **flxActCommonHandleSetCommType** with `commType` of `flxCommsMvsnSoap` to set the communication link to a license server.

   b. Use **flxActCommonHandleSetRemoteServer** to set the address of the Operations server.

5. Send the return request to an Operations server and process the response with **flxActAppReturnSend**. When a successful response is received, the fulfillment record is deleted from the application's trusted storage.

6. Release resources.

   a. Delete the license specification container with **flxActCommonLicSpcDelete**.

   b. Delete the application return object **flxActAppReturnDelete**.

In the example application activation utility, `appActUtil.c`, the function **sHandleReturn** includes the functionality to return the fulfillment record, specified using its fulfillment ID, to an Operations server when the switch `-comm` soap is used.

Repair from an Operations Server

The need for a repair of client-side trusted storage is indicated by the FlexEnabled application. When a checkout of a license held in trusted storage is attempted and fails because a repair of trusted storage is required, the Flexible API returns a `LM_REPAIR_NEEDED` error. Code in the FlexEnabled application can then call the application activation utility directly or indicate that a repair of trusted storage is required. The following description gives a typical sequence of operations required to repair a single fulfillment record.

Task ***To repair a single fulfillment record using an Operations server***

1. Create a repair object with **flxActAppRepairCreate**.
2. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
3. Examine fulfillment records and find the one that requires repair.
   a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.
   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to find the one that requires repair. Use **flxActCommonProdLicSpcTrustFlagsGet** to determine whether a fulfillment record requires repair. You can also use other **flxActCommonProdLicSpc*Get** functions to identify the specific fulfillment record that requires repair.
   c. Set the repair request parameters by adding the fulfillment record that requires repair to the repair object with **flxActAppRepairProdLicSpcSet**. Optionally, add vendor data to the repair object. Use **flxActAppRepairVendorDataSet** to specify key value pairs that are added to the repair request. For an overview, see Using the Vendor Dictionary to Send Custom Data.
4. Add the Operations server address to the repair object.
   a. Use **flxActCommonHandleSetCommType** with `commType` of `flxCommsMvsnSoap` to set the communication link to a license server.
   b. Use **flxActCommonHandleSetRemoteServer** to set the address of the Operations server.
5. Send the repair request to an Operations server and process the response with **flxActAppRepairSend**. If the Operations server allows the repair, the fulfillment record is repaired. If the Operations server does not allow the repair, then the fulfillment record is not repaired and **flxActAppRepairSend** returns non success.
6. Release resources.
   a. Delete the license specification container with **flxActCommonLicSpcDelete**.
   b. Delete the repair object **flxActAppRepairDelete**.

In the example application activation utility, `appActUtil.c`, the function **sHandleRepair** includes the functionality to repair the fulfillment record with the specified fulfillment ID if it requires repair.

11

Building a Server Activation Utility for Single-Action Transactions

The server activation utility manages license rights on an enterprise license server. The utility uses the Activation API to initiate and manage activation transactions between the license server and either the publisher's activation server, a FlexEnabled client, or another license server.

About Single-Action Transactions

This chapter describes the steps for creating and deploying a server activation utility that uses *single-action transactions*. This type of transaction performs one activation action (for example, one activation, repair, or return for one fulfillment record) per transaction. Single-action transactions can be used to perform activations between the publisher's activation server and the license server. However, best practice is always to use composite transactions.

If your server activation utility uses single-action transactions, you should use the information in this chapter to create and provide a publisher-specific version of this utility. An example server activation utility is provided with the FlexNet Publisher Licensing Toolkit in the following location:

```
<platform_dir>/serveractutil.exe
```

The corresponding source code can be found at this location:

```
<install_dir>/examples/activation/serverActUtil.c
```

Activation Utilities for Composite Transactions

This chapter focuses on building server activation utilities that support single-action transactions. For information about building activation utilities that support composite transactions, see Chapter 12, "Building an Activation Utility for Composite Transactions."

Path and Directory Conventions Used in This Chapter

References to directories and paths in this chapter use the following conventions:

- The term `<install_dir>` is used to represent the path and directory in which your FlexNet Publisher toolkit is installed.
- The term `<platform_dir>` is used to refer to the path and directory in which the platform-specific components of the FlexNet Publisher Licensing Toolkit are located. For example, for a Windows 32-bit system, `<platform_dir>` represents `<install_dir>\i86_n3`. For a Sun Microsystems 32-bit Sparc running Solaris 10.*x*, `<platform_dir>` represents `<install_dir>/sun4_u10`.
- File paths for locations common to both UNIX and Windows platforms are constructed using forward slashes.

Creating an Activation Utility

Use the server-specific and common API functions defined in the activation API when creating an server activation utility. These are the functions that can be used to create and access the server-side trusted storage. The basic structure of the utility is described below.

Task ***To create the sequence of Activation API calls in a server activation utility***

1. Install the FlexNet Licensing Service (if it is required by the platform and has not been previously installed on the machine).

   Note • *The FlexNet Licensing Service can be installed by the FlexEnabled application installer or by a separate utility that runs during the application initialization. Publishers must develop this capability within their application. See Chapter 6, "Initializing Trusted Storage," for more information.*

2. Call **flxActCommonLibraryInit** to initialize the library. This must occur before any other call to the Activation API. It loads the activation library.

   Note • *In trusted storage–based applications,* **flxActCommonLibraryInit** *is the same as the Flexible API function* **lc_flexinit** *(described in Chapter 5, Adding Function Calls to a FlexEnabled Application). If you are using both Activation API and the Flexible API in this utility, ensure that you call either* **flxActCommonLibraryInit** *or* **lc_flexinit,** *but not both.*

3. Open a server activation handle by calling **flxActCommonHandleOpen** with the mode parameter set to `FLX_ACT_SVR`. This sets up access to server-side trusted storage. Server-side trusted storage is used to store license rights served to FlexEnabled applications by a license server.
4. Design logic to initiate and manage one or more activation transactions and to access fulfillment records in trusted storage. See Typical Transactions.
5. Close the activation handle by calling **flxActCommonHandleClose**, which releases all resources associated with it.

6. Release the Activation API by calling **flxActCommonLibraryCleanup** to shutdown the shared component. No other Activation API routines can be called after this point.

Important • *Note the following about* **flxActCommonLibraryCleanup***:*

- *This function is also performed by the Flexible API function* **lc_flexinit_cleanup** *(described in Chapter 5, Adding Function Calls to a FlexEnabled Application). If your utility previously called* **lc_flex_init** *(see Step 2), use* **lc_flexnit_cleanup** *here, not* **flxActCommonLibraryCleanup***.*
- *On Windows platforms, do not call* **flxActCommonLibraryCleanup** *or* **flxActCommonLibraryInit** *in* `DllMain`*. These APIs result in calls to the Microsoft Win32 FreeLibrary and LoadLibrary APIs. Calls to these Win32 APIs in* `DllMain` *are widely known to be associated with issues.*

Typical Transactions

The following is a list of typical transactions, each of which is described in subsequent sections:

- Using the Vendor Dictionary for Custom Data
- Transactions with a FlexNet Operations Server
- Activation from a FlexNet Operations Server
- Return to a FlexNet Operations Server
- Repair from a FlexNet Operations Server
- Transfer of Licenses from a License Server
- Return a Fulfillment Record to a License Server
- Activation From a Manual Response Generator
- Return of Fulfillment Record to Manual Response Generator
- Repair From a Manual Response Generator

Fulfillment Record Extraction Functions

The following sections describe functions for extracting information from records in trusted storage:

- Display Current License Rights in Trusted Storage
- Display License Rights Activated by This Server
- Display Complete Contents of Trusted Storage
- Enable Fulfillment Record when No Response to Return Request
- Display License Rights Held in Remote Server-Side Trusted Storage

Building and Deploying Your Activation Utility

Task ***To build and deploy a server activation utility***

1. Compile and link the server activation utility.

   An sample makefile is included in the toolkit (`.../<platform_dir>/makefile.act`). Use the build steps for `serveractutil.exe` as an example. The sample build steps include information about compilation, linking, and the use of `preptool` (see step 2).

2. For standard toolkits, use `preptool` to prepare the server activation utility and to generate the publisher-specific library for use by the server activation utility. See Chapter 13, "Working with the Software Publisher Utilities," especially the `preptool` section.

   Note • *If you have created your own publisher-specific version of* `preptool`*, use that version (which might have a different name).*

   For `notr` toolkits, use the activation library component, `demo_server_libFNP_notr.*`, which was created when you built the toolkit.

3. Set up your product installer to install the utility and associated components in the same directory. The following components are required:

   - The prepped version of the server activation utility
   - The activation library created in step 2
   - For Windows, the FlexNet Licensing Service installer component (`FNP_Act_Installer.dll`), needed for installation of the Licensing service, whether this is performed as part of the product installation or separately

     Note • *Ensure that the correct version of* `FNP_Act_Installer.dll` *is added to the FlexEnabled application. For more details, see the section "The FlexNet Licensing Service Component" in the Development Environment Guide.*

   - To enable transactions with an FlexNet Operations server, the `FnpCommsSoap` module

   Optionally, the product installer can install the FlexNet Licensing Service. (This is the recommended method for installing the Licensing service.) To install this service, the product installer must do the following:

   - For Windows platforms, use the API provided by the FlexNet Licensing Service installer component (`FNP_Act_Installer.dll`). For more instructions see Chapter 6, "Initializing Trusted Storage."

     Note • *If your product installer is InstallShield-based, you can find examples of installing the Licensing Service in* `<platform_dir>\examples\serviceinstall`.

   - For OS X and UNIX platforms, run the `install_fnp.sh` script.

Sending a Reason with a Request Message

With activation requests and return requests sent as XML-formatted messages, the activation utility can send a numeric value indicating the reason for the transaction. Software publishers must define their own values and reasons, and then define business rules that map these items. By default, this data is not captured in a request, but must be programmed into the activation utility. You cannot send this information when performing short-code transactions.

Use the following functions in the activation API allow you to manipulate this information in an activation object:

- **flxActSvrActivationReasonGet**
- **flxActSvrActivationReasonSet**

Use the following functions in the activation API allow you to manipulate this information in a return object:

- **flxActSvrReturnReasonGet**
- **flxActSvrReturnReasonSet**

See the FlexNet Publisher document titled *C/C++ Function Reference* for information about using these functions.

Using the Vendor Dictionary for Custom Data

This is an optional element in transaction requests and responses. It allows you to send additional data about the transaction. For example, it could be used to send registration information supplied by users when they request licenses. During a repair transaction, it can be used to send additional information about the user or the license server.

Vendor data can be added to both requests and responses. When vendor data is available in a response it is stored, together with the associated fulfillment record, in trusted storage. Then the vendor data can be read from trusted storage using Activation API functions.

The vendor dictionary data is stored in an XML file. The first element in an entry is the **Key** element that identifies the type of data in this entry. The second element of the entry, the **Value** element, contains the data for the entry. This type of entry is referred to here and elsewhere as a *key-value pair*.

It is visible in requests as the VendorDictionary element. The vendor data contained within an request can be viewed using the GUI version of the manual response generator, `responsegenUI.exe`. If you are using the command line version, `responsegen.*`, you will have to examine the request before using the manual response generator to generate the appropriate response. For details, see Chapter 14, “The Manual Response Generator.” See the FlexNet Operations Server documentation for information about how vendor data is used by the FlexNet Operations Server.

The following functions allow you to manipulate custom data:

- **flxActSvrActivationVendorDataGet**
- **flxActSvrActivationVendorDataSet**
- **flxActSvrReturnVendorDataGet**
- **flxActSvrReturnVendorDataSet**
- **flxActSvrRepairVendorDataGet**
- **flxActSvrRepairVendorDataSet**

Local Activation

Local activation on the license server retrieves information to load licenses from an activation specification record (ASR) installed on the server. To implement the local activation on the server, use the same sequence of operations used for local activation on the FlexEnabled client. (See Local Activation in Chapter 10, "Local Activation.")

Example code for implementing server-side local activation is found in `<install_dir>/examples/activation/server/serverActUtil.c` (the source file for the sample server activation utility, `serverActUtil`). Note especially the function `sHandleLocalActivation`, which installs fulfillment records from the ASRs in a specified directory or buffer into trusted storage.

Transactions with a FlexNet Operations Server

The following sections describe how to use the activation API to implement transactions with a FlexNet Operations server:

- Activation from a FlexNet Operations Server
- Return to a FlexNet Operations Server
- Repair from a FlexNet Operations Server

Important • *Although activations from the FlexNet Operations server support single-action transactions, best practice is to use composite transactions for such activations. The following sections describe how to build functionality into an activation utility that supports single-action transactions. To build a server activation utility that supports composite transactions, see Chapter 12, "Building an Activation Utility for Composite Transactions."*

Activation from a FlexNet Operations Server

The following description gives the sequence of operations required to perform an activation transaction with a FlexNet Operations server. A successful activation transaction results in license rights being installed in trusted storage on the license server.

Task ***To perform an activation transaction with a FlexNet Operations server***

1. Create a server activation object with **flxActSvrActivationCreate**.
2. Add activation request parameters to the server activation object.
3. The request parameters required depend on the processing logic in the Operations server. See the documentation for the Operations server for details. Optionally add vendor data to the activation object. Use **flxActSvrActivationVendorDataSet** to specify key value pairs that are added to the activation request. For an overview, see Using the Vendor Dictionary for Custom Data.
4. Add the operations server address to the activation object.
   a. Use **flxActCommonHandleSetCommType** with `commType` of `flxCommsMvsnSoap` to set the communications link to a FlexNet Operations server.
   b. Use **flxActCommonHandleSetRemoteServer** to set the address of the Operations server.

c. Optionally, provide settings and information for this communications link: **flxActCommonHandleSetTimeout**, **flxActCommonHandleSetProxyDetails**, or **flxActCommonHandleSetSSLDetails**.

d. Optionally, add a callback, **flxActCommonHandleSetStatusCallback**, that reports status changes during a transaction. This also allows functionality to be added that enables the end user to cancel the current transaction.

5. Send the activation request to a FlexNet Operations server and process the response with **flxActSvrActivationSend**. This function returns success when a fulfillment record is successfully added to trusted storage. It returns a non-success when the transaction fails for any reason.

6. Delete the activation object with **flxActSvrActivationDelete**.

In the example activation utility, `serverActUtil.c`, the function **sHandleActivation** includes the functionality to obtain the license rights specified from a FlexNet Operations server when the switch `-comm soap` is used.

Note • *On Windows systems, neither the flxActCommonHandleSetProxyDetails nor flxActCommonHandleSetSSLDetails functions are needed when the environment uses a proxy server and HTTPs communication, respectively.*

Return to a FlexNet Operations Server

The following description gives a typical sequence of operations required to return license rights to a FlexNet Operations server.

Task ***To return license rights to a FlexNet Operations server***

1. Create a server return object with **flxActSvrReturnCreate**.

2. Read fulfillment records from trusted storage.

   a. Call **flxActCommonLicSpcCreate** to create a license specification container.

   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.

3. Examine fulfillment records and choose the one that is to be returned.

   a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.

   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to choose the one to be returned. The following elements in a fulfillment record can be examined:

      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - License group counts—**flxActCommonProdLicSpcCountGet**
      - Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
      - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
      - Expiry date—**flxActCommonProdLicSpcExpDateGet**

- Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
- Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
- Fulfillment type—**flxActCommonProdLicSpcFulfillmentTypeGet**

4. Set the return request parameters by adding the fulfillment record that is to be returned to the return object with **flxActSvrReturnProdLicSpcSet**. Optionally, add vendor data to the return object. Use **flxActSvrReturnVendorDataSet** to specify key value pairs that are added to the activation request. For an overview, see Using the Vendor Dictionary for Custom Data.
5. Add the Operations server address to the return object.
   a. Use **flxActCommonHandleSetCommType** with `commType` of **flxCommsMvsnSoap** to set the communication link to a license server.
   b. Use **flxActCommonHandleSetRemoteServer** to set the address of the Operations server.
6. Send the return request to a FlexNet Operations server and process the response with **flxActSvrReturnSend**. When a successful response is received, the fulfillment record is deleted from the application's trusted storage.
7. Release resources.
   a. Delete the license specification container with **flxActCommonLicSpcDelete**.
   b. Delete the return object **flxActSvrReturnDelete**.
8. If the returned fulfillment record contains concurrent or hybrid licenses, update the counts in memory.

   Use `lmreread.exe` as described in *Programming Reference for License File–Based Licensing*. This updates the license group counts held in memory. For more information, see Chapter 2, "Transactions to a License Server," and the section FulfillmentDictionary Data in Server-side Trusted Storage in Chapter A, "XML Schemas."

In the example application activation utility, `serverActUtil.c`, the function **sHandleReturn** includes the functionality to return the fulfillment record, specified using its fulfillment ID, to a FlexNet Operations server when the switch `-comm soap` is used.

Repair from a FlexNet Operations Server

The need for a repair of server-side trusted storage is indicated by a message in the debug log. When the vendor daemon starts up, it reads hybrid and concurrent license counts from fulfillment records in trusted storage. Any untrusted fulfillment records that contain hybrid or concurrent license counts are not used and an error is logged to the debug log. Fulfillment records that contain activatable licenses and are untrusted are logged to the debug log when they are accessed, that is, when the vendor daemon is processing an activation, return or repair request.

Note • *Composite transactions in FlexNet Operations 12.7 support repair-all.*

The following description gives a typical sequence of operations required to repair a single fulfillment record.

Task ***To repair a single fulfillment record***

1. Create a repair object with **flxActSvrRepairCreate**.
2. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
3. Examine fulfillment records and find the one that requires repair.
   a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.
   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to find the one that requires repair. Use **flxActCommonProdLicSpcTrustFlagsGet** to determine whether a fulfillment record requires repair. You can also use other **flxActCommonProdLicSpc*Get** functions to identify the specific fulfillment record that requires repair.
4. Set the repair request parameters by adding the fulfillment record that requires repair to the repair object with **flxActSvrRepairProdLicSpcSet**. Optionally, add vendor data to the repair object: use **flxActSvrRepairVendorDataSet** to specify key value pairs that are added to the activation request. For an overview, see Using the Vendor Dictionary for Custom Data.
5. Add the Operations server address to the repair object.
   a. Use **flxActCommonHandleSetCommType** with `commType` of **flxCommsMvsnSoap** to set the communication link to a license server.
   b. Use**flxActCommonHandleSetRemoteServer** to set the address of the Operations server.
6. Send the repair request to a FlexNet Operations server and process the response with **flxActSvrRepairSend**. If the Operations server allows the repair, the fulfillment record is repaired. If the Operations server does not allow the repair, then the fulfillment record is not repaired and **flxActAppRepairSend** returns non success.
7. Release resources.
   a. Delete the license specification container with **flxActCommonLicSpcDelete**.
   b. Delete the repair object **flxActSvrRepairDelete**.

In the example server activation utility, `serverActUtil.c`, the function **sHandleRepair** includes the functionality to repair the fulfillment record with the specified fulfillment ID if it requires repair.

Transactions Between License Servers

License rights can be transferred between license servers. This functionality can be constrained using the hops element in the fulfillment record.

The following sections describe how to implement transactions between license servers:

- Transfer of Licenses from a License Server

- Return a Fulfillment Record to a License Server

Transfer of Licenses from a License Server

The following description gives the sequence of operations required to perform an activation transaction from one license server to another. A successful transaction results in license rights being transferred from a source license server to the destination license server that requested the license rights.

Task ***To transfer license rights***

1. Create a server activation object with **flxActSvrActivationCreate**.
2. Add activation request parameters to the activation object.

   Set the following mandatory request parameters:

   - **Entitlement ID**—Use **flxActSvrActivationEntitlementIdSet** to set the entitlement ID. This identifies the entitlement from which the license rights are being requested.
   - **License group counts**—Use **flxActSvrActivationCountSet** to set the quantity from each license group that will be transferred. At least one license group count must be set and when licenses of different types are required call **flxActSvrActivationCountSet** multiple times.
   - **Repair count**—Use **flxActSvrActivationCountSet** to request the number of repairs that are allowed for this fulfillment.

   Entitlement ID, expiration date, license group counts, and repair count are determined by external means.
3. Add the license server address to the application activation object.
   a. Use **flxActCommonHandleSetCommType** with `commType` of **flxCommsMvsnFlex** to set the communication link to a license server.
   b. Use **flxActCommonHandleSetRemoteServer** to set the address of the license server.
4. Send the activation request to a license server and process the response with **flxActSvrActivationSend**. This function returns success when a fulfillment record is successfully added to trusted storage, that is, license rights are transferred from a license server to this license server. It returns a non-success when the transaction fails for any reason.
5. Delete the activation object with **flxActSvrActivationDelete**.

In the example server activation utility, `serverActUtil.c`, the function **sHandleActivation** includes the functionality to obtain the license rights specified from a license server, that is, a transfer of license rights from one license server to another license server.

To enable the source or destination license server to view fulfillment records in the other server's trusted storage, see Display License Rights Held in Remote Server-Side Trusted Storage.

Return a Fulfillment Record to a License Server

The following description gives a typical sequence of operations required to return license rights to a license server.

Task ***To return license rights to a license server***

1. Create a server return object with **flxActSvrReturnCreate**.
2. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
3. Examine fulfillment records and choose the one that is to be returned.
   a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.
   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to choose the one to be returned. The following elements in a fulfillment record can be examined:
      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - License group counts—**flxActCommonProdLicSpcCountGet**
      - Repair count—**flxActCommonProdLicSpcCountGet**
      - Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
      - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
      - Expiry date—**flxActCommonProdLicSpcExpDateGet**
      - Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
      - Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
      - Fulfillment type—**flxActCommonProdLicSpcFulfillmentTypeGet**
4. Set the return request parameters by adding the fulfillment record that is to be returned to the return object with **flxActSvrReturnProdLicSpcSet**.
5. Add the license server address to the application activation object.
   a. Use **flxActAppReturnProdLicSpcSet** with `commType` of **flxCommsMvsnFlex** to set the communication link to a license server.
   b. Use **flxActCommonHandleSetRemoteServer** to set the address of the license server.
6. Send the return request to a license server and process the response with **flxActSvrReturnSend**. When a successful response is received, the fulfillment record is deleted. If an error response is received, then the fulfillment record status is changed to not disabled.
7. Release resources.
   a. Delete the license specification container with **flxActCommonLicSpcDelete**.
   b. Delete the return object **flxActSvrReturnDelete**.
8. If the returned fulfillment record contains concurrent or hybrid licenses, update the counts in memory.

Use `lmreread.exe` as described in *Programming Reference for License File–Based Licensing*. This updates the license group counts held in memory. For more information, see Chapter 2, "Transactions to a License Server," and the section FulfillmentDictionary Data in Server-side Trusted Storage in Chapter A, "XML Schemas."

In the example server activation utility, `serverActUtil.c`, the function **sHandleReturn** includes the functionality to return the fulfillment record specified using its fulfillment ID to a license server.

Using the Manual Response Generator

The manual response generator should be used for testing only. It provides an illustration of a manual transaction using XML files. When using XML files in an activation transaction, the files can be transmitted by any suitable means, for example email. However the entire file must be transmitted unaltered.

The following sections describe how to use the activation API to implement transactions using the manual response generator:

- Activation From a Manual Response Generator
- Return of Fulfillment Record to Manual Response Generator
- Repair From a Manual Response Generator

Activation From a Manual Response Generator

The following description gives in outline the sequence of operations required to perform an activation from a manual response generator.

Task ***To perform an activation from a manual response generator***

1. Create a server activation object with **flxActSvrActivationCreate**.
2. Add activation request parameters to the activation object.

   Set the following mandatory request parameters:

   - **Entitlement ID**—Use **flxActASvrActivationEntitlementIdSet** to set the entitlement ID. This identifies the entitlement for which the license rights are being requested.
   - **License rights expiration date**—Use **flxActASvrActivationExpDateSet** to enter a specific expiration date for the license rights.
   - **License group counts**—Use **flxActASvrActivationCountSet** to set the required license group count. At least one license group count must be set when licenses of different types are required call **flxActSvrActivationCountSet** multiple times.
   - **Repair count**—Use **flxActASvrActivationCountSet** to request the number of repairs that are allowed for this fulfillment.

     Entitlement ID, expiration date, license group counts, and repair count are determined by external means.

   Optionally set the following request parameters:

   - **User name**—Use **flxActASvrActivationUsernameSet** to set the name of the user making the request.

- **Host name**—Use **flxActASvrActivationHostnameSet** to set the name of the system that initiated this request.
- Use **flxActASvrActivationIncludeEnterpriseData** to include user name and/or host name in the request.
- **Vendor data**—Use **flxActASvrActivationVendorDataSet** to specify key-value pairs that are added to the activation request. For an overview, see Using the Vendor Dictionary for Custom Data.

3. Create an activation request using **flxActSvrActivationReqSet**.
4. Output the activation request.
5. Clean up memory and delete temporary files. Delete the activation object using **flxActSvrActivationDelete**.
6. Copy the activation request and paste it into the manual response generator.
7. Use the manual response generator to create the required response: either a config response or an activation response. For instructions on operating the manual response generator, see Chapter 14, "The Manual Response Generator."
8. Create a server activation object with **flxActSvrActivationCreate**.
9. Input the response and process it using **flxActSvrActivationRespProcess**. If the response contains a Trusted Configuration, it is used to create the requested trusted section of trusted storage. If the response contains a fulfillment record, it is placed in trusted storage. The parameter `bIsTrustedConfig` records whether the response contained a Trusted Configuration or a fulfillment record.
10. Clean up memory and delete temporary files. Delete the activation object with **flxActSvrActivationDelete**.
11. If the activation response contained a Trusted Configuration, repeat steps 1 to 10.

    The activation request created in Step 3 contains details of the newly created trusted section. Then at step 7, an activation response that contains the fulfillment record requested can be generated because the required trusted section is available on the requesting system.

In the example server activation utility, `serverActUtil.c`, the function **sHandleActivation** includes the functionality to generate an activation request as an XML file (Steps 1 through 5). The function **sHandleManualProcessing** inputs the response from the manual response generator and processes it (Steps 8 through 10). Step 11 is not demonstrated in the example activation utility.

Multiple-activate and single-step activate are being supported by composite transactions and the manual response generator. For an example, see `<platform_dir>\publisher\xml\Version6\W7_AppEx1SingleStepActivate.xml`

Product upgrade is also supported by composite transactions. Product upgrade involves modifying the count of an existing fulfillment and creating a new fulfillment. Deduction record data can then be retained, unlike with single action transactions, when the entire fulfillment has to be returned first before getting the fulfillment for the upgrade product. For an example, see `<platform_dir>\publisher\xml\Version6\W7_AppEx2ProductUpgrade.xml`.

Global and local vendor dictionaries are supported in composite transactions. Local vendor dictionaries apply to the fulfillment records within the local create, modify, or repair action. Global vendor dictionaries apply to all fulfillment records in any create, modify, or repair action in that response. For an example, see `W7_AppVendorDictionary.xml`.

Return of Fulfillment Record to Manual Response Generator

The following description gives in outline the sequence of operations required to return a fulfillment record to a manual response generator.

Task ***To return a fulfillment record to a manual response generator***

1. Create a server return object with **flxActSvrReturnCreate**.
2. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
3. Examine fulfillment records and choose the one that is to be returned.
   a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.
   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to choose the one to be returned. The following elements in a fulfillment record can be examined:
      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - License group counts—**flxActCommonProdLicSpcCountGet**
      - Repair count—**flxActCommonProdLicSpcCountGet**
      - Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
      - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
      - Expiry date—**flxActCommonProdLicSpcExpDateGet**
      - Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
      - Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
      - Fulfillment type—**flxActCommonProdLicSpcFulfillmentTypeGet**

   Note • *Return-all is supported by composite transactions and the manual response generator.*

4. Set the return request parameters by adding the fulfillment record that is to be returned to the return object with **flxActSvrReturnProdLicSpcSet**. Optionally add vendor data to the return object: use **flxActSvrReturnVendorDataSet** to specify key value pairs that are added to the return request. For an overview, see Using the Vendor Dictionary for Custom Data
5. Create a return request using **flxActSvrReturnReqSet**. The fulfillment record is marked as disabled: it cannot be used.
6. Output the return request.
7. Clean up memory and delete temporary files. Delete return object using **flxActSvrReturnDelete**.
8. Copy the return request and paste it into the manual response generator.
9. Use the manual response generator to create the required response: either a config response or a return response. For instructions on operating the manual response generator, see Chapter 14, “The Manual Response Generator.”
10. Create a return object using **flxActSvrReturnCreate**.

11. Input the response and process it using **flxActSvrReturnRespProcess**. If the response contained a Trusted Configuration, it is used to do one of the following:

    a. Update the settings for an existing trusted section. This is the more probable scenario for a Trusted Configuration in a return response.

    b. Create a trusted section.

    If the return response reported success, then the fulfillment record is deleted from the server-side trusted storage. If an error response is received, then the fulfillment record status is changed to not disabled.

    The parameter `bIsTrustedConfig` records whether the response contained a Trusted Configuration or the result of the return request.

12. Clean up memory and delete temporary files. Delete the return object with **flxActSvrReturnDelete**.

13. If the return response contained a Trusted Configuration, repeat steps 1 to 12.

14. If the returned fulfillment record contains concurrent or hybrid licenses, update the counts in memory.

    Use `lmreread.exe` as described in *Programming Reference for License File–Based Licensing*. This updates the license group counts held in memory. For more information, see Chapter 2, "Transactions to a License Server," and the section FulfillmentDictionary Data in Server-side Trusted Storage in Chapter A, "XML Schemas."

In the example server activation utility, `serverActUtil.c`, the function **sHandleReturn** includes the functionality to generate a return request as an XML file (Steps 1 through 5). The function **sHandleManualProcessing** inputs the response from the manual response generator and processes it (Steps 10 through 13). Step 14 is not demonstrated in the example activation utility.

Repair From a Manual Response Generator

The need for a repair of server-side trusted storage is indicated by a message in the debug log. When the vendor daemon starts up, it reads hybrid and concurrent license counts from fulfillment records in trusted storage. Any untrusted fulfillment records that contain hybrid or concurrent license counts are not used and an error is logged to the debug log. Fulfillment records that contain activatable licenses and are untrusted are logged to the debug log when they are accessed, that is, when the vendor daemon is processing an activation, return or repair request.

The following description gives a typical sequence of operations required to repair a single fulfillment record.

Task ***To repair a single fulfillment record***

1. Create a repair object with **flxActSvrRepairCreate**.

2. Read fulfillment records from trusted storage.

    a. Call **flxActCommonLicSpcCreate** to create a license specification container.

    b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.

3. Examine fulfillment records and find the one that requires repair.

    a. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specifications it contains. One product license specification objects corresponds to one fulfillment record.

   b. Step through the fulfillment records with **flxActCommonLicSpcGet** and examine each one in turn to find the one that requires repair. Use **flxActCommonProdLicSpcTrustFlagsGet** to determine whether a fulfillment record requires repair. You can also use other **flxActCommonProdLicSpc*Get** functions to identify the specific fulfillment record that requires repair.

4. Set the repair request parameters by adding the fulfillment record that is to be repaired to the repair object with **flxActSvrRepairProdLicSpcSet**. Optionally add vendor data to the repair object: use **flxActSvrRepairVendorDataSet** to specify key value pairs that are added to the activation request. For an overview, see Using the Vendor Dictionary for Custom Data.
5. Create a repair request using **flxActSvrRepairReqSet**.
6. Output the repair request.
7. Clean up memory and delete temporary files. Delete repair object using **flxActSvrRepairDelete**.
8. Copy the repair request and paste it into the manual response generator.
9. Use the manual response generator to create the required response: either a config response or a repair response. For instructions on operating the manual response generator, see Chapter 14, "The Manual Response Generator."
10. Create a repair object using **flxActSvrRepairCreate**.
11. Input the response and process it using **flxActSvrRepairRespProcess**. If the response contained a Trusted Configuration, it is used to do one of the following:
    - Update the settings for an existing trusted section. This is the more probable scenario for a Trusted Configuration in a repair response.
    - Create a trusted section.

    If the response is a repair response, then the fulfillment record is repaired. If the response reports an error, then the fulfillment record is not repaired and **flxActSvrRepairRespProcess** returns non success. An error response is returned when the repair is not allowed and also to report an error in the transaction.

    The parameter `bIsTrustedConfig` records whether the response contained a Trusted Configuration or the result of the repair request.
12. Clean up memory and delete temporary files. Delete the repair object with **flxActSvrRepairDelete**.
13. If the repair response contained a Trusted Configuration, repeat steps 1 to 12.

In the example server activation utility, `serverActUtil.c`, the function **sHandleRepair** includes the functionality to generate a repair request as an XML file (Steps 1 through 5). The function **sHandleManualProcessing** inputs the response from the manual response generator and processes it (Steps 10 through 12). Step 13 is not demonstrated in the example activation utility.

Extracting Information from Trusted Storage

The following sections describe how to use the activation API to extract information from the fulfillment records held in trusted storage:

- Display Current License Rights in Trusted Storage
- Display License Rights Activated by This Server
- Display Complete Contents of Trusted Storage

- Enable Fulfillment Record when No Response to Return Request
- Display License Rights Held in Remote Server-Side Trusted Storage

Display Current License Rights in Trusted Storage

The following description gives a possible sequence of operations using the Activation API to display license rights available in the server-side trusted storage. These license rights were supplied to the license server using an activation transaction. They are available to FlexEnabled clients or other license servers either as activatable licenses or concurrent licenses.

Task ***To display license rights available in the server-side trusted storage***

1. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
2. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specification objects it contains. One product license specification object corresponds to one fulfillment record.
3. Iterate through all product license specifications in turn. For each specification:
   a. Get each product license specification object at the specified index using **flxActCommonLicSpcGet**.
   b. Extract required information from the fulfillment record. The following information in a fulfillment record is available. Use the appropriate method from the Activation API:
      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - License group counts—**flxActCommonProdLicSpcCountGet**
      - Repair count—**flxActCommonProdLicSpcCountGet**
      - Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
      - Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
      - Expiry date—**flxActCommonProdLicSpcExpDateGet**
      - Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
      - Trust flags—**flxActCommonProdLicSpcTrustFlagsGet**
      - Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
      - Disabled because return in progress—**flxActCommonProdLicSpcIsDisabled**
      - Server chain—**flxActCommonProdLicSpcServerChainGet**
      - Fulfillment type—**flxActCommonProdLicSpcFulfillmentTypeGet**
   c. Format the extracted information and output it.

4. Release the license specification object resources and delete the license specification object with **flxActCommonLicSpcDelete**.

Note • *The setting of the mode parameter to* `FLX_ACT_SVR` *with* **flxActCommonHandleOpen** *defines that this process outputs details of licenses held in the server's trusted storage (server-side trusted storage).*

The function **sDumpTS**, in the example server activation utility, and `serverActUtil.c` display the license rights held in server-side trusted storage.

Display License Rights Activated by This Server

The following description gives a possible sequence of operations using the Activation API to display license rights issued in response to an activation request received by this server and still active on the remote system. These license rights were supplied to the license server using activation and were subsequently issued to either FlexEnabled clients or license servers via an activation transaction.

In these cases the license server records details of the issued license rights so that when they are no longer valid on the remote systems any remaining valid license rights automatically revert back to the issuing license server. License rights revert when the expiry date set in the remote fulfillment record is earlier than the expiry date for the fulfillment record on the issuing license server.

This information is recorded in *deduction records*. See Deduction Records for more details. Deduction records that refer to unexpired license rights are referred to as *valid* deduction records. Note that, as validity is a function of the expiry date, it is not a permanent quality of a deduction record: it will change over time.

License rights issued using activation can also be returned to the license server that issued them using a *return* transaction. This results in the deletion of the deduction record that recorded their issue.

Task ***To display license rights activated by this server***

1. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
2. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specification objects it contains. One product license specification object corresponds to one fulfillment record.
3. Iterate through all product license specifications in turn. For each specification:
   a. Get each product license specification object at the specified index using **flxActCommonLicSpcGet**.
   b. Get the number of deduction records associated with the fulfillment record using **flxActCommonProdLicSpcNumberDedSpcGet**.
   c. Iterate through each deduction record using **flxActCommonProdLicSpcDedSpcGet** and determine whether it references unexpired license rights on a remote system:

- **flxActCommonDedSpcTypeGet**—only deduction records of type *activation* or *transfer* contain details of license rights on remote systems
- **flxActCommonDedSpcExpDateGet**—examine expiry date of the remote fulfillment record to determine if the license rights have expired

d. Extract any additional information from each valid deduction record using the appropriate functions as follows:

- **flxActCommonDedSpcDestinationFulfillmentIdGet**—fulfillment identity of fulfillment record issued by this license server
- **flxActCommonDedSpcDestinationSystemNameGet**—the name of the system that the fulfillment record was issued to
- **flxActCommonDedSpcCountGet**—the value of the specified count in the fulfillment record issued by this license server

e. Format the extracted information and output it.

4. Release the license specification object resources and delete the license specification object with **flxActCommonLicSpcDelete**.

Note • *The setting of the mode parameter to* `FLX_ACT_SVR` *with* **flxActCommonHandleOpen** *defines that this process outputs details of licenses held in the server's trusted storage (server-side trusted storage).*

The function **sOutputDedSpc**, in the example server activation utility, and `serverActUtil.c` display the contents of a single deduction record. Using the command switches `-view -long` outputs both the contents of fulfillment records and deduction records in server-side trusted storage.

Deduction Records

Deduction records are created on a license server in the following situations:

- An activation request is processed and a fulfillment record is created and sent in response to the request.
- A repair request is processed for a fulfillment record.
- A license is borrowed using the Activation Borrow capabilities.
- An activatable overdraft license is not returned before its expiry date.

The following information is recorded in the deduction record:

- **Transaction Type**—the type of transaction that resulted in this deduction record's creation. The following types are used:
  - Activation—activation request from a FlexEnabled client.
  - Transfer—activation request from a license server.
  - Repair—repair request.
  - Unknown—implies that the deduction record is corrupt.

- **Destination fulfillment ID**—the fulfillment ID of the remote fulfillment record created by the license server as the result of the original activation request. This remote fulfillment record is either on a FlexEnabled client (Transaction type=activation) or on a license server (Transaction type=transfer).
- **Destination system name**—system name of the FlexEnabled client or license server that the fulfillment record was sent to.
- **Counts**—the number of licenses of various types available in the remote fulfillment record. For transfers, the deduction record may contain the repair count. The repair count for a transfer deduction record records the number of repairs that the remote license server can provide for activatable licenses included in the remote fulfillment record. For information on license counts, see Chapter 2, "Transactions to a License Server," and the section FulfillmentDictionary Data in Server-side Trusted Storage in Chapter A, "XML Schemas." Counts are not available in repair deduction records.
- **Expiry date**—the expiry date for the remote fulfillment record. The expiry date is not available in repair deduction records.

Display Complete Contents of Trusted Storage

Server-side trusted storage contains two types of records:

- Fulfillment records
- Deduction records

Deduction records are associated with a fulfillment record. For a description, see Deduction Records.

The following description gives a possible sequence of operations using the Activation API to display the contents of trusted storage.

Task ***To display the contents of trusted storage***

1. Read fulfillment records from trusted storage.
   a. Call **flxActCommonLicSpcCreate** to create a license specification container.
   b. Call **flxActCommonLicSpcPopulateFromTS** to read the fulfillment records from trusted storage and add them to the license specification container as product license specification objects.
2. Query the license specification container with **flxActCommonLicSpcGetNumberProducts** to find out how many product license specification objects it contains. One product license specification object corresponds to one fulfillment record.
3. Iterate through all product license specifications in turn. For each specification:
   a. Get each product license specification object at the specified index using **flxActCommonLicSpcGet**.
   b. Extract required information from the fulfillment record in the product license specification. The following information in a fulfillment record is available. Use the appropriate method from the Activation API:
      - Product ID—**flxActCommonProdLicSpcProductIdGet**
      - License group counts—**flxActCommonProdLicSpcCountGet**
      - Repair count—**flxActCommonProdLicSpcCountGet**

- Entitlement ID—**flxActCommonProdLicSpcEntitlementIdGet**
- Fulfillment ID—**flxActCommonProdLicSpcFulfillmentIdGet**
- Expiry date—**flxActCommonProdLicSpcExpDateGet**
- Suite ID—**flxActCommonProdLicSpcSuiteIdGet**
- Trust flags—**flxActCommonProdLicSpcTrustFlagsGet**
- Feature lines—**flxActCommonProdLicSpcFeatureLineGet**
- Disabled because return in progress—**flxActCommonProdLicSpcIsDisabled**
- Server chain—**flxActCommonProdLicSpcActServerChainGet**
- Fulfillment type—**flxActCommonProdLicSpcFulfillmentTypeGet**
- Custom Data in the Fulfilment Re cord–

  flxActCommonProdLicSpcVendorDataGetByKey

  flxActCommonProdLicSpcVendorDataGetCount

  flxActCommonProdLicSpcVendorDataGetByIndex

c. Get the number of deduction records associated with the fulfillment record using **flxActCommonProdLicSpcNumberDedSpcGet**.

d. Iterate through each deduction record using**flxActCommonProdLicSpcDedSpcGet** and determine whether it references unexpired license rights on a remote system:

- **flxActCommonDedSpcTypeGet**—only deduction records of type *activation* or *transfer* contain details of license rights on remote systems
- **flxActCommonDedSpcExpDateGet**—examine expiry date of the remote fulfillment record to determine if the license rights have expired

e. Extract any additional information from each valid deduction record using the appropriate functions as follows:

- **flxActCommonDedSpcDestinationFulfillmentIdGet**—fulfillment identity of fulfillment record issued by this license server
- **flxActCommonDedSpcDestinationSystemNameGet**—the name of the system that the fulfillment record was issued to
- **flxActCommonDedSpcCountGet**—the value of the specified count in the fulfillment record issued by this license server

f. Format the extracted information and output it.

4. Release the license specification object resources and delete the license specification object with **flxActCommonLicSpcDelete**.

Note • *The setting of the mode parameter to* `FLX_ACT_SVR` *with* **flxActCommonHandleOpen** *defines that this process outputs details of licenses held in the server's trusted storage (server-side trusted storage).*

Using the example server activation utility, `serverActUtil.c`, with the command switches `-view -long` outputs both the contents of fulfillment records and deduction records in server-side trusted storage.

Enable Fulfillment Record when No Response to Return Request

When a return request is sent, the fulfillment record that is being returned is disabled. The client then waits until a response is received from the activation server. In the case where no response or an invalid response is received, the fulfillment record remains disabled and the license rights it contains are unavailable. The **flxActSvrReturnCance**l function allows you to enable the fulfillment record and make the license rights available.

For example code that demonstrates the use of this function, see the example activation utility, `serveractutil`. The `serveractutil` command line syntax to demonstrate the use of **flxActSvrReturnCancel** is the following:

```
serveractutil -return <fulfillmentID> -cancel
```

Display License Rights Held in Remote Server-Side Trusted Storage

To display the license rights read from remote server-side trusted storage, create a function that uses steps similar to those for the function **sDumpServerTS** in the example application activation utility, `appActUtil.c`. (The exception is that the mode is set to `FLX_ACT_SVR`.) The function you create is called by the `-serverview` switch. For more information, see Display Server-Side License Rights Remotely On the Client in Chapter 10, “Building an Application Activation Utility for Single-Action Transactions.”

Important • *The license server whose trusted storage you want to view must be running.*

12

Building an Activation Utility for Composite Transactions

Activation utilities manage license rights in trusted storage. This chapter provides steps for creating and deploying an activation utility that uses composite transactions to handle activations between a publisher's activation server and an activation client (either a license server or a FlexEnabled client).

About Composite Transactions

A *composite transaction* can perform multiple activation actions in a single transaction. For example, one composite transaction can activate several fulfillment records or perform a combination of returns and activations. Best practice is to use composite transactions; single-action transactions have been deprecated (though they are still supported).

If your license model requires an activation utility (application, server, or both) that supports composite transactions, use the information in this chapter to create and provide a publisher-specific version of this utility. The following example application and server activation utilities for composite transactions are provided with the FlexNet Publisher Licensing Toolkit:

```
<platform_dir>/appcomptranutil.exe
<platform_dir>/servercomptranutil.exe
```

The corresponding source code can be found at this location:

```
<install_dir>/examples/activation/comptranutil/compTranUtil.c
```

Activation Utilities for Single-Action Transactions

This chapter focuses on building activation utilities that support composite transactions. For information about building activation utilities that support single-action transactions, see Chapter 10, "Building an Application Activation Utility for Single-Action Transactions," and Chapter 11, "Building a Server Activation Utility for Single-Action Transactions."

Path and Directory Conventions Used in This Chapter

References to directories and paths in this chapter use the following conventions:

- The term `<install_dir>` is used to represent the path and directory in which your FlexNet Publisher Licensing Toolkit is installed.
- The term `<platform_dir>` is used to refer to the path and directory in which the platform-specific components of the FlexNet Publisher Licensing Toolkit are located. For example, for a Windows 32-bit system, `<platform_dir>` represents `<install_dir>\i86_n3`. For a Sun Microsystems 32-bit Sparc running Solaris 10.*x*, `<platform_dir>` represents `<install_dir>/sun4_u10`.
- File paths for locations common to both UNIX and Windows platforms are constructed using forward slashes.

About Creating Activation Utilities for Composite Transactions

This section provides an overview of the following:

- API Used to Create an Activation Utility for composite transactions
- Role of the Activation Utility in the Composite-Transaction Process

API Used to Create an Activation Utility

In general, activation utilities use the FNP Activation Library API to create requests and process responses. A set of functions in the header file `flxActTransaction.h`, beginning with **flxActTran***, are used specifically for the creation of composite transaction utilities. For a detailed description of each of these functions, see the *C/C++ Function Reference*.

The composite transaction API is common to both application and server activation utilities (the `isserver` attribute in the `preptool` XML is used to select the required mode). Developers can write a single program and prepare it separately to create either type of utility.

Role of the Activation Utility in the Composite-Transaction Process

The activation utility must handle these steps in the composite-transaction process:

1. Create a request, or load a previously created request from trusted storage.
2. Send the request to the activation server, obtain a response, and then process the response, either online or manually.
3. Determine whether error reporting or recovery is needed.
4. Perform any post-processing tasks (for example, reading and acting on vendor dictionary entries).
5. Delete the request and corresponding response from trusted storage.

In addition to this chapter, refer to these materials for information about creating activation utilities for composite transactions:

- For an overview of the objects, API functions, and other data that the activation utility needs to perform these steps, review the API header file `flxActTransaction.h`, found in the `<platform_dir>/machind/activation/include` directory.
- To view sample code that implements the API functions and processes covered in this chapter, review the C source files located in the `<install_dir>/examples/activation/comptranutil` directory.
- For a detailed description of each API function used in the activation utility, see the *C/C++ Function Reference*.

Creating an Activation Utility

Use the common and composite-transaction API functions defined in the Activation API when creating an activation utility that supports composite transactions. The same set of calls are used to create the either the server or the application activation utility. The basic structure of the utility is described below.

Task ***To develop the sequence of API calls used to handle a transaction***

1. Install the FlexNet Licensing Service (if it is required by the platform and has not been previously installed on the machine).

   Note • *The FlexNet Licensing Service service can be installed by the FlexEnabled application installer or by a separate utility that runs during the application initialization. Publishers must develop this capability within their application. See Chapter 6, "Initializing Trusted Storage," for more information.*

2. Call **flxActCommonLibraryInit** to initialize the library. This must occur before any other call to the Activation API because it loads the activation library.

   Note • *In trusted storage–based applications,* **flxActCommonLibraryInit** *is the same as the Flexible API function* **lc_flexinit** *(described in Chapter 5, Adding Function Calls to a FlexEnabled Application). If you are using both Activation API and the Flexible API in this utility, ensure that you call either* **flxActCommonLibraryInit** *or* **lc_flexinit,** *but not both.*

3. Open an Activation API handle by calling **flxActCommonHandleOpen** with the mode parameter set to `FLX_ACT_PREP`.
4. Open a transaction handle by calling **flxActTransactionCreate**.
5. Design logic to create a request and process the response in the transaction. See Creating a Request and Processing the Response.
6. Clean up the transaction by calling **flxActTransactionDestroy**. This function removes the subordinate objects created in the previous steps.
7. Close the Activation API handle by calling **flxActCommonHandleClose**, which releases all resources associated with it.
8. Release the Activation API by calling **flxActCommonLibraryCleanup** to shutdown the shared component. No other Activation API routines can be called after this point.

Important • *Note the following about* **flxActCommonLibraryCleanup***:*

- *This function is also performed by the Flexible API function* **lc_flexinit_cleanup** *(described in Chapter 5, Adding Function Calls to a FlexEnabled Application). If your utility previously called* **lc_flex_init** *(see Step 2), use* **lc_flexnit_cleanup** *here, not* **flxActCommonLibraryCleanup***.*
- *On Windows platforms, do not call* **flxActCommonLibraryCleanup** *or* **flxActCommonLibraryInit** *in* `DllMain`*. These APIs result in calls to the Microsoft Win32 FreeLibrary and LoadLibrary APIs. Calls to these Win32 APIs in* `DllMain` *are widely known to be associated with issues.*

Creating a Request and Processing the Response

Use the following function sequence to have the activation utility create a request and process the response.

Task ***To specify the function calls for creating a request and processing the response***

1. Use **flxActTransactionCreateRequest** to create the request.

   Note • *Optionally, use the reference string supplied with the* `pReference` *parameter in* **flxActTransactionCreateRequest** *function to initialize the* `FLX_ACT_TRAN_REQ_REFERENCE` *attribute. This attribute can be used later to identify the request when enumerating stored requests.*

2. Specify the request attributes using **flxActTranRequestSetAttribute**.
3. (Optional) Specify vendor data for the request using **flxActTranRequestGetVendorDictionary** and **flxActTranDictionarySetByKey**.
4. Include one or more actions for this transaction to perform. See Adding Actions to the Transaction.
5. Use **flxActTranRequestMatchStored** to determine whether this request matches a request already stored in trusted storage; and, if no match exists, use **flxActTranRequestSave** to save the request to trusted storage. For more information about how these functions are used, see the comments in the file `<install_dir>/examples/activation/comptranutil/compTranUtilCreateRequest.C`.
6. Specify logic to complete the transaction. See Completing the Transaction.
7. Specify logic to query the response content if needed. See Querying the Response Content.
8. Delete the request (and associated response) using **flxActTransactionDeleteStoredRequest**.

Adding Actions to the Transaction

Use the following sequence to define an activation action. This sequence is repeated for each action specified in the request.

Task ***To define logic for each possible action in a request***

1. Use **flxActTranRequestAddAction** to add the action, specifying the action type:
   - `FLX_ACT_TRAN_REQ_ACTION_ACTIVATE`
   - `FLX_ACT_TRAN_REQ_ACTION_RETURN`
   - `FLX_ACT_TRAN_REQ_ACTION_REPAIR`
2. (Optional) Specify vendor data for the action:
   a. Call **flxActTranReqActionGetVendorDictionary** to establish a vendor dictionary.
   b. For each dictionary entry you want to add, use **flxActTranDictionarySetByKey** to specify the dictionary key and its value.
3. To define attributes specific to the action, do the following:
   a. Call **flxActTranReqActionSetAttribute**.
   b. Define attributes specific to the action:

Action Type	Mandatory Attribute	Optional Attributes
Activate	`FLX_ACT_TRAN_REQ_ACT_RIGHTS_ID_VALUE`	• `FLX_ACT_TRAN_REQ_ACT_REASON` • `FLX_ACT_TRAN_REQ_ACT_EXPIRATION_VALUE` • `FLX_ACT_TRAN_REQ_ACT_COUNT` • Other attributes ***Note*** • *Although not required by FlexNet Publisher, one or more of these attributes might be required by the publisher's activation server (such as FlexNet Operations).*
Return	`FLX_ACT_TRAN_REQ_ACT_FULFILLMENT_ID`	• `FLX_ACT_TRAN_REQ_ACT_REASON` • `FLX_ACT_TRAN_REQ_ACT_COUNT` • Other `FLX_ACT_TRAN_REQ_ACT_*` attributes ***Note*** • *Although not required by FlexNet Publisher, one or more of these attributes might be required by the publisher's activation server (such as FlexNet Operations).*
Repair	`FLX_ACT_TRAN_REQ_ACT_FULFILLMENT_ID`	None

Completing the Transaction

Include the functions needed first to send the request to the publisher's activation server and then to process the response sent by the activation server. This process can be online or offline.

Completing the Transaction Online

Completing the transaction online involves an online connection to the activation server (such as FlexNet Operations or a custom activation server).

Task ***To set up logic to complete an online transaction***

1. Set the server communications attribute using **flxActTransactionSetCommAttribute(FLX_ACT_TRAN_SVR_URI)**.
2. (Optional) Specify any of these additional communication attributes. They define how to connect to the activation server, depending on the network and server architecture (for example, whether a proxy is in use):
   - `FLX_ACT_TRAN_SVR_PROXY_HOST`
   - `FLX_ACT_TRAN_SVR_PROXY_PORT`
   - `FLX_ACT_TRAN_SVR_PROXY_USER`
   - `FLX_ACT_TRAN_SVR_PROXY_PASSWORD`
   - `FLX_ACT_TRAN_SVR_SSL_CACERT`
   - `FLX_ACT_TRAN_SVR_SSL_CAPATH`
   - `FLX_ACT_TRAN_SVR_TIMEOUT`
   - `FLX_ACT_TRAN_SVR_POLL_INTERVAL`

   Note • *`FLX_ACT_TRAN_SVR_SSL_CACERT` and `FLX_ACT_TRAN_SVR_SSL_CAPATH` are ignored on Windows platforms.*
3. Call **flxActTransactionSend**. This function sends the request to the activation server and then obtains and processes the response.

For more information about how these functions are used in this process, see information about the function **flxCtuDoCommandTransaction** in the file `<install_dir>/examples/activation/comptranutil/compTranUtilSend.c`

Completing the Transaction Offline

Completing the transaction offline involves manually transferring the request and response between the activation server and the license server or FlexEnabled client.

Task ***To set up logic to complete an offline transaction***

1. Call **flxActTranRequestGetXML** to return the request that needs to be sent to the activation server.
2. Call **flxActTransactionProcessResponse** to process the response from the activation server.

For more information about how these functions are used in this process, see the information about the function **flxCtuDoCommandProcess** in the file `<install_dir>/examples/activation/comptranutil/compTranUtilResponse.c`.

Querying the Response Content

You can set up the utility to query the response. In many cases, such a query is not necessary because processing the response automatically updates trusted storage with the new rights. However, this action is useful when you need to obtain the following information:

- Vendor data from the response
- Details of actions that were performed
- Error details if response-processing for an action fails

You can set up the activation utility to query the following:

- Obtain Vendor Data at the Response Level
- List Actions in the Response
- Obtain Vendor Data for a Specific Action
- Obtain Attribute Value for a Specific Action
- Obtain Results of Performing a Specified Action

For more information about how the functions in this process are used, see information about the function **flxCtuResponseOnProcessSuccess** in the file `<install_dir>/examples/activation/comptranutil/compTranUtilResponse.c`.

Obtain Vendor Data at the Response Level

Use this sequence of function calls to obtain data from the vendor dictionary defined at the response level.

Task ***To obtain vendor information from the vendor dictionary defined at the response level***

1. Call **flxActTranResponseGetVendorDictionary**.
2. Obtain dictionary entries:
   - To list all entries in the vendor dictionary, call **flxActTranDictionaryGetCount** to get the entry count in the dictionary; then call **flxActTranDictionaryGetByIndex** repeatedly for each index entry 0 to *count*-1.
   - To list a specific dictionary entry by its key, call **flxActTranDictionaryGetByKey**.

List Actions in the Response

Use these functions to return one or more actions defined in the response.

Task ***To list the actions defined in the response***

1. Call **flxActTranResponseGetActionCount**.
2. Call **flxActTranDictionaryGetByIndex** repeatedly for each index position 0 (first) to *count*-1 (last) to obtain its corresponding entry.

Task ***To obtain the type of a specific action whose handle is specified in the call***

Call **flxActTranRspActionGetType**.

Obtain Vendor Data for a Specific Action

Use this sequence of function calls to return data from the vendor dictionary defined for the specified action.

Task ***To obtain vendor data for a specific action***

1. Call **flxActTranRspActionGetVendorDictionary** to obtain the vendor dictionary for a specific action.
2. Obtain the dictionary entries:
   - To list all entries in the vendor dictionary, call **flxActTranDictionaryGetCount** to get the entry count in the dictionary; then call **flxActTranDictionaryGetByIndex** repeatedly for each index entry 0 to *count*-1.
   - To list a specific dictionary entry by its key, call **flxActTranDictionaryGetByKey**.

Obtain Attribute Value for a Specific Action

Use this function to return the value of a specific attribute for the specified action.

Task ***To obtain the value of a specific attribute for an action***

Call **flxActTranRspActionGetAttribute**.

Obtain Results of Performing a Specified Action

Use this function to return the result of a specified action performed during response-processing. The result shows that the action succeeded or provides the reason why it failed.

Task ***To obtain the result of a specific action***

Call **flxActTranRspActionGetResult**.

Re-Enable Returned Fulfillment Records

When a return request is sent, the fulfillment record that is being returned is disabled. When the activation utility receives and processes the response, the fulfillment record is deleted if the return was successful. However, if the return was denied, the fulfillment record is automatically re-enabled.

In the case where no response or an invalid response is received, the fulfillment record remains disabled, and the license rights it contains are unavailable. To re-enable the fulfillment record, the request must be deleted from trusted storage. Use the following procedure to create a facility that deletes the request.

Task ***To create a facility that deletes a request for the purpose of re-enabling a fulfillment***

1. Call the functions **flxActTransactionGetStoredRequestCount** and **flxActTransactionGetStoredRequest** to enumerate the request.
2. Call **flxActTransactionDeleteStoredRequest** to delete the request that contains the return action for the fulfillment.

For more information about this process, see information about the function **flxCtuDoCommandCancelRequest** in the file `<install_dir>/examples/activation/comptranutil/compTranUtilManageRequests.c`.

To see this facility work, use one of the sample builds of the `compTranUtil` program to cancel individual or all stored requests. For example, enter the following:

```
appCompTranUtil -cancelAll
```

Important • *This facility might pose a security risk. For example, a fraudulent user could request a return (which is granted by the server) but not process the response. The user can then run this facility, thereby obtaining an extra license. If this risk is unacceptable, do not include this facility in the activation utility. Instead, have problems related to re-enabling fulfillment records handled through your support channels.*

Managing Stored Requests

Use the following functions to create a facility for removing unused requests saved in trusted storage. Removing them is not mandatory, but still good practice.

Note • *See the previous section Re-Enable Returned Fulfillment Records for a related security risk.*

Task ***To create a facility to remove unused stored requests from trusted storage***

1. Call the functions **flxActTransactionGetStoredRequestCount** and **flxActTransactionGetStoredRequest** to enumerate the request.
2. Call **flxActTransactionDeleteStoredRequest** to delete the stored request from trusted storage.

Building and Deploying Your Activation Utility

Task ***To build and deploy the server or application activation utility***

1. Compile and link the activation utility.

   A sample makefile is included in the toolkit (`.../<platform_dir>/makefile.act`). Depending on whether you are building an activation utility for your FlexEnabled client or the license server, use the build steps for either `appcomptranutil.exe` or `servercomptranutil.exe` as an example. This section of the makefile includes information about compiling, linking, and the using the `preptool` (see step 2).

2. For standard toolkits, use `preptool` to prepare the server activation utility and to generate publisher- and application-specific library for use by the server activation utility. See Chapter 13, "Working with the Software Publisher Utilities," especially the preptool section.

   Note • *The preparation utility provided with the toolkit is called* `preptool`*. However, if you create a publisher version of* `preptool`*, it might have a different name.*

   For `notr` toolkits, use the activation library component, `demo_server_libFNP_notr.*`, which was created when you built the toolkit.

3. Set up your product installer to install the utility and associated components in the same directory. The following components are required:

   - The prepped version of the server activation utility
   - The activation library created in step 2
   - For Windows, the FlexNet Licensing Service installer component (`FNP_Act_Installer.dll`)

     Note • *Ensure that the correct version of* `FNP_Act_Installer.dll` *is added to the FlexEnabled application. For more details, see the chapter "Distributing the FlexEnabled Application for Trusted Storage–Based Licensing" in FlexNet Publisher's Development Environment Guide.*

   - For transactions with a FlexNet Operations server, the `FnpCommsSoap` module

   You can design your product installer to install the FlexNet Licensing Service as well:

   - On Windows, use the instructions in Chapter 6, "Initializing Trusted Storage," to have your product installer also install the FlexNet Licensing Service if it has not been installed by another application. This is the recommended method for installing the FlexNet Licensing Service.

Note • *If your product installer is InstallShield-based, you can find examples of installing the FlexNet Licensing Service in* `<platform_dir>\examples\serviceinstall`.

- For Mac OS X and UNIX platforms, have the product installer run the `install_fnp.sh` script to install the FlexNet Licensing Service.

Additional Information About Activation Utilities for Composite Transactions

The following sections provide additional background on how the activation utility supports composite transactions. Use this information to help you plan how to structure your activation utility:

- Managing Outstanding Requests
- Re-using Stored Requests
- About Composite Transaction Requests
- About Composite Transaction Responses
- Information Used to Identify the System Initiating the Request
- Error Reporting and Recovery

Managing Outstanding Requests

When a composite request is created and saved, it is written to trusted storage, and remains there until explicitly deleted. Normally, saving the request occurs once the response has been received and processed.

Table 12-1 • Request Status

Status	Description
Pending	The request is waiting for a response to be processed. Requests with status `Pending` should normally be retained until the response is received. If necessary, they can be re-sent to the activation server using **flxActTransactionSend** or **flxActTranRequestGetXML**. However, if no prospect of a response exists, delete it using **flxActTransactionDeleteStoredRequest**.
Untrusted	The request is waiting for a response to be processed, but a local repair has been performed on trusted storage since the time that the request was stored. When the response is processed, the status becomes `Recover`. The action taken is the same as for `Pending`.

Table 12-1 • Request Status

Status	Description
Complete	Normally, the activation utility deletes the request once a transaction completes. This status suggests that activation utility successfully processed the response but failed sometime before it could delete the request. Consequently, both the response and request are in trusted storage. Reload the request, and complete any post-processing (such as vendor-data processing) that might have been curtailed by the failure. Then delete the request. The response is automatically deleted with it.
Recover	The response has been processed, but errors occurred. Create a recovery request, and send it to the activation server. A recovery request is essentially a copy of the original request, with an additional recovery section indicating what went wrong during the response-processing phase.

Re-using Stored Requests

The following describes the process of re-using stored requests:

1. When a user generates a request, it is saved in trusted storage. For example, the user might issue the following command:

   ```
   appcomptranutil -new request1.xml -activate MyRights count=5
   ```

   The resulting request is stored in trusted storage; and, if the response is not received for some reason, the request remains in storage.

2. The user can regenerate the request by repeating the command:

   ```
   appcomptranutil -new request2.xml -activate MyRights count=5
   ```

   This command is probably performed to resend the request to the server (although it might be the second request for an additional 5 licenses). If the intention is to resend the request, it is important that the same request is used so that the server can implement its recovery policy. That is, the activation utility attempts to match the new request to an existing one in storage; and, if a match exists, the utility automatically re-uses the existing request. (A match occurs only if the request actions and attributes are identical in both requests.)

3. The user can force the creation of a new request by including a unique reference, which prevents any match. The following command shows an example of using a unique reference:

   ```
   appcomptranutil -new request2.xml -activate MyRights count=5 reference=Another5
   ```

About Composite Transaction Requests

A number of data items that are included in the request need to have their values defined by the activation server. (The schemas specify them only as strings.)

- Activation reason
- Return reason
- Rights Id type

- Reference type (for recovery, Rights Id type plus Fulfillment Id)

Identifying the Requesting Machine

A composite request contains information that identifies the requesting machine. The activation server uses this information to determine whether to grant the request, such as a repair or reinstall. For more details about these machine identifiers, see Information Used to Identify the System Initiating the Request.

Additionally, if a prepped Trusted Configuration is enabled, requests from new machines contain information about the new trusted section created in trusted storage by the activation utility at the time the request is generated. However, if a prepped Trusted Configuration is not enabled, the configuration actions used to create the initial trusted section still need to be performed; therefore, the request does not contain trusted-section information.

Composite Request Data Items

These data items are used in requests originating from physical and virtual environments.

Table 12-2 • Composite Request Data Items

Data Item	Description
Trusted storage serial number (TSSN)	A unique identifier generated each time a trusted-storage file is created. ***Note •*** *FlexNet Operations does not use this item currently.*
Generated	The date, time and time zone that the request was created.
Requester reference	An arbitrary string optionally supplied by the activation application when the request was created. Intended for identifying the request for re-use.

Additional Composite Request Data Items for Virtual Environments

These additional items are used in requests originating from a virtual machine. The items are found in the `<Virtualization>` section of the request and provide information about environment hosting the virtual machine:

Table 12-3 • Additional Composite Request Data Items for Virtual Environments

Data Item	Description
Family	The name of the virtualization-system provider: `MICROSOFT`, `VMWARE`, `XEN`, or `UNKNOWN`.
Name	The name of the virtualization-system hypervisor product, such as `HYPERV`, `XEN`, `VMWARE`, or `UNKNOWN`. If the hypervisor name is not available, the `Family` value is used in the request.

Table 12-3 • Additional Composite Request Data Items for Virtual Environments

Data Item	Description
VMID	The universally unique ID (UUID) of the virtual machine, as provided by the hypervisor, in a hashed format. FlexNet Publisher uses this value to bind the machine to trusted storage. This same value is also used as the UniqueMachineNumber 3 (UMN3) value for requester verification. If used, the VMID is the only binding element. If the VMID is not available, the element is omitted from the request. ***Note •*** *The VMID is based on the same (virtualized) hardware item as the VM_UUID hostid described in the Programming Reference for License File–Based Licensing.*

Composite Request Actions

A composite request can contain any combination of activate, return, or repair actions. It can also include existing fulfillments present in trusted storage, as well as information about trusted storage breaks.

When used in composite transactions, these actions have the following additional data items associated with them:

Table 12-4 • Activate Actions Additional Data Items

Data Item	Description
RightsId type	The type of RightsId that follows, as defined by the activation server.
RightsId	The ID allocated by the activation server to associate this activation with the appropriate database entries.
Reason	A string supplied by the activation application from a set defined by the server.
Count	Either `all` or a positive integer used to request a partial activation.
Expiration Type	The type of RightsId that follows, as defined by the server.
Expiration Value	The expiration date being request (units and format defined by the server through Expiration Type).
Server Counts	Counts maintained by the enterprise license server for the following license types: Activatable, Activatable Overdraft, Concurrent, Concurrent Overdraft, Hybrid, Hybrid Overdraft, Repair.
Fulfillment Id	The ID for the existing fulfillment associated with this action (the fulfillment to upgrade). More information about the fulfillment can be obtained from its entry in the `<ExistingFulfillments>` element.

Table 12-5 • Return Actions Additional Data Items

Data Item	Description
Fulfillment Id	The ID for the fulfillment that is being returned. More information about the fulfillment can be obtained from its entry in the `<ExistingFulfillments>` element.
Reason	A string supplied by the activation application from a set defined by the server.
Count	Either `all` or a positive integer used to request a partial return.

Table 12-6 • Repair Actions Additional Data Items

Data Item	Description
Fulfillment Id	ID for the fulfillment that needs repair. More information about the fulfillment can be obtained from its entry in the `<ExistingFulfillments>` element.

Table 12-7 • Existing Fulfillments Additional Data Items

Fulfillment Id	ID as set by the server (or ASR) when the record was created.
Fulfillment Type	ID as set by the server (or ASR) when the record was created.
isTrusted	Trusted status of the fulfillment record: ● `true` if the record is fully trusted ● `false` if the record is not trusted (values of the individual trust flags also listed)
isEnabled	Status of the fulfillment record: ● `true` if the record is enabled ● `false` if the record is disabled because it is the object of a return in a previous request (excluding this request)
Trusted Id	Trusted section containing the existing fulfillment.
Fulfillment record	(Optional) XML file containing the fulfillment record description.

About Composite Transaction Responses

Response actions are not implicitly linked to request actions in the sense that the activation server can respond with any actions, and is not bound by what has been requested. However, normally each request action is either granted or denied using one or more response actions.

For details on how composite transactions responses are processed, review the steps contained in the API header file, `flxActTransaction.h`. Also see API function descriptions in the *C/C++ Function Reference*.

Any combination and number of response actions can be used to grant request actions. However, in practice, such combination mostly applies to activations. Return and repair request actions usually result in a single delete or repair response action.

Note • *Response actions are automatically performed unless the activation utility finds it necessary to inform the user about the results of an action. In practice, the* `Transaction Successful` *message is enough. The example activation utility* `appcomptranutil` *reports on response actions as an aid to development.*

A composite transaction response can include any of the following actions (as listed in the response's `<ResponseActions>` element:

Table 12-8 • Composite Transaction Response Actions

Action Type	Description
Config	Creates a new trusted section or updates an existing section. ***Note*** • *The* `<OriginData>` *in a request identifies the trusted sections that exist on the requesting machine, including the revision details of the latest Trusted Configuration used.*
Create	Creates a new fulfillment record.
Modify	Modifies one or more dictionary entries of an existing fulfillment record (for example, the expiry or count of a fulfillment record)
Delete	Deletes a fulfillment record.
Repair	Repairs a fulfillment record.
Activate deny	Informs the activation utility to deny the activation.
Return deny	Informs the activation utility to deny the return and re-enables the fulfillment record being returned.
Repair deny	Informs the activation application to deny the repair.

Requester Verification

The requester verification data in the response is used by FlexNet Publisher when processing the response to check that the machine is the same one that generated the request. This information helps to protect against security exploits where trusted storage is copied after the request is generated. The activation server should specify the item(s) that it uses to identify the machine in transactions:

- UMN1 and UMN2 for non-virtual environments
- UNM3 (and, if available, UMN1, UMN2, and UMN5) for virtual environments

When generating the response, the activation server normally includes (in the `RequesterVerification` section) any UMN values sent in the response.

For details about the information used to verify the requesting machine, see Information Used to Identify the System Initiating the Request.

Information Used to Identify the System Initiating the Request

Both requests and responses in composite transactions contain the following information (when it is available) to validate the identity of the machine that initiated the request:

- Unique Machine Numbers (UMNs)
- Machine Identifier (MID)
- Platform Type Information

Unique Machine Numbers (UMNs)

Activation request messages, both XML and short codes, include Unique Machine Number (UMN) values by default. In a non-virtual environment, FlexNet Publisher attempts to retrieve the UMN1 and UMN2 values. In a virtual environment, it attempts to retrieve UMN3 and UMN5 as well. The activation server stores this information to identify the requesting machine when determining whether to grant repair and reinstallation requests. On the activation client side, these values are processed in the response to validate that the machine doing the processing is the same as the requesting machine. For more information about UMNs and how they are used, see the Flexera publication *Understanding the UMN in FlexNet Publisher*.

UNM Support by Platform

UMN values are retrieved from system hardware information (or, for virtual machines, from hypervisor-controlled information). The following table lists the types of UMNs retrieved, the platforms on which they are available, and the means used to obtain the UMN value.

Table 12-9 • UMN Support by Platform

Platform	UMN1	UMN2	UMN3	UMN5
Windows	Boot disk serial number ***Important*** • *Know this:* • *This value is often null on a virtual machine.* • *The FlexNet Licensing Service must be installed on the machine in order to obtain UMN1.*	Primary Ethernet MAC address ***Note*** • *This value is determined by ignoring any class of network interface that is not on a physical bus. Removable and virtual network interfaces are not considered, ensuring the reliability of UMN2 on Windows.*	The virtual machine image identifier (VMID)—a hash of the universally unique ID (UUID) that the hypervisor issues whenever a virtual machine is created or cloned ***Note*** • *The UUID is the same value as the* `VM_UUID` *hostid described in the Programming Reference for License File–Based Licensing.*	Windows along with Hyper-V is supported, a native ACPI (Gencounter) driver is required for extracting Generation ID.
Mac	MAC Unique system ID ***Important*** • *The FlexNet Licensing Service must be installed on the machine in order to obtain UMN1.* ***Note*** • *On newer systems, this ID is hardcoded on the motherboard; the ID cannot change. For older systems, this information is on the disk and can be overwritten by low-level formatting only.*	Primary Ethernet MAC address ***Important*** • *This value changes if the Ethernet adaptor is changed.* ***Note*** • *All Macs capable of running OS X have an onboard Ethernet adaptor.*	Not applicable	Not applicable

Table 12-9 • UMN Support by Platform

Platform	UMN1	UMN2	UMN3	UMN5
Linux	Composite of the serial numbers for all non-removable IDE devices ***Important*** • *Know this:* • *Root privilege is required to extract UMN1. In turn, this requires that the FlexNet Licensing Service be installed on the machine.* • *Due to strict uniqueness policies, FlexNet Publisher might return a null value rather than retrieve a non-unique value.* • *A UMN1 value is not available on all virtual machines.*	Primary Ethernet MAC address ***Important*** • *This value changes if the Ethernet adaptor card is changed.*	The virtual machine image identifier (VMID)—a hash of the universally unique ID (UUID) that the hypervisor issues whenever a virtual machine is created or cloned ***Important*** • *The FlexNet Licensing Service must be installed on the machine in order to obtain UMN3.* ***Note*** • *The UUID is the same value as the* `VM_UUID` *hostid described in the Programming Reference for License File–Based Licensing.*	Not applicable
AIX	Unique hardware serial number, available on all PCI-based AIX hardware	Primary Ethernet MAC address ***Important*** • *This value changes if the Ethernet adaptor card is changed.*	Not applicable	Not applicable
HP/UX	Unique hardware security key	Primary Ethernet MAC address ***Important*** • *This value changes if the Ethernet adaptor card is changed.*	Not applicable	Not applicable

Table 12-9 • UMN Support by Platform

Platform	UMN1	UMN2	UMN3	UMN5
Solaris	Serial number generated during manufacturing and written to the EEPROM ***Important •*** *If Solaris is running on an Intel-based system (either Sun hardware or generic PC-based hardware), the UMN1 value changes if the operating system is reinstalled. This does not occur on SPARC systems.*	Primary Ethernet MAC address	Not applicable	Not applicable

Important Information About UMN Usage

Keep the following additional information in mind about UMNs:

- To keep up with developments in technology and provide the best possible unique and robust identifiers for host machines, Flexera reserves the right to change the source of any UMN value between versions of FlexNet Publisher. Should the source of any UMN change, you can find a description of this change in the relevant *FlexNet Publisher Release Notes* version.
- On Linux systems, best practice is always install the FlexNet Licensing Service on the customer platform so that UMN1 is generated if it is available. UMN1 is important for successful life-cycle transactions, especially re-installations or upgrades, in which the original activation transaction involved an earlier version of FlexNet Publisher.
- Short codes support UMN2 and UMNE. UMNE is UMN1 for physical environments and UMN3 for virtual environments. To keep the codes short only the least significant 8 characters (64 bits) of UMN are sent by default but this can be changed using an ASR override. For more information about this, see ShortCodeData Element in the section B_ActivationSpecificationRecord.xsd in Appendix A, “XML Schemas.”
- FlexNet Publisher supports its virtualization policy for only those requests sent from FlexEnabled clients or FlexNet license servers that run on virtual Windows or Linux machines. Additionally, these machines must be created in VMware, Microsoft Hyper-V, or Xen environments. See the current *FlexNet Publisher Release Notes* for specific operating-system and virtual-hypervisor versions.
- FlexNet Publisher attempts to obtain UMN3 and UMN5 when it detects that the requesting machine is running in a virtual environment. If it is present UMN5 is the first-choice UMN for identifying the virtual machine making the request and then UMN3.
- In a virtual environment, UMN1 and UMN2 are generated using methods that are basically the same as those used in a non-virtual environment. However, in a virtual environment, these UMNs pertain to the operating system of the virtual machine, not to the operating system of the physical host machine. Because these values do not change if the virtual machine is rehosted or cloned, security issues can occur. Additionally, in a virtual environment, these UMNs are not always available.
- Composite transactions support UMN3 and UMN5; single-action transactions do not. Issues can arise if you attempt to mix composite and single-action transactions (although both are supported for life-cycle operations between an

activation server and a client). For example, if a composite transaction is used initially to activate licenses on a virtual machine, the UMN3, if obtained, becomes the primary UMN. If a single-action transaction is used later to return a license, the process fails because the return request contains no UMN3 to identify the requesting machine. Best practice is always to use composite transactions.

- Virtualization stack that supports extracting GenerationID: VMWare ESXi 5.1, VMWare Workstation 9 or 10, Hyper-V with Windows Server 2012 (or R2), Hyper-V with Windows 8.1, XenServer 6.2. Guests should include those that support GenerationID (Windows 8+, Windows Server 2012+, Windows7 on Hyper-V with integrated services installed)

Machine Identifier (MID)

The machine identifier (MID) is generated from a composite of all entities used to bind a trusted section in trusted storage. Consequently, the trusted section must exist before the MID is sent in requests. With the introduction of the prepped Trusted Configuration in FlexNet Publisher 11.10.1, the activation utility automatically creates the initial trusted section (if it does not already exist) before sending the request. Consequently, the MID, along with UMNs, is now sent in the initial activation request to be stored as a requesting-machine identifier on the activation server. Although the MID is less stable than a UMN (it can change when binding entities change), it is useful in most cases and always serves as a backup identifier if one or more of the UMNs are unavailable. The MID is not used for requester verification in the response.

Platform Type Information

By default, the platform type is sent in all XML requests, but not in short-code requests. The platform type identifies the operating system of the machine initiating the request. The following table describes the platform types sent:

Table 12-10 • Platform Types Sent in XML Requests (currently provided toolkits)

Platform Type Value	Description
i86_lsb	Linux Standard Base (LSB) on x86
i86_mac10	Mac OS X on x86
i86_n3	Microsoft Windows 32-bit on x86
ppc_u5	AIX 32-bit on PowerPC and RS/ 6000
ppc64_u5	AIX 64-bit on PowerPC and RS/ 6000
sun4_u10	Solaris 10 32-bit on SPARC
sun64_u10	Solaris 10 64-bit on SPARC
x64_lsb	Linux Standard Base (LSB) 64-bit on x64
x64_mac10	Mac OS X 64-bit platforms
x64_n6	Microsoft Windows 64-bit on x64
x64_sun10	Solaris 10 64-bit on x64

Table 12-10 • Platform Types Sent in XML Requests (currently provided toolkits)

Platform Type Value	Description
x86_sol10	Solaris 10 32-bit on x86

The following table lists the platform types that are no longer provided and are classed as obsolete:

Table 12-11 • Obsolete Platform Types

Platform Type Value	Description
amd64_re3	Red Hat 3 64-bit on x64
hp64_u11i	HP-UX 64-bit on PA-RISC 64-bit
hp700_u11i	HP-UX 32-bit on PA-RISC 32-bit
i86_f6	FreeBSD 32-bit
i86_re3	Red Hat 3 32-bit on x86
i86_se9	SUSE Linux Enterprise 9 32-bit on x86
it64_n	Microsoft Windows 64-bit on Intel Itanium
it64_re3	Red Hat Enterprise Linux 3 64-bit on Intel Itanium
it64_re4	Red Hat Enterprise Linux 4 64-bit on Intel Itanium
it64_re5	Red Hat Enterprise Linux 5 64-bit on Intel Itanium
ppc_lsb	Linux Standard Base (LSB) 32-bit on PowerPC
ppc_mac10	Mac OS X 32-bit on PowerPC
ppc_re3	Red Hat Enterprise Linux 3 32-bit on PowerPC
ppc_se9	SUSE Linux Enterprise 9 32-bit on PowerPC
ppc64_lsb	Linux Standard Base (LSB) 64-bit on PowerPC
ppc64_mac10	Mac OS X 64-bit on PowerPC
ppc64_re3	Red Hat Enterprise Linux 3 64-bit on PowerPC
ppc64_se9	SUSE Linux Enterprise 9 64-bit on PowerPC
sgi64_u6	Silicon Graphics 64-bit

Table 12-11 • Obsolete Platform Types

Platform Type Value	Description
sun4_u8	Solaris 32-bit on SPARC 32-bit
sun4_u9	Solaris 32-bit on SPARC 32-bit
sun64_u8	Solaris 64-bit on SPARC 64-bit
sun64_u9	Solaris 64-bit on SPARC 64-bit
x64_se9	SUSE Linux Enterprise 9 64-bit on x64
x86_sol8	Solaris 8 32-bit on x86
x86_sol9	Solaris 9 32-bit on x86

Error Reporting and Recovery

The following error messages can be returned for composite transactions responses.

Note • *Best practice for error reporting is to call* **flxActTransactionsLogError** *when an error occurs. This function writes the error to the trusted storage event log.*

Table 12-12 • Error Messaging

Message	Description
Response not received by the activation libraries	The response is not detected by the activation utility. In this situation, it cannot be determined whether the activation server actually received and processed the request. Recovery action for the activation utility is to re-send the request. Using using the request data (such as the sequence number, UMN, and RightsId), the activation server can detect whether it has previously processed the request.
Response invalid	One of the following events occurred: • The response was corrupted in transit. • The user supplied an incorrect response to the activation utility during a manual transaction. • The response was incorrectly constructed or signed (usually the result of a development issue with the activation server). The activation utility usually cannot determine which event caused the error. Therefore, the recovery action is the same as if the response was not received.

Table 12-12 • Error Messaging

Message	Description
Response actions cannot be performed	The response is valid, but the actions cannot be performed for one of the following reasons (identified by an error code `FLX_ACT_TRAN_ERR_xxx`). ● Trusted storage break has occurred since request generated. ● Config action has failed. ● Second or subsequent signature is invalid. ● Action attribute is invalid or missing. ● One or more response actions failed. ● Trusted storage load failed. ● Trusted storage save failed. ● Individual response action has generated an error (`FLX_ACT_TRAN_RSP_ACT_xxx`). ● Action was not attempted due to another type of error. ● Action is valid but not performed because of another type of error. ● Fulfillment record already exists (for a create). ● Fulfillment record does not exist (for a modify, delete, or repair). ● Application policy does not allow execution in a virtualized environment. ● Application policy does not allow execution in a physical (non-virtualized) environment. In this case, the activation utility should generate a recovery request, which is a clone of the original request with recovery information included. A new sequence number is generated, and the original sequence number is included. See the `Recovery` element of the `X_CompositeRequest` schema. The activation server can recover by backing out the previous transactions and sending a deny response to the requesting machine, or by sending a new response that addresses the issues reported.

Transactions Using Request and Response Codes

Normal activation transactions use XML documents that must be transmitted as either a data stream (without end-user intervention) or as an XML file (which may require end-user intervention). FlexNet Publisher Licensing Toolkit provides transactions using request and response codes on certain platforms when a network connection is not available or the XML documents are too large.

These transactions are referred to as *short-code transactions* as they use coded requests and responses. Traditionally this type of transaction is used for telephone transactions either with a customer service representative or using automated touch tone telephony. It is also used in high security situations where only non-electronic communication is allowed.

The limitations imposed on the length of the request and response codes require that detailed information, normally included in the activation transactions, is already present on the FlexEnabled client.

The following information is provided for short-code transactions in an ASR:

- Short-code alias—Identity used to identify transactions using this ASR.
- Short-code scheme ID—Default format of the request and response codes.
- Trusted Configuration—Data required to create, identify, and configure the trusted section in which the fulfillment record is to be stored.
- Fulfillment record
- Overrides—Optional entries that modify the default format of the request and response codes. Typical functionality provided by overrides includes the following (links are to Appendix A, XML Schemas):

  Adding Custom Data to an Activation Request—Add custom data (for example, data that to enables the end user to send a product code).

  Using the Dictionary Element to Set Alphanumeric Codes—Use alphanumeric codes to reduce the size of request and response codes.

  Specifying the Minimum Size of Response Codes—Set a standard size for request and response codes: additional characters are added to shorter codes.

  Specifying License Expiration in a Short-Code Activation— Allow your activation server to override the expiry date in the fulfillment record contained in the ASR.

  Note • *When using FlexNet Operations, overrides are automatically added from the options specified when creating the ASR. FlexNet Operations users should only edit the ASR manually when explicitly advised by Flexera (for example: for a workaround).*

For a product that requires both local and short-code transactions, create separate ASRs for local and short-code transactions. A short-code ASR (that is, an ASR with a `ShortCodeData` element), cannot be used for local activation.

The following sections describe sequences of operations required when using short-code transactions:

- Activation Using Request and Response Codes
- Return Using Request and Response Codes
- Repair Using Request and Response Codes
- Display Pending Request Code
- Cancel Outstanding Short-Code Transaction
- Adding Custom Data to a Short-Code Request

Activation Using Request and Response Codes

The following description outlines the sequence of operations required to perform an activation using request and response codes and a manual response generator.

Note • *Short code transaction are supported on both application and server trusted storage using the same API functions. The mode used to prepare the application determines which is used.*

Task ***To perform an activation using request and response codes***

1. Create short code object with **flxActShortCodeCreate** or **flxActShortCodeCreateFromBuffer**.
2. Generate an activation request code with **flxActShortCodeGetActivationRequest**.

   The ASR used to generate the activation request code is determined by external means. If it does not already exist, the required trusted section is created using the Trusted Configuration contained in the ASR.
3. Output the activation request code.
4. Clean up memory and delete temporary files. Delete the short code object using **flxActShortCodeDestroy**.
5. Copy the activation request code and paste it into the manual response generator.
6. Use the manual response generator to create the required response code. For instructions on operating the manual response generator, see Chapter 14, “The Manual Response Generator.”
7. Input the response code and process the response using **flxActShortCodeProcessResponse**.

   If the response code is valid and does not report an error, the fulfillment record from the ASR is committed to trusted storage.

Return Using Request and Response Codes

The following description gives in outline the sequence of operations required to return a fulfillment record to a manual response generator.

Task ***To return a fulfillment record to a manual response generator***

1. Create short code object with **flxActShortCodeCreate** or **flxActShortCodeCreateFromBuffer**.
2. Generate a return request code using **flxActShortCodeGetReturnRequest**.
3. Output the return request code.
4. Clean up memory and delete temporary files. Delete the short code object using **flxActShortCodeDestroy**.
5. Copy the return request and paste it into the manual response generator. Use the manual response generator to create the required response code. For instructions on operating the manual response generator, see Chapter 14, “The Manual Response Generator.”
6. Input the response and then process it using **flxActShortCodeProcessResponse**.

   If the return response reported success, then the fulfillment record is deleted from the application’s trusted storage.

In the example application activation utility, *compTranUtil* (binaries appcomptranutil and servercomptranutil) the file `compTranUtilShortCode.c` includes the functionality to perform short code transactions..

Repair Using Request and Response Codes

The need for a repair of client-side trusted storage is indicated by the FlexEnabled application. When a checkout of a license held in trusted storage is attempted and fails because a repair of trusted storage is required, the Flexible API returns a `LM_REPAIR_NEEDED` error. Code in the FlexEnabled application can then call the application activation utility directly or indicate that a repair of trusted storage is required.

The following description gives a typical sequence of operations required to repair a single fulfillment record.

Task ***To repair a single fulfillment record using request and response codes***

1. Create a short code object with **flxActShortCodeCreate** or **flxActShortCodeCreateFromBuffer**.
2. Generate a repair request code using **flxActShortCodeGetRepairRequest**.
3. Output the repair request code.
4. Clean up memory and delete temporary files. Delete the short code object using **flxActShortCodeDestroy**.
5. Copy the repair request code and paste it into the manual response generator. Use the manual response generator to create the required response code. For instructions on operating the manual response generator, see Chapter 14, "The Manual Response Generator."
6. Input the response, and then process it using **flxActShortCodeProcessResponse.**
7. If the response is a repair response, then the fulfillment record is repaired. If the response reports an error, then the fulfillment record is not repaired and **flxActShortCodeProcessResponse** returns non success. An error response is returned when the repair is not allowed and also to report an error in the transaction.

Display Pending Request Code

For a general introduction to request and response codes, see Transactions Using Request and Response Codes.

Only one short-code transaction per short-code alias can be in progress at any time. So, if there is an outstanding short-code transaction and a another short-code transaction is required, the activation utility may need to remind the end user that a transaction is in progress. Then the user can decide whether to complete the current transaction before requesting a new transaction. When the user decides to complete the current transaction they will require the request code for the outstanding transaction.

The Activation API functions can be used to display a pending request code as given in the following typical sequence of operations.

Task ***To display a pending request code***

1. Create a short code object with **flxActShortCodeCreate** or **flxActShortCodeCreateFromBuffer**.
2. Obtain the pending request code, if any, using **flxActShortCodeGetPendingRequest**.
3. If there is a current short-code transaction, output the request code.
4. Clean up memory and delete temporary files. Delete the short code object using **flxActShortCodeDestroy**.

In the example application activation utility, *compTranUtil* (binaries appcomptranutil and servercomptranutil) the file compTranUtilShortCode.c includes the functionality to perform short code transactions.

Cancel Outstanding Short-Code Transaction

For a general introduction to request and response codes, see Transactions Using Request and Response Codes.

Only one short-code transaction can be in progress at any time for a given short code ASR. So, if there is an outstanding short-code transaction and the pending request cannot be used (for example because it is the wrong type), the activation utility must cancel the outstanding transaction before creating a new transaction.

The following procedure provides a typical sequence of operations required to cancel an outstanding short-code transaction.

Task ***To cancel an outstanding short-code transaction***

1. Create short code object with **flxActShortCodeCreate** or **flxActShortCodeCreateFromBuffer**.
2. Cancel the outstanding short-code transaction using **flxActShortCodeCancelRequest**.
3. Clean up memory and delete temporary files. Delete the short code object using **flxActShortCodeDestroy**.

In the example application activation utility, *compTranUtil* (binaries appcomptranutil and servercomptranutil) the file `compTranUtilShortCode.c` includes the functionality to perform short code transactions.

Error Handling

The following functions are provided for error handling.

- **flxActCommonHandleGetError**
- **flxActShortCodeGetDenyReason**
- **flxActShortCodeGetWarning**
- **flxActShortCodeGetErrorDetail**

For examples of their use see the compTranUtil sample file `compTranUtilShortCode.c`.

Adding Custom Data to a Short-Code Request

You can send custom data in a short-code request by defining a request override in the ASR. This allows your application code to add custom data to the vendor data dictionary and this information is then included in the request code. Note that when using FNO as the server the Custom Data override is specified when the Short Code ASR is created. The ASR should only be edited manually for non-FNO servers or if advised by Flexera (for example for a workaround).

Task ***To add custom data to a request code***

1. Create a short-code ASR.

   See Creating an Activation Specification Record and ShortCodeData Element (in Appendix A, XML Schemas) for information about short-code ASRs. To create a short-code ASR that includes custom data, the default format for a request code must be overridden by adding an `Overrides` entry (and the required XML sub-elements) to the `ShortCodeData` section of the ASR.

   To avoid name clashes with items in other data dictionaries, it is recommended that you prefix `VENDOR_` (for example, **VENDOR_DATA1**) to each custom data item name (where **vendor** is your vendor name).

2. Create a short-code request. See transaction descriptions in previous sections.

3. Build capability into the utility to capture the value of the custom information, either from the system automatically or from the user.

4. Add the custom data to the request using **flxActShortCodeSetVendorData**.

5. Generate the request code using one of the following functions:

   - **flxActShortCodeGetActivationRequest**
   - **flxActShortCodeGetReturnRequest**
   - **flxActShortCodeGetRepairRequest**

   The custom data is included in the short code.

Example Overrides Elements for Request Code

This example shows an override with the key name of “VENDOR_DATA1”in the vendor dictionary. This example allows you to pass custom data in the activation request message. The activation utility passes the value `VENDOR_DATA1` as the key argument to the **flxActAppActivationVendorDataSet** function when setting the value.

```
<RequestOverrides>
   <Items>
      <Item>
         <Group>REQ_GROUP_ACTIVATION</Group>
         <Name>VENDOR_DATA1</Name>
         <Type>DEC</Type>
         <Size>4</Size>
         <Location>VENDOR_DICTIONARY</Location>
      </Item>
   </Items>
</RequestOverrides>
```

13

Working with the Software Publisher Utilities

This chapter provides usage information for utilities that you may need during the development and packaging phases of your licensing implementation. The following utilities are available in the toolkit:

- `lmcrypt`—Signs license file containing feature definition lines.
- `preptool`—Processes and prepares FlexEnabled applications, activation applications, and the vendor daemon to run in a trusted storage–based licensing environment. It is also used to create publisher-specific versions of activation utilities, and to add tamper resistance to your applications. (Not required when using activation in a `notr` toolkit.)
- `tcencrypt`—Encrypts sensitive data that defines the anchoring and binding used for trusted storage.
- `xmlsign`—Signs ASRs and Trusted Configuration files.
- `tsreset`—Resets the state of trusted storage.
- `tsactdiags`—Diagnoses trusted storage–related issues for client-side and server-side applications.
- `asrgen`—Creates an ASR used for local activation.
- The Manual Response Generator utility is available to test activation transactions using messages stored in XML files.

lmcrypt

The `lmcrypt` utility can be used to generate licenses. The source to `lmcrypt` is in the `machind` directory and can be used by you as the basis for your own license generation utility.

Usage

```
lmcrypt [files][-i infile] [-o outfile] [-maxlen n] [-e errfile]
       [-verfmt { 2 | 3 | 4 | 5 | 5_1 | 6 | 6_1 | 7 | 7_1 | 8 }]
        [-decimal] [-help]
```

If no input file is specified, or if specified as "-" or stdin, standard input is used. If no output file is specified, or if specified as "-" or stdout, standard output is used. `files` are read and written back in place. If no file arguments are specified, `lmcrypt` reads from stdin and writes to stdout. All signatures are recomputed. `lmcrypt` works only on lines that have a vendor name matching the vendor's daemon name.

If this is not possible, an error is produced, and the affected license line is left unaltered.

The maximum line length is controlled with the `-maxlen` argument. A value of 50 is commonly used to make shorter lines less subject to mailers inserting new lines. Licenses can be generated to be compatible with older versions by using the `-verfmt` argument.

License File

Once you know the format of the FEATURE or INCREMENT lines that you need for your license model, you create a license file. `lmcrypt` replaces the signature in this template, which can be entered simply as `SIGN=0`, with the correct signature. The license is then ready to ship to the end user.

Using lmcrypt

To use `lmcrypt`, you need to either understand the license file syntax or use an example license file. Examples are available in the `examples/licenses` directory.

Task ***To use lmcrypt:***

1. Copy an existing good file to another name, say, `newlicense`.
2. Edit `newlicense` and make any desired changes, such as changing the feature name, the number of licenses, or adding new features, and then save the file.
3. Change the existing signature on each FEATURE line to `SIGN=0`.
4. Type the following:

   `lmcrypt newlicense`

   `newlicense` is then filled with correct signatures and is usable by an end user. Comments are passed through unaltered.

Error Returns

Errors are printed to stderr, or as specified with `-e`. Most errors prevent generation of signatures and the text is output unchanged from the input. An example of error reporting: If `-e` is not used, the error messages also appear at the top of the output file.

Input:

```
FEATURE f1 demo 1.a50 01-jan-2020 uncounted HOSTID=08002b32b161 SIGN=0
```

Error reported:

```
stdin:line 1:Bad version number — must be floating point number, with no letter.
```

preptool

The `preptool` utility is used to create a publisher- or product-specific activation library component for an application or utility that requires it. It also adds the necessary public and private keys to the components to enable security and activation.

The `preptool` utility can also be used to create a *publisher* `preptool`. A publisher `preptool` is a version of the `preptool` that has publisher-specific information embedded within it. You can distribute this version to your development teams to ensure that the correct publisher-specific information is used for all products. For more details, see Publisher-Specific Preptool.

`preptool` is not backward compatible with older versions of `preparchive*.yaa`. Ensure the version of `preptool` matches the version of `preparchive*.yaa` that is included in the toolkit.

The `preptool` must be used with all the following application binaries:

- FlexEnabled applications that use tamper-resistant application security
- FlexEnabled applications that use licenses held in trusted storage
- Vendor daemons that serve activated licenses (either concurrent or activatable)
- Activation utilities (see Chapter 10, “Building an Application Activation Utility for Single-Action Transactions,” Chapter 11, “Building a Server Activation Utility for Single-Action Transactions,” or Chapter 12, “Building an Activation Utility for Composite Transactions.”)
- `tsreset`
- `tsactdiags`

Note • *While* `preptool` *is used to prepare applications to run in a trusted storage–based licensing environment, it is also required to prepare any license file–based licensed application that contains SDTs. For more information, see the chapter “Implementing Secure Data Types (SDTs)” in the Development Environment Guide.*

Using preptool Sequentially with Other Utilities

The order in which you run `preptool` and other utilities and applications can be important.

Signing Applications

Sign applications *after* using `preptool`. An error message is displayed if you attempt to prepare a Mac application that has been code-signed. On Windows, if you run `preptool` on a binary after it has been signed, the binary may not operate properly.

Any file that is prepped using `preptool` should also be signed by software publishers (for example, `demo_libFNP.dll` and `demo.exe`, where “demo” is the vendor name).

Usage

```
preptool [-v value] [-p preparchive_path] [-a activation_settings_path] [-s|-c] [-C publisher_preptool]
prepconfig
```

Where:

Table 13-1 • Preptool Application Input and Switches

Input/ Switch	Description
`-v` *`value`*	Sets level of command line output. When used, this option overrides the verbose `level` setting in the `preptool` configuration file.
`-p` *`preparchive_path`*	Path to the file `preparchive_releasenumber`, where *releasenumber* is the activation component's release number. If you use this option, ensure that the `preparchive` file resides in the current directory.
`-a` *`activation_settings_path`*	The pathname of an activation settings file. When used, this option overrides the setting in the `preptool` configuration file. This option is not available in a publisher `preptool`.
`-s`	Sets server mode so that the activation library creates and accesses server-side trusted storage. When used, this option overrides the `isserver` setting in the `preptool` configuration file.
`-c`	Sets client mode so that is the activation library creates and accesses client-side trusted storage. When used, this option overrides the `isserver` setting in the `preptool` configuration file.
`-C` *`publisher_preptool`*	Create a publisher `preptool`. The pathname of the publisher `preptool` is specified by `publisher_preptool`. The publisher `preptool` contains embedded values from the activation settings file specified in the `preptool` configuration file or with option `-a`. See Publisher-Specific Preptool for details of how to use it. This option is not available in a publisher `preptool`.
`-sdtonly`	Use to prepare a file-based licensing application to which you have added Secure Data Types (SDTs). ***Note*** *• Use this option only if you have purchased rights to the Trusted Storage/ Activation licensing package from Flexera, but are using this package to build instead a license–file–based licensed application that uses SDTs. However, publishers who have purchased rights to the license–file–based licensing package only and want to prepare an application containing SDTs must use a different activation settings file (`sdt_publisher.xml`) to specify this option. For more information, see the chapter "Implementing Secure Data Types (SDTs)" in the Development Environment Guide.*
`prepconfig`	The path and file name for the `preptool` configuration file.

Note • *The following are special concerns about* `preptool`*:*

- *The* `preptool` *utility is not provided in* `notr` *toolkits. A* `notr` *toolkit uses a publisher-specific version of the activation library that is created when the toolkit is built. For details, see Chapter 1, "Overview of Trusted Storage–Based Licensing."*
- *If both 32-bit and 64-bit standard toolkits are installed on the same system, ensure that the* `PREP_ARCHIVE` *environment variable is set correctly for the toolkit you are using. For details, see the section "32-bit and 64-bit Toolkit Installation" in the Development Environment Guide.*

Examples of Preptool Configuration Files

Use the following sample `preptool` configurations to set up your own configuration files.

Preptool Configuration File for Windows Activation Utility

The following sample `preptool` configuration prepares an activation utility for use on Windows operating systems.

To support best practice, this configuration sample points to two prepped Trusted Configurations (one each for physical and virtual environments) that are, in turn, included in the utility code when the utility is built. The utility then uses the appropriate prepped Trusted Configuration to generate the initial section in trusted storage when needed.

For more details, see the section Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage."

Configuration Description

This sample `preptool` configuration includes the following settings:

- The activation settings file used in the preparation (in this example, `demo.xml`). For a file description, see S_ActivationSettings.xsd in Chapter A, "XML Schemas."
- The names of the files containing the two prepped Trusted Configurations (in this example, `DemoTrustedConfigOut.xml` for physical environments and for virtual environments).
- The `isserver=false` option, indicating that `preptool` is to generate an activation library for client-side usage (that is, a library configured to access trusted storage on the system where the FlexEnabled application is installed).
- Certificate settings (which are blank here because they are for use by UNIX systems only).
- The name of the activation library component to be created (in this example, `appcomptranutil_lib.dll`).
- The name of the client-side activation utility using this library (in this example, `appcomptranutil.exe`).
- Preferences for the preparation process:
  - Level of command-line output. In this example, `verbose level=0` sets the output to the minimum.
  - Option to turn on or off Anti-Debugger techniques (ADT) that prohibit debugging. (ADT implementation is platform-specific). In this example, `debuggable value="false"` turns on ADT to prohibit debugging.

Configuration Text

The following shows the contents of the `preptool` configuration file:

```
<?xml version="1.0" encoding="UTF-8"?>
<config>
        <activationsettings path="ActivationSettings\publisher.xml" isserver="false"/>
```

```
        <trustedconfiguration path="publisher\DemoTrustedConfigOut.xml">
        <certificate path="." hostverify="2" peerverify="true" />
        <items installpath="." runtimeloadname="appcomptranutil_libFNP.dll" >
                <item path="appcomptranutil.exe" />
        </items>
        <preferences>
                <verbose level="0" />
                <debuggable value="true" />
                <stripwarnings value="false" />
        </preferences>
</config>
```

Preptool Configuration File for Windows FlexEnabled Application

The following sample `preptool` configuration prepares a FlexEnabled application for use on Windows operating systems.

Note • *Prepped Trusted Configuration files are not specified in this configuration because this application does not perform activations. (If an application performs activations, you should include the* `trustedconfiguration` *elements, as shown in the first example Preptool Configuration File for Windows Activation Utility.)*

Configuration Description

This sample `preptool` configuration includes the following settings:

- The activation settings file used in the preparation (in this example, `demo.xml`). For a file description, see S_ActivationSettings.xsd in Chapter A, "XML Schemas."
- No prepped Trusted Configuration files specified because this application does not perform activations. (If it did, the `trustedconfiguration` elements should be included, as shown in the first example Preptool Configuration File for Windows Activation Utility).
- The `isserver=false` option, indicating that `preptool` is to generate an activation library for client-side usage (that is, a library configured to access trusted storage on the system where the FlexEnabled application is installed).
- Certificate settings (which are blank because they are for use by UNIX systems only).
- The name of the activation library component to be created (in this example, `product_lib.dll`).
- The name of the FlexEnabled application using this library (in this example, `FlexEnabled.exe`).
- Preferences for the preparation process:
  - Level of command-line output. In this example, `verbose level=0` sets the output to the minimum.
  - Option to turn on or off Anti-Debugger techniques (ADT) that prohibit debugging. (ADT implementation is platform-specific). In this example, `debuggable value="false"` turns on ADT to prohibit debugging.

Configuration Text

The following shows the contents of the `preptool` configuration file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
  <activationsettings path="ActivationSettings\demo.xml" isserver="false" />
  <certificate path="" hostverify="" peerverify="" />
  <items runtimeloadname="product_lib.dll" />
```

```
    <item path="FlexEnabled.exe" >
  </items>
  <preferences>
    <verbose level="0" />
    <debuggable value="false" />
  </preferences>
</config>
```

Preptool Configuration File for Vendor Daemon on Mac

The following sample configuration prepares a vendor daemon for use on OS X.

Note • *Prepped Trusted Configuration files are not specified in this configuration because this application does not perform activations. (If an application performs activations, you should include the* `trustedconfiguration` *elements, as shown in the first example Preptool Configuration File for Windows Activation Utility.)*

Configuration Description

This sample `preptool` configuration includes the following settings:

- The activation settings file used in the preparation (in this example, `publisher.xml`). For a file description, see S_ActivationSettings.xsd in Chapter A, "XML Schemas."
- The `isserver=true` option, indicating that `preptool` is to generate an activation library for server-side usage (that is, a library configured to access trusted storage on the license server).
- Certificate settings:
  - The pathname of the SSL certificate (`UTN.pem`) to use for the open source code used by FlexNet Publisher Licensing Toolkit.
  - Option to verify the hostname in the SSL certificate. In this example, the value `2` turns on verification.
  - Option to verify the peer certificate against the list of certificates in the certificate path. In this example, `true` turns on verification.
- The name of the activation library component to be created (in this example, `myproduct.dylib`).
- The installation location and path used to call the component (in this example, `/User/myname/FNP` and called using the path `/User/myname/FNP`).
- Path (`VendorDaemon`) indicating that the activation library is for use by the vendor daemon.
- Preferences for preparing the vendor daemon:
  - Level of command-line output. In this example, `verbose level=2` provides a medium amount of output.
  - Option to turn on or off Anti-Debugger techniques (ADT) that prohibit debugging. (ADT implementation is platform-specific). In this example, `debuggable value="false"` turns on ADT to prohibit debugging.
  - Option to generate a warning message if the application has not been stripped of items that might compromise license security. In this example, `stripwarnings value="true"` turns on message generation. (This is the default.)

Configuration Text

The following shows the contents of the `preptool` configuration file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
  <activationsettings path="/Users/myname/FNP/ActivationSettings/publisher.xml" isserver="true"/
  <certificate path="UTN.pem" hostverify="2" peerverify="true" />
  <items runtimeloadpath="/Users/myname/FNP/" runtimeloadname="myproduct.dylib" installpath="/User/
myname/FNP >
    <main path="VendorDaemon" />
  </items>
  <preferences>
    <verbose level="2" />
    <debuggable value="false" />
    <stripwarnings value=”true” />
  </preferences>
</config>
```

Preptool Configuration File for Activation Library Used by Multiple Executables

When your product consists of multiple executables (for example, both the executable and the DLLs that are called by the executable), consider having `preptool` create a single activation-library component that can be accessed by all executables. This single activation library is normally required only when both the FlexEnabled application uses SDTs and its licensing functionality is distributed among several executables.

Important • *Each executable must load this activation library separately if the executable contains either licensing functions (check-out or check-in) or SDTs. For details, see* *Chapter 5, “Adding Function Calls to a FlexEnabled Application.”*

Configuration Text

The following sample `preptool` configuration file creates the single activation library, `product_lib.dll`, which three executables (`FlexEnabled.exe`, `ModuleA.dll`, and `ModuleB.dll`) can access:

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
  <activationsettings path="ActivationSettings\demo.xml" isserver="false" />
  <certificate path="" hostverify="" peerverify="" />
  <items runtimeloadname="product_lib.dll" />
    <item path="FlexEnabled.exe" >
    <item path="ModuleA.dll" >
    <item path="ModuleB.dll" >
  </items>
  <preferences>
    <verbose level="0" />
    <debuggable value="false" />
  </preferences>
</config>
```

Note • *Prepped Trusted Configuration files are not specified in this configuration because this application does not perform activations. (If an application performs activations, you should include the* `trustedconfiguration` *elements, as shown in the first example* *Preptool Configuration File for Windows Activation Utility**.)*

Preptool Configuration File for Creating Licensed Product for Third-Party Developers

The `preptool` configuration file allows you to create a licensed product package that includes both a *release* (not debuggable) and a debuggable version of the activation library. This configuration is provided so that you can supply a single package to third-party developers who are creating add-ons for a licensed product. This enables them to test their add-ons in an environment where they can use debugging tools and then subsequently test their add-ons with the release version of the activation library.

Caution • *Use the functionality described in this section for creating packages for third-party developers only. To maintain license security, do not release the debuggable version of the activation library to customers. Whenever the activation library is created inside a package (for example on the Mac), remove the debuggable version of the activation library before giving the licensed product to customers. This library might not be visible in the directory structure displayed.*

Generating Both Release and Debuggable Versions of the Product

The standard `preptool` configuration file includes the `<debuggable value="value" />` element that uses a `true` or `false` value to allow or prohibit debugging. To create both a release and a debuggable version of the activation library in a single run of `preptool`, replace this element with the following element:

```
<security value="both" />
```

Configuration Sample

The following sample `preptool` configuration creates both a release and a debuggable version of the activation library. (The release version is named `myproduct.dylib`; the debuggable version is automatically named `myproduct.dylib.debuggable`.)

Note • *Prepped Trusted Configuration files are not specified in this configuration because this application does not perform activations. (If an application performs activations, you should include the* `trustedconfiguration` *elements, as shown in the first example* *Preptool Configuration File for Windows Activation Utility**.)*

For a descriptions of the other settings, see the previous configuration samples.

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
  <activationsettings path="/Users/myname/FNP/ActivationSettings/publisher.xml" isserver="false" />
  <certificate path="UTN.pem" hostverify="2" peerverify="true" />
  <items runtimeloadpath="/Users/myname/FNP/" runtimeloadname="myproduct.dylib" installpath="/User/
myname/FNP >
    <main path="myproduct" />
  </items>
  <preferences>
    <verbose level="2" />
    <security value="both" />
    <stripwarnings value=”true” />
  </preferences>
</config>
```

The following table lists the valid values for the element security:

Table 13-2 • Valid Values for Preptool Configuration File Element Security

Value	Description
<security ="both" />	Creates both a release and debuggable version of the activation library. They have the same name, as specified in the runtimeloadname element, except that the debuggable version has the suffix .debuggable.
<security ="debuggable" />	Creates a debuggable version of the activation library with the name specified in the runtimeloadname element.
<security ="nodebug" />	Creates a release version of the activation library with the name specified in the runtimeloadname element.

Publisher-Specific Preptool

A publisher preptool is a version of the preptool that has embedded in it publisher specific information. The standard preptool obtains this information from the activation settings file:

- Publisher ID (supplied by Flexera)
- Vendor name (supplied by Flexera)
- Security key seeds

A publisher preptool has these values embedded within it.

To create a publisher preptool, use the -C option with preptool. It is recommended that a publisher preptool has a easily recognizable name, for example preptoolpub. Use a publisher preptool in the same way as preptool.

For usage, see preptool. The syntax and the command-line options available for a publisher preptool are the same as for preptool except that the following options are not available:

- -a *activation_settings_path*
- -C *publisher_preptool*

The activationsettings path defined in the preptool configuration file is not used.

You can create a publisher preptool and give it to your application developers to use to prep their applications. This ensures that all your licensed applications use the correct publisher specific settings.

Identifying Publisher preptools

To identify whether your preptool version is a standard preptool or a publisher preptool, issue one of these commands: preptool, preptool -h or preptool --help. The following text is included in the top section of the output:

```
Publisher-Specific Version
```

tcencrypt

The `tcencrypt` utility encrypts sensitive data in an XML document. It is used with documents that contain data that defines the publisher-specific Trusted Configuration for anchoring and binding of trusted storage. It also contains the seeds for creating the activation transaction keys. The XML document to be signed by `tcencrypt` must conform to the format specified in O_ConfigPhaseOne.xsd (in Chapter A, XML Schemas). An example document included in the toolkit so that you can see how `tcencrypt` works

- `DemoTrustedConfigIn.xml`

Usage

```
tcencrypt -i infile -o outfile [-k key_outfile] [-s]
```

Where the following switches and input are used:

Table 13-3 • tcencrypt Utility Switches and Input

Input/ Switch	Description
-i *infile*	The pathname to an unencrypted XML document whose format is defined by the `O_ConfigPhaseOne.xsd` schema.
-o *outfile*	The pathname of the resulting encrypted XML document. This output file has a format defined by the `P1_ConfigPhaseTwo.xsd` schema.
-k key_outfile	(Optional) The pathname for the resulting key file. This file is used by the manual response generator. The format of the file contents is defined by the `P2_ConfigKeyOut.xsd` schema. ***Note •*** *It is important that you keep your encryption seeds settings secure,. For more details see the Development Environment Guide, section "Keeping Encryption Settings Secure".*
-s	(Optional) Option that sets the `Revisiontype` to SPE within the outfile. This setting ensures that this Trusted Configuration overwrites any other Trusted Configuration and can only be overwritten by another Trusted Configuration with a SPE setting.

The input XML document that you specify is encrypted and output in two forms:

- **outfile**—This file is used in one of three ways:
  - As an input to `asrgen` when creating an ASR.
  - As input to an activation utility's `preptool` configuration. When `preptool` is run on the utility executable, the Trusted Configuration information is inserted into the utility code. This inserted configuration, known as the *prepped Trusted Configuration*, is then deployed as needed to create an initial trusted section automatically *before* an activation request is sent. For more information about prepped Trusted Configuration, see the section Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage."
  - To create XML documents used by the Manual Response Generator to create confg responses. Config responses contain the binding parameters and anchor settings used to create or modify a trusted section in trusted storage on the requesting system. When the `outfile` is used in this way it must first be signed using `xmlsign`.

- **key_outfile**—This file contains the private keys needed by the Manual Response Generator when processing requests that use the Trusted Configuration specified in the input XML document. It is recommended that you name this file so that it can be readily identified within the manual response generator. (For example, where the trusted ID is 12, the key file could be named `TrustedId12Keys.xml`.)

See Also
Transactions with a FlexEnabled Client
Summary of the Activation Process
infile—O_ConfigPhaseOne.xsd
outfile—`asrgen`, `P1_ConfigPhaseTwo.xsd`, `W4C_CreateConfigResponseClientParameter.xsd`, and `W4S_CreateConfigResponseParameter.xsd`
key file—The Manual Response Generator and `P2_ConfigKeyOut.xsd`

xmlsign

The `xmlsign` utility signs XML documents used for activation based on seeds defined in the activation settings file.

Usage

`xmlsign -a` *`as_file`* `-i` *`infile`* `-o` *`outfile`*

Where:

Table 13-4 • `xmlsign` Utility

Input	Description
`-a` *`as_file`*	The pathname to an activation settings file (S_ActivationSettings.xsd).
`-i` *`infile`*	The pathname to an XML document. The following documents must be signed: ● Activation specification record, *ASR* (B_ActivationSpecificationRecord.xsd) ● Trusted Configuration (P1_ConfigPhaseTwo.xsd)
`-o` *`outfile`*	The pathname of the resulting signed XML document.

Using the seeds provided in the activation settings file, a private key is generated and used to add a signature to every section in the `infile` that has a “Signature” element.

The signature is created using all the elements at the same level as the “Signature” element in the schema. This tamper proofs all these elements in the document: any changes to any of these elements result in a failed validation for that section of the document. The signatures computed by the xmlsign utility are written into the “Signature” elements in the output document and replace any existing signatures.

Note • *You do not need to remove existing signatures when editing XML documents used as input to xmlsign utility. The existing signatures are recomputed.*

Example

The following diagram shows a typical signed XML document. The side bars indicate the elements included within the signature.

```
  <?xml version="1.0" encoding="utf-8" ?>
- <SecureTrustedConfigInformation
    xmlns="http://www.macrovision.com/fnp/2004/11/trusted">
  - <ConfigData>
      <PublisherId>256</PublisherId>
    - <ServerConfig>
        <Id>1</Id>
        <Revision>NumberOne</Revision>
        <Data>8EF07B8AFBC7E4C12442E953BBAB6B147000B</Data>
      </ServerConfig>
    - <ClientConfig>
        <Id>1</Id>
        <Revision>NumberOne</Revision>
        <Data>B7C07D6EA4FA53589B2B88E2A01788DE3E6B5</Data>
        <Signature>01aecac14b59b300291299f40</Signature>
      </ClientConfig>
      <Signature>0a7f0b067076b483bc7b287cb</Signature>
    </ConfigData>
  </SecureTrustedConfigInformation>
```

Figure 13-1: Signed XML Document

tsreset

The `tsreset` utility cleans up a publisher's trusted storage either by clearing the data from trusted storage or by removing the publisher's trusted-storage files altogether. This utility is especially useful should trusted storage at the end-user's site enter an invalid state. The publisher can provide this utility to the end user, who then runs the appropriate utility command as part of the trusted-storage recovery process.

On Windows platforms, `tsreset` can also be used to clean up excess anchor copies. Orphaned anchors typically occur on machines used for testing older versions of FlexNet Publisher, for example where `tsreset` is run repeatedly, or where trial anchors are reset. Once they exist they can affect all versions of FlexNet Publisher. If the copies build up to large numbers, they can lead to long delays in opening trusted storage. Therefore, some of the `tsreset` options also provide functionality to remove these orphaned anchors.

Use the `preptool` to configure `tsreset` to act on only your publisher-specific trusted-storage files and on either client-side or server-side trusted storage. The utilities `tsreset_app` and `tsreset_svr` are created (under `<platform_dir>`) when you build the toolkit:

- `tsreset_app` cleans up client-side trusted storage
- `tsreset_svr` cleans up server-side trusted storage

The out-of-the-box utilities `tsreset_app` and `tsreset_svr` can be used to clean up the sample trusted storage created during demonstration or trial-licensing activities.

Important • *The* `tsreset` *utility operates on only the trusted-storage data and files associated with the publisher for which the utility is prepared (using* `preptool`*). It does not touch trusted-storage data and files for other publishers in the trusted-storage folder; nor does it ever remove the trusted-storage folder.*

Usage

Run the appropriate utility with one of the following clean-up options:

Table 13-5 • `tsreset _app` and `tsreset_svr` Options

Input	Description
`-reset`	Deletes all fulfillment records in trusted storage. In addition, it deletes the segmentation (sections) of trusted storage used to store the Trusted Configuration data. On Windows platforms, also deletes orphaned anchors.
`-reset nottc`	Deletes all fulfillment records in trusted storage. However, the segmentation (sections) of trusted storage used to store the Trusted Configuration data remains in tact. On Windows platforms, also deletes orphaned anchors.
`-delete`	Deletes the trusted-storage data files. However, the command generates time-stamped archive copies of all deleted files (*.sav files), which might be useful when investigating problems. On Windows platforms, also deletes orphaned anchors.
`-delete all`	Deletes all the trusted-storage files, including any archive copies. On Windows platforms, also deletes orphaned anchors.
`-logreport`	Writes anchor counts to the event log. Use this command to see if orphans are present. (Windows platforms only)
`-logreport verbose`	Writes details of individual anchors to the event log. (Windows platforms only)
`-anchors orphan`	Removes only excess anchor copies; it does not reset or delete trusted storage. This enables users to recover from issues related to orphaned anchors without having to re-establish licenses on a production server. (Windows platforms only)

Tip • *You can also use the sample activation utilities to delete fulfillment records in the sample trusted storage (for example,* `appcomptranutil -deleteFR <fulfillmentID>` *or* `servercomptranutil -deleteFR <fulfillmentID>`*.*

tsactdiags

Use the `tsactdiags` utility to diagnose trusted storage–related issues for client-side and server-side applications. In addition, `tsactdiags` can be used to clean up orphaned anchors. `tsactdiags` outputs a log in unencrypted format, reporting on the past and current state of trusted storage. This utility is suitable for distribution to the end user.

`tsactdiags` is currently available only for Windows platforms.

Use the `preptool` to configure `tsactdiags` to report on your publisher-specific trusted-storage files and on either client-side or server-side trusted storage. The `preptool` configuration files are included in the toolkit under `<install_dir>\machind\activation\prep_xml`. When you build the toolkit, the utilities `tsactdiags_<vendor>_app` and `tsactdiags_<vendor>_svr` are created under `<platform_dir>` (where `<vendor>` is your publisher name):

- `tsactdiags_<vendor>_app` diagnoses client-side trusted-storage issues.
- `tsactdiags_<vendor>_svr` diagnoses server-side trusted-storage issues.

It is recommended that the Revision and ID fields in the Trusted Configuration file for `tsactdiags` be different to the values of these fields that are embedded in the FlexEnabled application.

Note • *The `tsactdiags` utility reports on only the trusted-storage data associated with the publisher for which the utility is prepared (using `preptool`).*

Usage

`tsactdiags [-v] [-o file_name] [-r] [-h]`

The following table describes the syntax elements:

Table 14 • `tsactdiags` Command-Line Syntax

Command	Description
-v	Displays the version of this utility.
-o *file_name*	Name of the output file that will contain the decrypted and filtered event log entries.
-r	Removes orphaned anchors.
-h	Displays usage information.

asrgen

The `asrgen` utility creates an activation specification record (ASR) for deployment on a FlexEnabled client or a FlexNet license server. The following inputs are required:

- **License template**—The file that contains the feature definition lines required in the ASR.
- **Encrypted Trusted Configuration output file**—The file created when the `tcencrypt` utility is run on the Trusted Configuration input file. This file is required whether you are deploying a prepped Trusted Configuration or embedding the Trusted Configuration definition in the ASR.

Usage

`asrgen -c license_template -i config_template -o asr_output_file [-t trial_id] [-x XML_Snippet_file] [-v VM_ONLY/PHYSICAL] [-d] [-s]`

Where:

Table 13-1 • asrgen Utility Switches and Inputs

Input	Description
`-c license_template`	A file that contains the feature definition lines required in the ASR and a `SERVER` line that contains a HostId that is difficult to spoof.
`-i config_template`	The Trusted Configuration file. This output file from `tcencrypt` contains details of the trusted section in trusted storage in which the fulfillment record contained within this ASR will be stored. ***Note*** • *If the activation utility is set up to deploy a prepped Trusted Configuration to create an initial trusted section, you must still specify the Trusted Configuration file. The utility needs this file to obtain the necessary Trusted ID and Publisher ID. For more information about the prepped Trusted Configuration, see the section Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage."*
`-o asr_output_file`	The pathname of the resulting ASR file. This file must be edited and then signed using `xmlsign` before it is used.
`-t trial_id`	A number from 1 to 255 that uniquely identifies the trial anchor to be created. This command line input generates the `PUBLISHER_TRIAL_ID` element in the ASR and sets the value to *trial_id*. For more information about trial anchors, see Anchoring. ***Note*** • *`trial_id` can be greater than 255 if an xml snippet is also specified with the `-x` option. Each trial pack grants 255 additional trials.* When you define a Trial ID using this input, the `PUBLISHER_DEFINED_CRITERION` value is automatically set to **`LOCAL_ONCE`**. You cannot set the `PUBLISHER_DEFINED_DURATION` value at the command line. This must be configured manually in the ASR. For more information about the meaning of these elements, see DataDictionary Element in Appendix A, "XML Schemas."
`-x XML_Snippet_file`	The Trial Pack ID XML snippet file that contains the XML necessary to implement additional trials. For more information, see Using Trial Packs and Trial Pack ID XML Snippet in Chapter 4, "Implementing Trial Licenses and Trial Packs."

Table 13-1 • asrgen Utility Switches and Inputs

Input	Description
`-d`	The switch required to omit the Trusted Configuration definition from the ASR. Use this option if you are using a prepped Trusted Configuration to create the initial section in trusted storage for loading licenses from the ASR. (You must still specify the Trusted Configuration output file using the `-i` switch.) For more information about prepped Trusted Configuration, see the section Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage." ***Note*** • *This switch is applicable for ASRs that install license rights on a FlexEnabled client. For ASRs that install license rights on a license server, this switch is irrelevant because the `-s` switch, required for generating license-server ASRs, forces the omission of the Trusted Configuration definition in the ASR for security purposes.*
`-s`	The switch required if the ASR is being generated for use on a license server. When generated, the ASR includes the appropriate `<IsServer>true</IsServer>` designation and, for security purposes, omits the Trusted Configuration definition. ***Important*** • *The ASR being generated for use on a license server requires an existing trusted section. Therefore, you must also include the `-i` switch to specify the Trusted Configuration file from which* `asrgen` *retrieves information about this section. If the section does not exist when the ASR is processed, the activation utility either deploys a prepped Trusted Configuration to create an initial trusted section or fails the activation altogether. See Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage.")*

Discussion

For information about ASRs, how and when they are used, read Chapter 1, "Overview of Trusted Storage–Based Licensing."

Task ***To use the asrgen utility***

1. Create a license template file that contains the feature definition lines required in the fulfillment record.
2. Create an encrypted Trusted Configuration file (or use the prepped Trusted Configuration file set up with your toolkit to create an initial trusted section).
3. Use `asrgen` to create an unsigned ASR file.
4. Edit the entries in the ASR file as required.
5. Use `xmlsign` to sign the ASR file.

License Template

The license template file required as input to `asrgen` should have the following structure:

- SERVER line
- VENDOR line
- FEATURE and INCREMENT lines

The hostid in the SERVER line is used to add the signatures to the feature definition lines. You must choose a hostid that is not easy to spoof to ensure that feature definition lines cannot be used outside of a fulfillment record. A dongle identity is secure provided that it is not in use. For example, **FLEXID=9-a6300155** is secure because Flexera will never release it for use. The hostid specified in the license template file is added by asrgen to the fulfillment record it creates as the value for the serverHostID entry.

A feature definition line contains the basic license information for a feature. It is similar to standard FEATURE and INCREMENT lines but has some limitations on its contents. For details, see Chapter 15, "Defining Feature Definition Lines for a Fulfillment Record."

Trusted Configuration

A Trusted Configuration defines a section of trusted storage reserved for use by a specific publisher. This section, call a *trusted section*, is used to hold fulfillment records containing license rights for a FlexEnabled product. Each trusted section, identified by a Trusted ID, is secured with the following mechanisms:

- Anchors to the system, configured for each operating system
- Binding used to lock the trusted section to the machine

The transaction keys, used to secure the contents of activation transactions, are also defined per Trusted ID. The security settings used for trusted storage and activation transactions define the security of the license. License security, however, is not synonymous with the product security, which depends also on how securely the licensing components are bound to the product.

A Trusted Configuration, used to create a trusted section, is generated from an XML document defined by the schema O_ConfigPhaseOne.xsd. (For a description of this schema and instructions on how to use the XML document created from it, see O_ConfigPhaseOne.xsd in Appendix A, "XML Schemas.") Depending on your toolkit configuration, the encrypted configuration file is used to deploy a prepped or non-prepped Trusted Configuration.

For either type of deployment, asrgen requires the Trusted Configuration file as input.

Prepped vs Non-Prepped Trusted Configuration

If your toolkit is set up to deploy a prepped Trusted Configuration, the activation utility automatically creates an initial trusted section if necessary when invoked to process the ASR. Therefore, the ASR does not require that the Trusted Configuration be embedded within in it. (The -d switch omits the Trusted Configuration data in this case.) However, you still need to identify the encrypted Trusted Configuration file to asrgen so that it can obtain the necessary Trusted ID and Publisher ID to include in the ASR.

If non-prepped Trusted Storage is being deployed, the asrgen utility extracts the Trusted Configuration from the file and inserts it in the ASR. When the ASR is processed, the embedded Trusted Configuration information is used to create the trusted section if it does not already exist. If the settings within the ASR differ from those in an existing trusted section, then the trusted section for that Trusted ID might be updated.

Note • *For security reasons, ASRs generated for a license server require that specified trusted section already exist and that the Trusted Configuration not be embedded in the ASR. Therefore, license-server ASRs require that prepped Trusted*

Configuration be enabled to create the initial trusted section if necessary. While still a best practice, use of prepped Trusted Configuration is optional for ASRs generated for FlexEnabled clients.

For more information about the prepped Trusted Configuration, see the section Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage."

Functions of asrgen

The asrgen utility does the following to create the ASR:

- Adds signatures to the feature definition lines using `lmcrypt`.
- Creates a fulfillment record.

  The fulfillment record includes the signed feature definition lines and the hostid used to sign the feature definition lines. The fulfillment ID is generated and has the format `FID-<random number>`. The other entries for the fulfillment record are generated from a typical fulfillment record for a local license. These entries require editing to identify your product and the purpose of the ASR. For further details, see I_FulfillmentRecord.xsd in Appendix A, XML Schemas (and note specifically the references to `fulfillmentType` which must be set to correctly reflect how the ASR will be used).
- Adds the Trusted Configuration and the fulfillment record to a typical ASR.
- Prevents misuse of ASRs by marking them as intended for server-side activation only (thus preventing client-side ASRs from being used on the server and vice versa).

The ASR created by asrgen has the following settings:

- **LOCAL_ONCE**—Load ASR once only.
- **PUBLISHER_DEFINED_DURATION**—Expiry date is calculated from the current system date when the ASR is loaded.

Editing the ASR

The ASR generated by asrgen must be edited before it is signed and used. As a minimum, the following entries in the ASR should be edited:

- **PUBLISHER_DEFINED_DURATION**—Edit the `<Value>` element. Set it to **`0`** (zero) when a duration is not required. When a duration is required, set it to the duration in seconds. For example, to set a one-day license duration, enter the value is **`86400`** as shown in the following fragment from an ASR.

  ```
  <Entry>
  <Key>PUBLISHER_DEFINED_DURATION</Key>
  <Value>86400</Value>
  </Entry>
  ```
- **ProductId**—Insert your own product identity in the `<Value>` element. The following sample excerpt uses **`MyProduct-34657`** as the product name:

  ```
  <Entry>
  <Key>ProductId</Key>
  <Value>MyProduct-34657</Value>
  </Entry>
  ```
- **EntitlementId**—Insert your own Entitlement ID in the `<Value>` element. The following sample excerpt uses **`MyEntitilement-7849`** as the Entitlement ID:

```
<Entry>
<Key>EntitlementId</Key>
<Value>MyEntitlement-7849</Value>
</Entry>
```

If you have set the `PUBLISHER_DEFINED_DURATION` value element to zero, the duration is defined by the `Expiry` key in the fulfillment record. You might need to edit the `<Value>` element for this key, as shown in the following sample excerpt from the fulfillment record section:

```
<Entry>
<Key>expiry</Key>
<Value>12-Dec-2020</Value>
</Entry>
```

Example Files To Use with asrgen

Example files are included in the toolkit that enable you to use `asrgen`. See FlexNet Publisher's *Development Environment Guide* for instructions.

Manual Response Generator

The toolkit contains a tool called the manual response generator that enables software publishers to test the activation process using XML request and response messages stored as files. For more information about this tool, see Activation From a Manual Response Generator in Chapter 11, "Building a Server Activation Utility for Single-Action Transactions."

14

The Manual Response Generator

The manual response generator, `responsegenUI.exe` and `responsegen.*`, is a tool that can be used to test transactions. The toolkit for all platforms contains a command-line interface tool, `responsegen.*`. The Windows toolkits also contain a GUI version of this tool, `responsegenUI.exe`. For an overview of the activation process and the transactions involved, read Chapter 1, "Overview of Trusted Storage–Based Licensing." The manual response generator should be used for testing only. It is not intended to be used in production environments.

The instructions for using the manual response generator are structured in several sections:

- XML Documents Required—A description of the XML documents that are used by the manual response generator and links to instructions for generating these documents.
- Using the Manual Response Generator—A quick guide to how the Windows generator interface works.
- Activation Request—Detailed instructions with explanations of how to use the manual response generator GUI to respond to an activation request.
- Return Request—Detailed instructions with explanations of how to use the manual response generator GUI to respond to a return request.
- Repair Request—Detailed instructions with explanations of how to use the manual response generator GUI to respond to a repair request.
- Short-Code Request—Detailed instructions with explanations of how to use the manual response generator GUI to respond to a request that is in the form of a code rather than an XML document.
- Command-Line Interface—Usage with examples for manual response generator command-line interface.

The manual response generator is installed as part of the FlexNet Publisher Licensing Toolkit installation. It can be installed (copied) and run on any other system: it does not have to be run from the installation directory. To install it on another system, copy the executable and the library it uses and create the required subdirectories for the XML documents. For details, see XML Documents Required.

Note that although single-action transactions are deprecated in favor of composite transaction, support for them is provided for backward compatibility.

XML Documents Required

In addition to the request XML the manual response generator requires XML documents to perform the following functions:

- **Create the response XML**—create response files you must create based on schemas provided in the toolkit.
- **Generate signatures in the XML responses**—key files you must generate.
- **Validate XML document structure**—schema files provided in the FlexNet Publisher Licensing Toolkit.
- **Create of response codes**—ASR files you must generate.

These documents must be present in specific locations so that the GUI version of the manual response generator available on Windows can locate and use them—the default location is `<platform_dir>/publisher/xml` and subfolders. To view the XML documents that are available to the generator click the appropriate tab as follows:

- **Create Response Files**—Response Data
- **Key Files**—Key Data. Key files are required for composite transactions.
- **Schema Files**—Schema Data
- **ASR Files**—ASR Data

Response Files

The response files contain data required to generate a response to a request. There is a schema for each response file as follows:

- **Composite Response**—`W7_CreateCompositeResponseParameter.xsd`
- **Single-Action Activation Response**—`W1_CreateActivationResponseParameter.xsd`
- **Single-Action Return Response**—`W2_CreateReturnResponseParameter.xsd`
- **Single-Action Repair Response**—`W3_CreateRepairResponseParameter.xsd`
- **Single-Action Config Response (Client)**—`W4C_CreateConfigResponseClientParameter.xsd`
- **Single-Action Config Response (Server)**—`W4S_CreateConfigResponseServerParameter.xsd`
- **Single-Action Failure Response**—`W5_CreateFailureResponseParameter.xsd`
- **Short-Code Response**—`W6_CreateResponseCodeParameter.xsd`

All these schemas are available in the toolkit and are documented in Appendix A, “XML Schemas.” Schemas may exist in several versions as changes are made to them to update licensing functionality. For detailed information, see Identifying Schema Versions in the appendix. The Windows manual response generator displays the version of the schema that has been used to create a response file when the response files are placed in the correct version directories.

Sample response XML files are provided in the toolkit under `<platform_dir>/publisher/xml/Version*`.

You must create all the response files that you require using the information given in the relevant section of the appendix. It is recommended that you name the response files based on the information they contain. This makes it easier for the user of the manual response generator to select the correct response file.

For example, for product A, which can be licensed directly to a FlexEnabled client or via a license server, there could be the following activation response files:

- Product A permanent license client TID255
- Product A permanent licenses server TID50

Place the response files that you have created in the `xml` sub folder you have created for your Key Files and in the correct version sub directory, for example, "Version1". A list of the response files in the `xml` folder is displayed on the generator's **Response Data** tab. The response files are displayed grouped by type as follows:

- Composite Response
- Single-Action Activation Response
- Single-Action Return Response
- Single-Action Repair Response
- Single-Action Config Response (Client)
- Single-Action Config Response (Server)
- Single-Action Failure Response
- Single-Action Short-Code Response

Activation Response—fulfillmentType

Activation response files contain the fulfillment record that is added to trusted storage and provides the license rights. The fulfillment record can contain details of the type of transaction that was used to add it to trusted storage. This is indicated by the value of `fulfillmentType`. If the Manual Response Generator is being used to simulate a FlexNet Operations server, then the value of `fulfillmentType` should be set to six. For further details, see I_FulfillmentRecord.xsd in Appendix A, "XML Schemas."

Key Files

The key files, required to generate signatures in some responses, must be created using the `tcencrypt` utility. For instructions, see Chapter 13, "Working with the Software Publisher Utilities," specifically tcencrypt. Create a subfolder below the generator folder named `xml` and copy the key files you have generated to this folder.

The generator selects the key file(s) that match the trusted ID(s) in the response, so a key file is required for each Trusted Configuration that you want to use.

Responses that contain only Trusted Configurations do not require a key file.

It is recommended that you include the trusted ID in the name of the key file so that it is easier to identify when the appropriate key file is not available. A list of the key files in the xml folder is displayed on the generator's **Key Data** tab in the **Key files** pane. The following shows the list of key files available in the toolkit. `DemoKeyout.xml` is the key file required when using the manual response generator for the usage cases demonstrated in the *Getting Started with Trusted Storage-Based Licensing*.

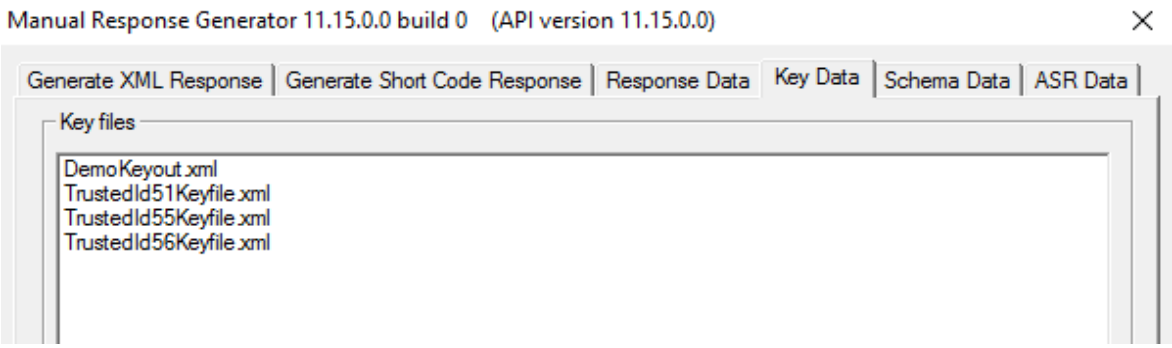


Figure 14-1: Manual Response Generator showing key files in toolkit

Note • *To produce a key file that supports request and response short codes, a* `TransactionKeyInfo` *element must exist in the input document for tcencrypt set to* `Short_2`*. See O_ConfigPhaseOne.xsd in Appendix A, "XML Schemas," for details.*

Schema Files

The schema files, required for validation of XML document structure, are provided in the toolkit. After installing your manual response generator, create a subfolder, "schemas," below the install folder. Copy all the schema files from the toolkit to the schemas folder you have created. Ensure that any subdirectories for schema versions are created and schemas are copied to the correct directories.

The top-level schemas are displayed on the generator's **Schema Data** tab. Schemas that are referenced by the top-level schemas (included in other schemas) are not displayed in the Schema Files pane. However, they are required for validation to succeed.

An error message similar to the following is output by the generator when validation fails because a schema file is missing:

```
Error code: -1072898035
Reason: The element 'FulfillmentRecord' is used but not declared in the DTD/Schema.
```

ASR Files

ASR files are required to use the Manual Response Generator for short-code transactions.

A copy of the ASR file used on the client system that initiated a short-code transaction is required to process a request code and create a response code. Place the required ASR files in the `xml` sub folder you have created for your Key Files.

Note • *Short-code aliases must be unique: if two ASRs with the same short-code alias are placed in the* `xml` *sub folder, only the first file that is read will be used.*

For information about the format of ASR files, see B_ActivationSpecificationRecord.xsd in Appendix A, "XML Schemas."

Using the Manual Response Generator

The following is an outline of typical steps in generating an XML response using the Windows manual response generator, `responsegenUI.exe`. Read this section if you want to get a quick overview of how to use the manual response generator.

You can also find more information about using the manual response generator in *Getting Started with Trusted Storage-Based Licensing*. This document includes a demonstration of activation using example files provided within the toolkit.

Task ***To generate an XML responsing using the response generator***

1. Run the manual response generator and click the **Generate XML Response** tab.

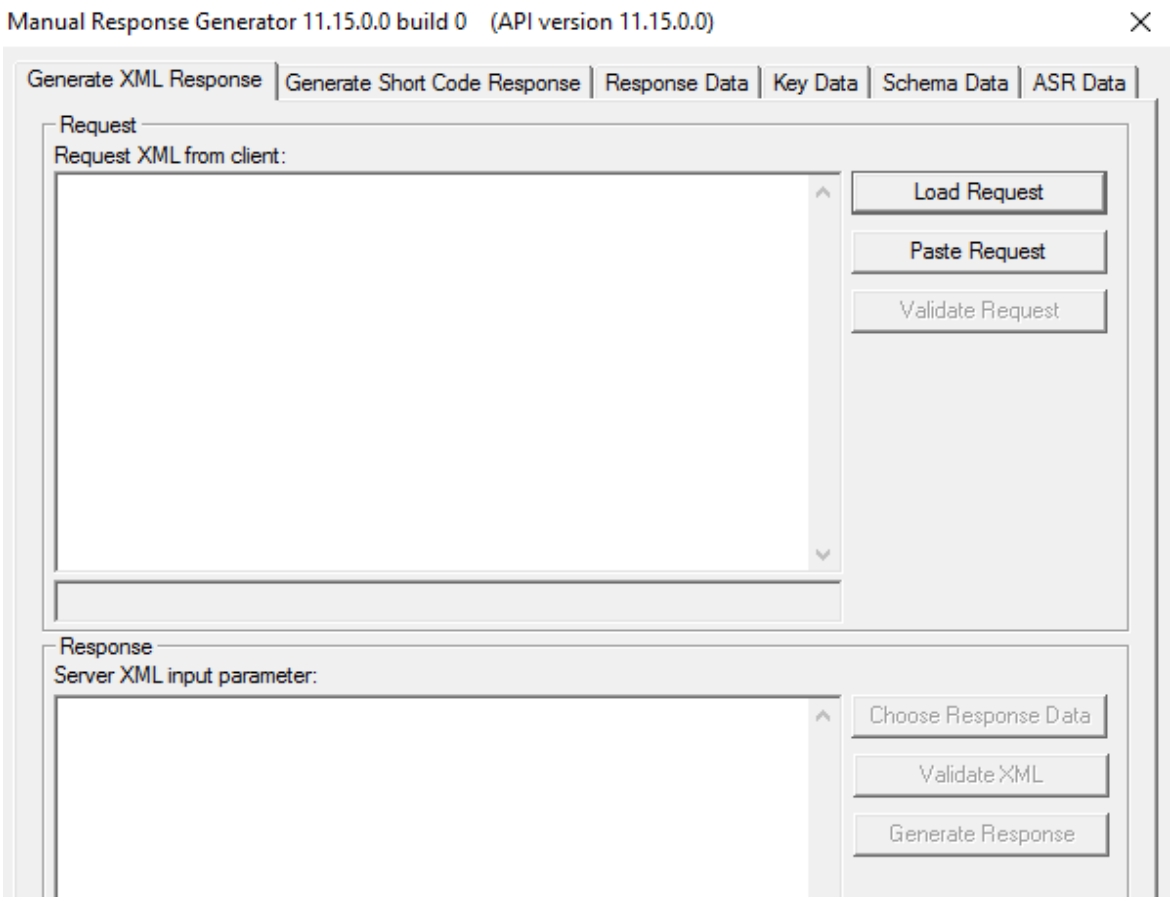


Figure 14-2: Manual Response Generator

2. Click **Load Request** and navigate to the request XML to load it. Alternatively, copy the request from a file and click **Paste Request** to paste it into the **Request XML from client** panel.

3. Click **Validate Request** to validate the XML against the appropriate schema and check the hash value (ensures that the contents of the request have not been tampered with).

Note • *Because of a historical error in schema version 5 and 6, a warning is displayed that the XML request version x schema validation was skipped. However, the request hash is verified.*

4. Read the request XML and determine what response is required. Click **Choose Response Data** and click the **Response Type** required. A list of create response XML files matching the selected Response Type is displayed.

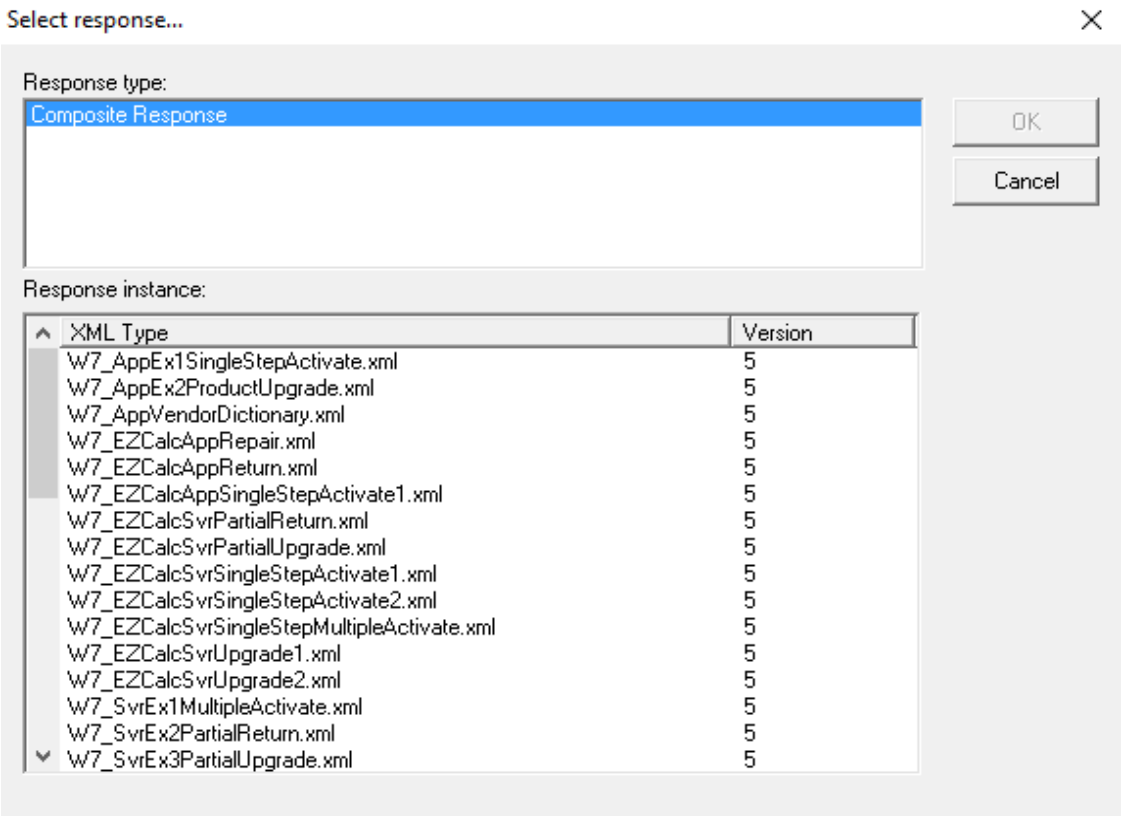


Figure 14-3: Manual Response Generator showing create response XML files

5. In **Response instance**, click the required response. Choose a response instance with a Version Number equal or less than that reported for the requestor in the request.

6. If further data must be added to the response, then additional text boxes are displayed at the bottom of the **Select response** dialog. Add any required additional data and click **OK**.

7. Click **Validate XML** to validate the create response XML document against the appropriate schema.

8. Click **Generate Response**. The response XML is displayed in the **Response** dialog.

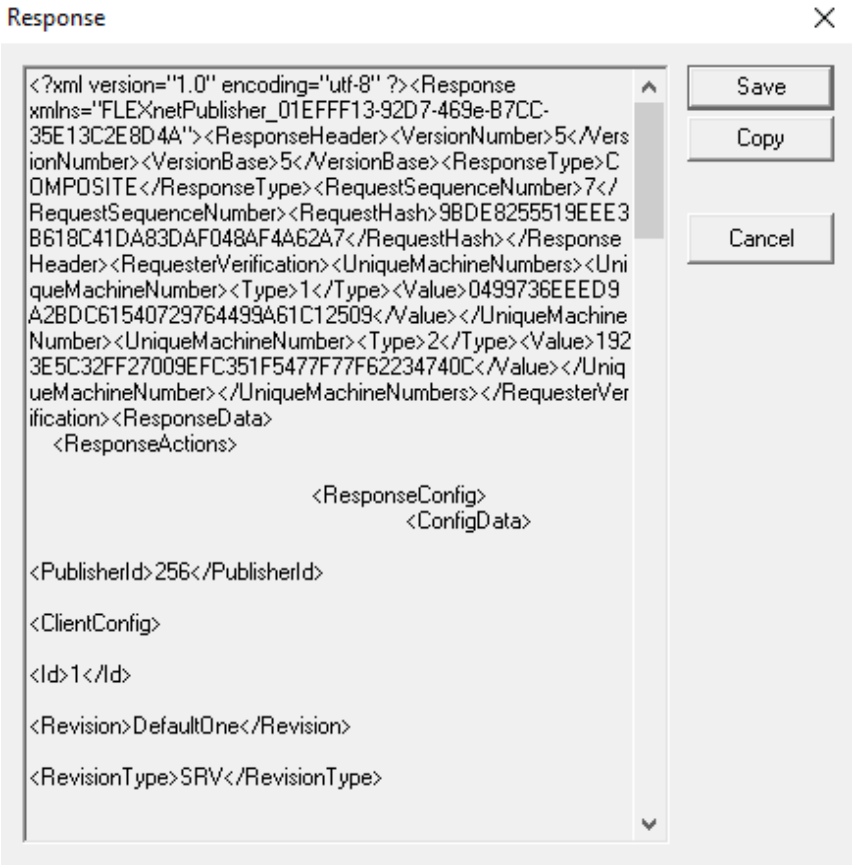


Figure 14-4: Response dialog showing response XML

9. Click **Save** to write the response to a file, or click **Copy** and then paste the response into a response document.
10. Click **Cancel** to return to the main dialog.
11. To generate a response from another request, click **Clear All**. The existing transaction data is deleted. Continue with generating the response from Step 2 onwards in this procedure.
12. To exit from the manual response generator, click either **OK** or **Cancel**.

Composite Request

This section gives detailed instructions for processing a composite request.

Task ***To process a composite request***

1. Run the manual response generator and click the **Generate XML Response** tab.
2. Click **Load Request** to read the request from a file. Alternatively, copy the composite request from the source document and click **Paste Request** to paste the request into the **Request XML from client** pane.

   A brief description of the request is displayed below the **Request XML from client** pane.

3. Click **Validate Request** to validate the composite request against `X_CompositeRequest.xsd` and check the hash value (this ensures that the contents of the request have not been tampered with).

   Note • *Because of a historical error in schema version 5 and 6, a warning is displayed that the XML request version x schema validation was skipped. However, the request hash is verified.*

4. Read the request and determine which response is required.
5. Click **Choose Response Data**.

   Composite requests contain one or more request actions, each of which requires a corresponding response action. Identify the appropriate response parameter file. In the **Response instance** panel, a list of XML create response documents matching the selected **Response Type** is displayed.

6. In **Response instance**, click the required response.
7. Click **OK**.
8. Click **Validate XML** to validate the create response XML document against the appropriate schema.
9. Click **Generate Response**.
10. The response XML is displayed in the **Response** dialog. Click **Save** to write the response to a file, or click **Copy** and then paste the response into a response document.

Activation Request

This section gives detailed instructions for processing an activation request.

Task ***To process an activation request***

1. Run the manual response generator and click the **Generate XML Response** tab.
2. Click **Load Request** to read the request from a file. Alternatively, copy the composite request from the source document and click **Paste Request** to paste the request into the **Request XML from client** pane.

   A brief description of the request is displayed below the **Request XML from client** pane.

3. Click **Validate Request** to validate the activation request against `C_ActivationRequest.xsd` and check the hash value (ensures that the contents of the request have not been tampered with).
4. Read the activation request and determine the response that is required.
   - If the validation of the activation request failed, respond with a Failure Response and include an appropriate error code.
   - Determine from the EntitlementData within the activation request which response file is required.
   - Determine whether the required trusted sections are available on the requesting system.
   - The **Requestor** entry indicates the highest version of the response that the requesting module can process. The response selected must have a version equal to or less than the version reported for the requestor. If a response is sent that is a later version, it will be rejected.

- Optionally, additional information may be included in the VendorDictionary element. This data is added by the application activation utility using **flxActAppActivationVendorDataSet**.

5. Click **Choose Response Data** and click the **Response Type** required.
   - **Activation Response**—lists the names of all the activation response files.
   - **Config Response (Client or Server)**—Lists the names of all the configuration response files. If the activation request is from a client, then only client configuration responses are listed. If the activation request is from a server, then only server configuration responses are listed.
   - **Failure Response**—lists the names of all the failure response files.

   Note • *If the response files have informative names, you can combine step 5 with step 6. Click **Choose Response** and then move the **Select Response** window so that you can view the Request XML in the main generator window. Then you can determine what response is required by using the **Select Response** window to display all the available response files.*

   A list of XML create response documents matching the selected **Response Type** is displayed.
6. In **Response instance**, click the required response.
7. If further data must be added to the response, then additional text boxes are displayed at the bottom of the **Select response** dialog.
   - **Activation Response**—edit the Fulfillment ID. Fulfillment identities should be unique in your system to enable tracking of activation transactions.
   - **Failure Response**—edit the Error Code and Error Reason as required. The values displayed are the values in the selected response file. The edited values are written into the response.

   Add any required additional data and click **OK**.
8. Click **Validate XML** to validate the create response XML document against the appropriate schema.
9. Click **Generate Response**.
10. The response XML is displayed in the **Response** dialog. Click **Save** to write the response to a file, or click **Copy** and then paste the response into a response document.

EntitlementData

This section of the activation request contains information that enables you to identify the customer's license entitlement (EntitlementId). This information enables you to select the required response file.

For an automated transaction in an operations server, your system may perform a validation of the license entitlement data (entitlement serial number). When you are using the manual response generator, any validation of the entitlement data must be manually initiated and a suitable response selected.

Trusted Sections on Requesting System

Details of the trusted sections available in trusted storage on the requesting system are included in the activation request. Each trusted section can have different binding and anchoring configured. This can be used to vary the binding and anchoring depending on the product and the license model in use. It is usual to specify a higher level of binding for license servers. When the trusted section required for the response is not already available on the requesting system, a Config Response must be sent.

The following information is available in the activation request in the ConfigData section:

- Whether the requesting system is a client or a server (IsServer true/false). The manual response generator automatically displays only client configuration responses, if the request is from a client, and server configuration responses, if the request is from a server.
- A list of existing trusted sections, each identified by a trusted ID. The following information is given for each trusted section under TrustedHostInformation:
  - TrustedId
  - Revision
  - MachineIdentifier
  - Status

A configuration response is required if the trusted section required for the activation, specified in the activation response, differs in any way from the trusted sections available on the requesting system. Thus, a configuration response may be required for any of the following reasons:

- The trusted ID does not exist on the requesting system. The trusted section must be created before a fulfillment record can be written to it.
- The revision number of the trusted ID on the requesting system does not match the revision number in the activation response file. The binding and/ or anchoring configuration for this trusted section requires updating.

Return Request

This section gives detailed instructions for processing a return request.

Task ***To process a return request***

1. Run the manual response generator and click the **Generate XML Response** tab.

   Click **Load Request** to read the request from a file. Alternatively, copy the composite request from the source document and click **Paste Request** to paste the request into the **Request XML from client** pane.

   A brief description of the request is displayed below the **Request XML from client** pane.

2. Click **Validate Request** to validate the return request against `E_ReturnRequest.xsd` and check the hash value (ensures that the contents of the request have not been tampered with).

3. Read the return request and determine what response is required.

a. If the validation of the return request failed, respond with a Failure Response and include an appropriate error code.

b. Determine which response file is required.

c. The **Requestor** entry indicates the highest version of the response that the requesting module can process. The response selected must have a version equal to or less than the version reported for the requestor. If a response is sent that is a later version, it will be rejected.

Note • *You can reply to a return request with a configuration response but this would be unusual.*

4. Click **Choose Response** and click the **Response Type** required.

   - **Return Response**—lists the names of all the return response files.
   - **Config Response (Client or Server)**—lists the names of all the configuration response files. If the return request is from a client, then only client configuration responses are listed. If the return request is from a server, then only server configuration responses are listed.
   - **Failure Response**—lists the names of all the failure response files.

   A list of XML create response documents matching the selected **Response Type** is displayed.

5. In **Response instance**, click the required response.

6. If further data must be added to the response, then additional text boxes are displayed at the bottom of the **Select response** dialog.

   - **Failure Response**—edit the Error Code and Error Reason as required. The values displayed are the values in the selected response file. The edited values are written into the response.

   Add any required additional data and click **OK**.

7. Click **Validate XML** to validate the create response XML document against the appropriate schema.

8. Click **Generate Response**.

9. The response XML is displayed in the **Response** dialog. Click **Save** to write the response to a file, or click **Copy** and then paste the response into a response document.

Repair Request

This section gives instructions for processing a repair request.

Task ***To process a repair request***

1. Run the manual response generator and click the **Generate XML Response** tab.

2. Click **Load Request** to read the request from a file. Alternatively, copy the composite request from the source document and click **Paste Request** to paste the request into the **Request XML from client** pane.

   A brief description of the request is displayed below the **Request XML from client** pane.

3. Click **Validate Request** to validate the repair request against `G_RepairRequest.xsd` and check the hash value (ensures that the contents of the request have not been tampered with).
4. Read the repair request and determine what response is required.
   a. If the validation of the repair request failed, you should respond with a Failure Response and include an appropriate error code.
   b. Determine from the data within the repair request whether the repair should be allowed. See Analyzing a Repair Request for an outline of the information available in the repair request.
   c. Establish rules for issuing repairs based on an assessment of whether the break in trusted storage is likely to be due to an attempt to circumvent the system.
   d. The **Requestor** entry indicates the highest version of the response that the requesting module can process. The response selected must have a version equal to or less than the version reported for the requestor. If a response is sent that is a later version, it will be rejected.
5. Click **Choose Response** and click the **Response Type** required.
   - **Repair Response**—lists the names of all the repair response files.
   - **Config Response (Client or Server)**—lists the names of all the configuration response files. If the repair request is from a client, then only client configuration responses are listed. If the repair request is from a server, then only server configuration responses are listed.
   - **Failure Response**—lists the names of all the failure response files.

   Note • *If the response files have informative names, you can combine Step 5 with Step 6. Click **Choose Response** and then move the **Select Response** window so that you can view the Request XML in the main generator window. Then you can determine what response is required by using the **Select Response** window to display all the available response files.*

   A list of XML create response documents matching the selected **Response Type** is displayed.
6. In **Response instance**, click the required response.
7. If further data must be added to the response, then additional text boxes are displayed at the bottom of the **Select response** dialog.
   - **Failure Response**—edit the Error Code and Error Reason as required. The values displayed are the values in the selected response file. The edited values are written into the response.

   Add any required additional data and click **OK**.
8. Click **Validate XML** to validate the create response XML document against the appropriate schema.
9. Click **Generate Response**.
10. The response XML is displayed in the **Response** dialog. Click **Save** to write the response to a file, or click **Copy** and then paste the response into a response document.

Analyzing a Repair Request

The security of fulfillment records within trusted storage is ensured in three ways:

- **Anchors**—Record the deletion and restoration of a trusted storage logical file.

- **Binding**—Locks a trusted section to the system on which it is located.
- **Time stamp**—Records when the fulfillment record was last accessed.

Fulfillment records can be *trusted* or *untrusted* based on whether the security of the trusted section where they are located is intact. An untrusted fulfillment record cannot be used and must be repaired.

Within the detailed request XML the following elements may be useful in deciding whether to allow a repair:

- **<TrustFlags>**—The status of the trust flags. The trust flags are identified as the elements <Anchoring>, <Binding> and <Time>. A value of 0 (zero) indicates untrusted. Potentially more than one trust flag can be untrusted.
- **<BreakInfo>**—Detailed information on why the repair is needed when this is an anchor or a binding break.
  - **<Anchoring>**—Anchors that are broken, either due to the anchor having changed or being not available. See O_ConfigPhaseOne.xsd in Appendix A, "XML Schemas," for details of anchors.
  - **<Binding>**—Binding identities that caused the break. See O_ConfigPhaseOne.xsd in Appendix A, "XML Schemas," for details of binding identities.
- **<TSSN>**—The Trusted Storage Serial Number. This is a generated value that is unique for a given trusted-storage file. This value is the same for all repair requests from a specific system provided that trusted storage has not been recreated. If trusted storage is deleted and recreated, a new TSSN is generated. The `<OriginalMachineIdentifier>` element does not identify a specific machine because it is based on the binding for a specific trusted section (each trusted section has a different machine identifier). Also when a binding break occurs, the machine identifier is recalculated; the current value is given in the repair request as the `<MachineIdentifier>` element within `<TrustedHostInformation>`.

When you paste the request XML, a brief description of the request is given below the request XML.This indicates whether the repair request is from a FlexEnabled client (`Requestor: client`) or from a license server (`Requestor: Server`). This enables you to implement different repair rules for requests from license servers where repairing a single fulfillment record potentially allows many copies of a FlexEnabled application to be licensed.

Short-Code Request

Manual and programmatic activation transactions use messages formatted as XML. These messages can be transmitted using either integrated communication (without end user intervention) or as an XML file (this may require end user intervention). When integrated communication is not an option (for example, the end user does not want to use it) or if the XML files are too large, the FlexNet Publisher Licensing Toolkit includes the option to perform activation transactions using request and response codes.

These transactions are referred to as *short-code transactions* because they use coded requests and responses. These requests and responses can be referred to as *short-code requests* and *short-code responses*. Typically this type of transaction is used for telephone transactions either with a customer service representative or using automated touch tone telephony.

Note • *Processing of short-code requests requires that ASR files are available to the manual response generator. For details, see Response Files.*

Task ***To use the manual response generator to determine the short-code transaction request type***

1. Run the manual response generator and click the **Generate Short Code Response** tab.
2. Copy the request code from the source document.
3. Click **Paste Request** to paste the request into the **Request Code from client** pane.
4. Click **Validate Request** to validate the request code. If the code is invalid, then the code cannot be used to generate a response. If the code is valid, the short-code alias in the request code is displayed.

   This identifies the ASR that is in use for this transaction. The **ASR Data** tab lists the aliases and ASRs that are available.

   When the ASR is not available in the xml sub folder, the following error message is displayed: `Unsupported short-code alias`. You must obtain the correct ASR and place a copy in the xml sub folder to enable you to analyze the request code.
5. Click **Analyze Request**. The following information is displayed:
   - **Request Type**—defines the type of transaction request. Possible values are ACTIVATION, RETURN, or REPAIR.
   - **Request Items**—lists the request items in the request code. The request items and their format is defined by the overrides in the ASR used for this type of transaction.

A typical display is as shown:

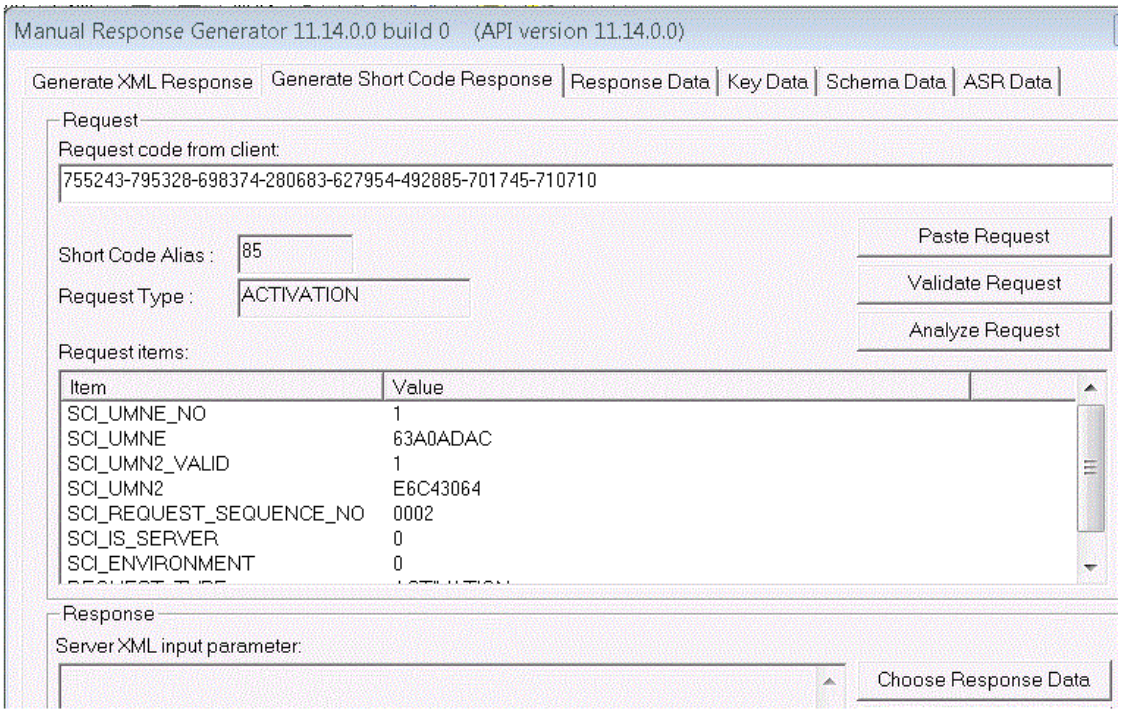


Figure 14-5: Manual Response Generator dialog

This example displays an activation request. It shows that the request is from a non-server machine in a physical environment and includes UMN1, UMN2 and the request sequence number.. The request items are configured in the ASR. for a description of request items, see ShortCodeData Element in Appendix A, “XML Schemas.”

Use the information in the display to determine what response to send as described in one of the following sections:

- Generate a Short-Code Activation Response
- Generate a Short-Code Return Response
- Generate a Short-Code Repair Response

Generate a Short-Code Activation Response

This section continues the instructions for generating a response to a short-code request when the request was for an activation. See Step 1 through Step 5 in Short-Code Request for the start of this process. The following steps continue that numbering.

1. Use the information in the **Request Items** and any additional information from the end user to determine the response required.

   The response to a short-code activation request can either be a short-code activation response or a short-code error response. When an activation response is sent, the fulfillment record in the ASR on the FlexEnabled client is written to trusted storage. When an error response is sent, the fulfillment record in the ASR on the FlexEnabled client is not written to trusted storage.

2. Click **Choose Response Data**. A list of XML create response documents for short-code transactions is displayed.
3. In **Response instance**, click the required response. Select either an activation response or an error response. For activations from one of the ShortCode2016*.asr ASRs choose W6_CreateResponseCodeParameterActivation2016.xml
4. Optionally click **Validate XML** to validate the create response XML document against the appropriate schema.
5. Click **Generate Response**, and the response code is displayed in the **Response** dialog.
6. Click **Save** to write the response to a file, or click **Copy** and then paste the response into a response document.

Generate a Short-Code Return Response

This section continues the instructions for generating a response to a short-code request when the request was for a return. See Step 1 through Step 5 in Short-Code Request for the start of this process. The following steps continue that numbering.

1. Use the information in the **Request Items** and any additional information from the end user to determine the response required.

   The response to a short-code return request can either be a short-code return response or a short-code error response. When a return response is sent, the fulfillment record on the FlexEnabled client is deleted from trusted storage. When an error response is sent, the fulfillment record on the FlexEnabled client remains in trusted storage.

2. Click **Choose Response Data**. A list of XML create response documents for short-code transactions is displayed.
3. In **Response instance**, click the required response. Select either a return response or an error response.
4. Optionally click **Validate XML** to validate the create response XML document against the appropriate schema.
5. Click **Generate Response**. The response code is displayed in the **Response** dialog.
6. Click **Save** to write the response to a file, or click **Copy** and then paste the response into a response document.

Generate a Short-Code Repair Response

This section continues the instructions for generating a response to a short-code request when the request was for a repair. See Step 1 through Step 5 in Short-Code Request for the start of this process. The following steps continue that numbering.

1. Use the information in the **Request Items** and any additional information from the end user to determine the response required.
2. Determine from the data within the repair request whether the repair should be allowed. The data included in the repair request are configured within the ASR for this short-code alias; therefore, no specific guidance can be given on how to interpret it. For a description of ASRs, see B_ActivationSpecificationRecord.xsd in Appendix A, "XML Schemas." However, some general information that may be relevant to the information in your repair request is given in Analyzing a Repair Request.

   You should establish rules for issuing repairs based on an assessment of whether the break in trusted storage is likely to be due to an attempt to circumvent the system.
3. Click **Choose Response Data**. A list of XML create response documents for short-code transactions is displayed
4. In **Response instance**, click the required response. Select either a repair response or an error response.
5. Optionally click **Validate XML** to validate the create response XML document against the appropriate schema.
6. Click **Generate Response**. The response code is displayed in the **Response** dialog.
7. Click **Save** to write the response to a file, or click **Copy** and then paste the response into a response document.

Command-Line Interface

This section describes the use of the manual response generator command-line interface. You should read the XML Documents Required section of this chapter before reading about the command-line interface. It describes the documents that are required to use the command-line interface.

The following sections contain information that may be required when using the command-line interface:

- EntitlementData—Activation requests
- Trusted Sections on Requesting System—Activation requests
- Analyzing a Repair Request—Repair requests

The command-line interface is provided by `responsegen.*`. This executable requires that the appropriate library file is available in the same directory. For example, on Windows, the files `responsegen.exe` and `responsegen.dll` are required.

Note • *Schema and response versions—ensure that a response based on the correct schema version is selected. The response will be rejected if it has a higher version than that supported by the requester. See Identifying Schema Versions in Appendix A, "XML Schemas," for more details.*

responsegen Syntax

The command-line syntax for responsegen is:

```
responsegen -i infile -o outfile -k keyfile
[-a activation_response_file]
[-r return_response_file]
[-p repair_response_file]
[-c config_response_file]
[-x][-f failure_response_file]
[-t shortcode_response_file] [-s asrfile]
[-z] [-h]
```

where:

Table 14-1 • Command-line Syntax for responsegen

Command-line Syntax	Description
-i *infile*	Transactions involve a request and a response. Use this option to specify an input file containing the request.
-o *outfile*	Transactions involve a request and a response. Use this option to specify an output file for the response to be written to. If -o is not specified, standard output is used.
-k *keyfile*	This option specifies the key file. (See Key Files.) It is not required when responsegen is generating a Config or a failure response, but if included will be ignored. Option can be repeated for each key file needed.
-a *activation_response_file*	Specifies an *activation* response and the appropriate activation response file. (See Response Files.)
-m *composite_activation_response file*	Specifies a *composite activation* response and the appropriate activation response file. (See Response Files.)
-r *return_response_file*	Specifies a *return* response and the appropriate return response file. (See Response Files.)
-p *repair_response_file*	Specifies a *repair* response and the appropriate repair response file. (See Response Files.)
-c *config_response_file*	Specifies a configuration response and the appropriate Config response file. (See Response Files.)
-x	This option forces a failure response to be returned regardless of the type of request and its validity. If this option is specified, -f must also be specified to give the required failure response file. This switch cannot be used with short codes: see -t for details.
-f *failure_response_file*	Specifies a failure response and the appropriate failure response file when using XML request and responses. (See Response Files.) This switch cannot be used with the -t switch.

Table 14-1 • Command-line Syntax for responsegen

Command-line Syntax	Description
`-t` *`shortcode_response_file`*	This option specifies the response file for short-code requests. `-x` and `-f` switches cannot be used with short codes: create an error response file for failed short-code transactions.
`-s` *`asrfile`*	Use this option to specify the `.asr` file corresponding to the short-code request being processed. (See Response Files.) This option is required when a short-code request is being processed. (See Short-Code Request.)
`-z`	Suppress on screen error messages: use for streamed output.
`-h`	Displays usage information.
`-m` *`composite_response_file`*	Use this option to specify a response file from a composite transaction.

Note *• Not all combinations of options are available or provide useful output. To ensure correct output, use only one option that specifies a response file. See Automated Use of responsegen for information on combining options that specify a response file in a single command.*

responsegen Errors

The following table lists error messages output to screen by responsegen.

Table 14-2 • responsegen Error Messages

Error Text	Description
Key file not specified	This command requires that a key file is specified using the `-k` option
Can't read key file	Unable to open key file.
Can't find TID in key file	Invalid or corrupt key file or the keyfile option specifies the wrong file.
Failure response file not specified	Invalid command: included `-f` without a failure response file.
Can't read failure response file	Invalid failure response file.
Generate response error	Failed to generate response.
Configuration response file not specified	Configuration option, `-c`, and configuration response file required in the command.
Can't read configuration response file	Unable to open configuration response file.

Table 14-2 • responsegen Error Messages

Error Text	Description
Activation response file not specified	Activation option, `-a`, and activation response file required in the command.
Can't read activation response file	Unable to open activation response file.
keyfile id does not match activation response XML tid	Unable to generate an activation response because the trusted ID specified in the key file does not match the trusted ID specified in the activation response file.
Return response file not specified	Return option, `-r`, and return response file required in the command.
Can't read return response file	Unable to open return response file.
Repair response file not specified	Repair option, `-p`, and repair response file required in the command.
Can't read repair response file	Unable to open repair response file.
Empty request code	No data in input file when processing short-code request.
Shortcode ASR file not specified	ASR option, `-s`, and ASR file required in the command.
Unable to read shortcode ASR file	Unable to open ASR file specified with `-s` option.
Empty ASR data	No data in ASR file specified with `-s` option.
Shortcode response file not specified	Short-code option, `-t`, and short-code response file required.
Unable to read shortcode response file	Unable to open short -code response file specified with `-t` option.
Empty response data	No data in short-code response file specified with `-t` option.
TID does not match keyfile	Unable to generate a response because the trusted ID specified in the key file does not match the trusted ID specified in the return or response file.
Invalid command-line syntax	See previous section for details of command-line syntax.
Unknown command-line option	See previous section for details of command-line syntax.

Operator Using responsegen

When an operator is using responsegen, the recommendation is to include only one response switch and response file. The operator:

1. Examines the request file
2. Determines the type of request

3. Selects the appropriate response file
4. Creates a responsegen command that specifies one response file.

For example when processing an activation request that requires a config response the following could be input:

```
responsegen -i request.xml -o config_response.xml \
            -c CreateConfigResponseServerParameter.xml
```

Where `request.xml` is a file containing the activation request, `CreateConfigResponseServerParameter.xml` is the config response file. This generates a config response in `config_response.xml` that is processed by the requesting activation utility. Then, a further activation request is sent by the activation utility. An activation response is required to this request, for example:

```
responsegen -i request.xml -o act_response.xml -k KeyFile.xml\
            -a CreateActivationResponseParameter.xml
```

Where `request.xml` is a file containing the activation request, `CreateActivationResponseParameter.xml` is the activation response file, and `KeyFile.xml` is the matching key file for the trusted ID specified in the config response. This generates an activation response in `act_response.xml` that is processed by the requesting activation utility.

Automated Use of responsegen

You may want to automate the use of responsegen. responsegen contains some logic so that an “appropriate” response can be generated. The following sections describe some ways in which responsegen can be used in an automated environment. When using responsegen for streamed output use the `-z` option.

Config Response

There is limited support for the automatic sending of a config response. The following options can be combined in one command:

```
responsegen -i infile -o outfile -k keyfile
-a activation_response_file
-r return_response_file
-p repair_response_file
-c config_response_file
```

This will result in a configuration file being generated only if there is no trusted storage on the requesting system and thus the activation request contains no trusted Ids.

Automated Selection of Response Type

For XML requests, responsegen will match the response file to the request type. So the following command creates a matching response:

```
responsegen -i infile -o outfile -k keyfile
-a activation_response_file
-r return_response_file
-p repair_response_file
```

This generates:

- An activation response file using the `activation_response_file` if the `infile` contains an activation request

- A return response file using the `return_response_file` if the `infile` contains a return request
- A repair response file using the `repair_response_file` if the `infile` contains a repair request

Automated Failure Response

There is limited support for the automatic sending of a failure response. The `-f` option can be used in a command that includes any combination of the following options:

```
-a activation_response_file
-r return_response_file
-p repair_response_file
```

This generates a failure response file when responsegen was unable to process the request file.

Demonstration Using responsegen

The scenarios described in this section include both server-side and client-side operations. In these examples, you use the same system for both types of operations. However, an enterprise environment typically has one, or more, dedicated license servers distinct from FlexEnabled clients. These examples use the example application activation utility, `appcomptranutil.*`, and example response files used in the *Getting Started with Trusted Storage-Based Licensing*. More demonstrations are provided in the *Getting Started with Trusted Storage-Based Licensing*.

The examples are based on Windows platforms, but can equally be run on UNIX platforms.

Short-Code Activation

The following example demonstrates using the manual response generator for a short-code activation transaction.

Task ***To perform a short-code activation***

1. From the `<platform_dir>` directory, type in the following to generate an activation request.

   `appcomptranutil.exe -sc publisher\xml\ShortCode2016Medium.asr -activate -brief > in.txt`

Syntax	Description
`appcomptranutil`	The application activation utility is invoked.
`-sc publisher\xml\ShortCode2016Medium.asr`	Specifies a short-code transaction and the appropriate `.asr` file from the `<platform_dir>\publisher\xml` directory. `Medium` indicates medium security.
`-activate`	Generates an activation request.
`-brief`	Suppresses messages so that only the code is written.
`in.txt`	The file name to receive the activation short code.

   The activation request is saved in file `in.txt`.

2. (Optional) Enter the following to view the activation request code:

```
type in.txt
```

The request code looks similar to the following:

```
781159-132828-438324-950634-979876-106445-376657-398715
```

3. From the `<platform_dir>\publisher` directory, use `responsegen` to generate a short-code response.

```
responsegen -i ..\in.txt -o ..\out.txt -s xml\ShortCode2016Medium.asr -t
xml\Version1\W6_CreateResponseCodeParameterActivation2016.xml -k xml\TrustedId55Keyfile.xml
```

Syntax	Description
`responsegen`	`responsegen` is invoked.
`-i ..\in.txt`	Reads the request text file, `in.txt`, from the `dist` directory.
`-o ..\out.txt`	Specifies an output text file, `out.txt`, to be placed in the `dist` directory.
`-s xml\ShortCode2016Medium.asr`	Specifies a short-code transaction and the appropriate ASR file from the `<platform_dir>\publisher\xml` directory. `Medium` indicates medium security.
`-t xml\Version1\ W6_CreateResponseCodeParameterActivation2016.xml`	Specifies a short-code response file for activations.
`-k xml\Trusted55Keyfile.xml`	Specifies the key file for medium strength short codes from the `<platform_dir>\publisher\xml` directory.

4. From the `<platform_dir>` directory, process the return response.

```
appcomptranutil -shortcoderesponse out.txt
```

Syntax	Description
`appcomptranutil`	The short-code response is processed by the application activation utility from the `<platform_dir>` directory.
`-shortcoderesponse out.txt`	Processes the response file, `<platform_dir>\out.txt`.

You should see a response similar to the following:

```
Processing response: 455976-326180-807501-999982-850286-428479-264703-943367-434780-429547-131530-
463747-395138-206711

Short code ACTIVATION response processed, fulfillment created
      Entitlement Id: ENTL-EZCALC-SC2016-1
      Fulfillment Id: FID-EZCALC-SC2016-1
          Product Id: NAME=EZCALC-A;VERSION=1.0
          Expiration: 2-feb-2018
```

Short-Code Return Request

The following example demonstrates returning an activated license from client-side trusted storage back to the server-side trusted storage using the manual response generator's command-line interface.

Task ***To return an activated license to trusted storage***

1. From the `<platform_dir>` directory, generate a return request.

   `appcomptranutil.exe -sc publisher\xml\ShortCode2016Medium.asr -return <fulfillmentId> -brief > in.txt`

Syntax	Description
`appcomptranutil`	The application activation utility is invoked.
`-sc publisher\xml\ShortCode2016Medium.asr`	Specifies a short-code transaction and the appropriate `.asr` file from the `<platform_dir>\publisher\xml` directory. `Medium` indicates medium security.
`-return` *`<fulfillmentId>`*	Specifies a return with the fulfillment ID as an argument. Use the fulfillment ID from the previous exercise, Short-Code Activation, `FID-EZCALC-SC2016-1`.
`-brief`	Suppresses messages so that only the code is written to `in.txt`.
`in.txt`	Specifies that the request be written to `<platform_dir>\in.txt`.

   The return is written to `in.txt`.

2. From the `<platform_dir>\publisher` directory, run `responsegen` to generate a return response.

   `responsegen -i ..\in.txt -o ..\out.txt -s xml\ShortCode2016Medium.asr -t xml\Version1\W6_CreateResponseCodeParameterReturn.xml -k xml\TrustedId55Keyfile.xml`

Syntax	Description
`responsegen`	`responsegen` is invoked.
`-i ..\in.txt`	Reads the request XML file, `in.txt`, from the `<platform_dir>` directory.
`-o ..\out.txt`	Specifies an output XML file, `out.txt`, to be placed in the `<platform_dir>` directory.
`-s xml\ShortCode2016Medium.asr`	Specifies a short-code transaction and the appropriate ASR file from the `<platform_dir>\publisher\xml` directory. `Medium` indicates medium security.
`-t xml\Version1\W6_CreateResponseCodeParameterReturn.xml`	Specifies a short-code response file for returns.

Syntax	Description
`-k xml\TrustedId55Keyfile.xml`	Specifies an appropriate key file from the `<platform_dir>\publisher\xml` directory.

3. From the `<platform_dir>` directory, process the return response.

   `appcomptranutil -scr out.txt`

Syntax	Description
`appcomptranutil`	The return response is processed by the application activation utility.
`-scr out.txt`	Processes the response file, `out.txt`.

You should see output similar to the following:

```
Processing response: 181640-161655-923282-715141-232098-710924-582768-195004-979048-615233-127940-
976969-335092-174716

Short code RETURN response processed, fulfillment returned
      Entitlement Id: ENTL-EZCALC-SC2016-1
      Fulfillment Id: FID-EZCALC-SC2016-1
          Product Id: NAME=EZCALC-A;VERSION=1.0
          Expiration: 2-feb-2018
```

Using responsegen API

The following example demonstrates the use of the responsegen API to validate an XML request and generate a response.

Task ***To use the responsegen API***

1. The tool can be rebuilt using `makefile.act responsegenapi`. The responsegenapi.exe will be created in the `publisher` sub-directory.
2. The responsegenapi sample application demonstrates the use of the responsegen API to validate an XML request and generate a response. For details see the source file in the kit folder `examples\activation\responsegen\ResponseGenApi.c`
3. To use the response generation API sample application

   `Usage: responsegenapi <requestXmlFile> [<responseParameterFile> <keyFile>...]`

   ```
   The request is verified.

   If a <responseParameterFile> is specified, a response is generated and printed.
       For composite requests the parameters must conform to the W7 XML schema.
       For single action requests the parameters must conform to W1 (activation)
       W2 (return) W3 (repair) W4S/C (config server/non-server) W6 (failure)
       (see kit publisher/XML/Schemas folders)
       For the out-of-the-box kit the keyFile is publisher/xml/DemoKeyout.xml
   ```

4. To validate a request

   `<Platform>\publisher>responsegen>responsegenapi request.xml`

```
Request is valid.
```

5. To generate a response

```
<Platform>\publisher>responsegen>responsegenapi request.xml
xml\Version6\W7_AppEx1SingleStepActivate.xml xml\DemoKeyout.xml
```

Limitations

Support for short code and multiple key files (needed for composite transactions that target more than one trusted section) is currently not supported.

15

Defining Feature Definition Lines for a Fulfillment Record

This chapter is a reference for feature definition line syntax. A feature definition line contains the basic license rights information for a specific feature. It is part of a fulfillment record placed in trusted storage during activation.

Feature definition lines are placed in the `value` element for the `features` key element in a fulfillment record.

Additional information held in the fulfillment record overrides the expiration and count field in the feature definition lines. These lines have a similar syntax to the license file lines.

Important • *The information in this section assumes that you have a thorough understanding of license files, license-file syntax, and feature lines as described in the document titled Programming Reference for License File–Based Licensing. If you do not have this knowledge, read the information in that document before continuing. You might also need to generate cross-version license signatures with certain feature definition lines. This capability is described in detail in the document Programming Reference for License File–Based Licensing (see the section titled Cross-Version License Files).*

Feature Definition Line Syntax

```
FEATURE|INCREMENT feature vendor feat_version date count OVERDRAFT=1 \
                  [vendor_keywords] SIGN=sign
```

Where:

Table 15-1 • Feature Definition Line Syntax

Syntax	Description
`FEATURE | INCREMENT`	Required. ● For license rights specified with a single feature line, use FEATURE. Only the first FEATURE line for a given feature is processed by the vendor daemon. ● When multiple feature lines are needed to specify the license rights, as is the case when defining license pools, use INCREMENT for adding to existing license rights. ● An INCREMENT line can be used in place of a FEATURE line, as well as to incrementally add licenses to a prior FEATURE or INCREMENT line in the fulfillment record. See License Pools for further information.
feature	Required. The name given to the feature. Feature names may contain 30 characters or less. The supported set of characters for feature names are: lowercase alphabet (a-z), uppercase alphabet (A-Z), digits (0-9), and underscore (_). Letters in the feature name are case insensitive by default. If case sensitivity is desired, set the `LM_A_LICENSE_CASE_SENSITIVE` attribute in the FlexEnabled application. See the *Programming Reference for License File–Based Licensing* for further details.
vendor	Required. The vendor daemon that authenticates the feature line. This is your vendor name as registered with Flexera and is that used when configuring your publisher-specific toolkit. For concurrent licenses, specifies the vendor daemon that serves the feature.
feat_version	Required. The latest (highest-numbered) version of this `feature` that is supported in this fulfillment record. The version is specified with the format `x.y`, as a floating point number with a ten-character maximum.
date	Required. The date of the feature line. Unlike feature lines in license files, this date has no effect on feature expiration. The expiration date is specified by the `expiry` key element in the fulfillment record and overrides `date`. Thus all feature definition lines within a fulfillment record have the same expiry date. The expiration date of the feature in the format: `dd-mmm-yyyy`, for example, `31-dec-2020`.
count	Required. The number of licenses for this feature specified as an integer `> 0`. For licenses on license servers: this value is used as a multiplier with the `availConc` and `availHyb` key element counts in the fulfillment record. For more information on feature counts, see Chapter 2, "Transactions to a License Server," and Appendix A, "XML Schemas." For license rights on FlexEnabled clients: even though required, this count is ignored. All such licenses are considered uncounted.

Table 15-1 • Feature Definition Line Syntax

Syntax	Description
OVERDRAFT=*n*	Required. A number of additional licenses that your end user is allowed to use, specified as an integer > 0, in addition to the licenses they have purchased. This is useful if you want to allow your end users to not be denied service when in a temporary overdraft state. Usage above the licensed limit will be reported by the FlexNet Manager reporting tool. This value is used as a multiplier with the availConcOD and availHybOD overdraft key element counts in the fulfillment record. This keyword must be defined but is only used in the context of license rights held in the license server's trusted storage. To disallow overdrafts, set availConcOD and availHybOD to 0. For more information on overdraft counts and conditions, see Overdraft Conditions and the section FulfillmentDictionary Data in Server-side Trusted Storage (in Appendix A, XML Schemas).
vendor_keywords	Optional. See Vendor Keywords for a complete listing.
SIGN=	Required. It is the signature for this FEATURE line and is produced by the lmcrypt utility, by a publisher-defined utility that calls **lc_crypstr**, or by the asrgen utility. When using lmcrypt, put SIGN=0 at the end of each FEATURE line, and lmcrypt will replace the 0 with the correct signature. See *Programming Reference for License File–Based Licensing* for complete information about feature line signatures, the lmcrypt utility, and **lc_crypstr**. For information about the asrgen utility, see asrgen in Appendix A, "XML Schemas."

Comment Lines

It is a convention that comment lines begin with the character, "#." However, all lines not beginning with a license file keyword are ignored and are considered comment lines.

Line Continuation

Long lines are broken up by line the continuation character, "\."

Overdraft Conditions

You can determine if a feature is in an overdraft condition by calling lc_get_attr(job, LM_A_VD_FEATURE_INFO,...) from your FlexEnabled application. The returned structure has three members of interest related to overdraft: tot_lic_in_use, num_lic, and overdraft. An overdraft state is defined as tot_lic_in_use > (num_lic – overdraft). Now, if tot_lic_in_use=num_lic, no more licenses re available; both the normal allotment and the overdraft allotment have been consumed.

License Pools

INCREMENT lines form license groups, called *license pools*, based on the following fields:

- feature name
- version
- DUP_GROUP
- FLOAT_OK
- HOST_BASED
- PLATFORMS
- USER_BASED
- VENDOR_STRING (if configured by the publisher as a pooling component)

Note • *Licenses can also be pooled by entitlement ID. This is described in Pooling by Entitlement ID.*

If two lines differ by any of these fields, a new license pool is created in the vendor daemon, and this group is counted independently from other license pools with the same feature name. Pooling candidates for a given feature are collected from features in trusted storage and license files combined. A FEATURE line does not give an additional number of licenses, whereas an INCREMENT line always gives an additional number of licenses.

Consider the following example that demonstrates license pooling, using INCREMENT lines from license files:

```
INCREMENT f1 demo 2.0 31-dec-2020 1 SIGN=...
INCREMENT f1 demo 2.0 31-dec-2020 2 SIGN=...
```

- `INCREMENT`—the server adds up licenses for all lines for the same feature name forming a license pool for feature "`f1`." The resulting `count` in this example is `3` (`1` + **2**). The count for INCREMENT lines appearing in fulfillment records is affected by the count information in containing fulfillment record, see FulfillmentDictionary Data in Server-side Trusted Storage in Appendix A, "XML Schemas," for further information about counts in fulfillment records.
- The first `INCREMENT` line could be a `FEATURE` line and the behavior would be the same.

Now, in contrast, this next example shows separate pooling:

```
INCREMENT f1 demo 2.0 31-dec-2020 1 DUP_GROUP=UH SIGN=...
INCREMENT f1 demo 2.0 31-dec-2020 2 DUP_GROUP=D SIGN=...
```

- The server tracks these licenses independently, in separate pools, because the DUP_GROUP fields are different.
- Since a license pool is formed, this independent tracking *only* works with INCREMENT, not FEATURE, because the server only recognizes the first FEATURE line for a given feature name. Subsequent ones are ignored.

A single checkout request can not span multiple license pools. That is, if a checkout is requesting more licenses than are available in a single license pool, the request is denied. Consider the following example:

```
INCREMENT f1 demo 1.0 31-dec-2020 3 SIGN=2B8F621C172C
INCREMENT f1 demo 2.0 31-dec-2020 4 SIGN=2B9F124C142C
```

These lines form two separate license pools, one with three licenses for version 1.0 and one with four licenses for version 2.0. A request for five licenses for version 1.0 is denied because neither pool has five licenses.

The license checkout behavior is slightly different when feature definition lines are configured such that they create multiple license pools, but the DUP_GROUP configuration is the same. For example:

```
INCREMENT f1 demo 1.0 31-dec-2020 1 SIGN=...
INCREMENT f1 demo 1.0 31-dec-2020 2 SIGN=...
```

A checkout request searches all license pools for an existing checkout of the feature made using the same checkout request identifier (in this example, it would search for the same display name) and keyword criteria. If one is found, that license is used to satisfy the request. If not, a new license is consumed.

Differences between FEATURE and INCREMENT Lines

To illustrate the difference between FEATURE and INCREMENT, consider these feature lines:

```
INCREMENT f1 demo 1.0 31-dec-2020 5 ....
INCREMENT f1 demo 1.0 31-dec-2020 4 ....
```

Because these two lines are specified with `INCREMENT` keyword and are identical except for the `count`, a license pool is formed. Both lines are combined and the result is nine licenses for version 1.0.

Now consider:

```
FEATURE f1 demo 1.0 31-dec-2020 5 ....
FEATURE f1 demo 1.0 31-dec-2020 4 ....
```

Because these two lines are specified with the `FEATURE` keyword, a license pool is not formed and the second FEATURE line is ignored. The result is just five licenses for version 1.0.

Contrast with the following example:

```
INCREMENT f1 demo 2.0 31-dec-2020 5 ....
INCREMENT f1 demo 1.0 31-dec-2020 4....
```

Since the `feat_version` values are different, these two lines do not pool. The result is four licenses for version 1.0 *and* five licenses for version 2.0 and earlier being available, giving a total of nine licenses for "`f1`."

Pooling by Entitlement ID

Licenses can be pooled by their entitlement ID. This can be useful in scenarios where a license administrator needs to differentiate between features that have identical names but belong to different products.

To enable pooling by entitlement ID, the variable `ls_entitlement_based_pooling` in `lsvendor.c` must be set to 1. By default, pooling is enabled. For more information, see ls_entitlement_based_pooling.

The following diagrams illustrate the behavior when pooling by entitlement ID is disabled or enabled.

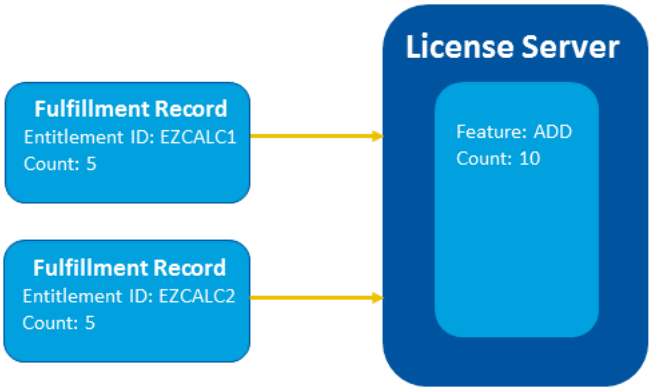


Figure 15-1: Pooling by entitlement ID is disabled.

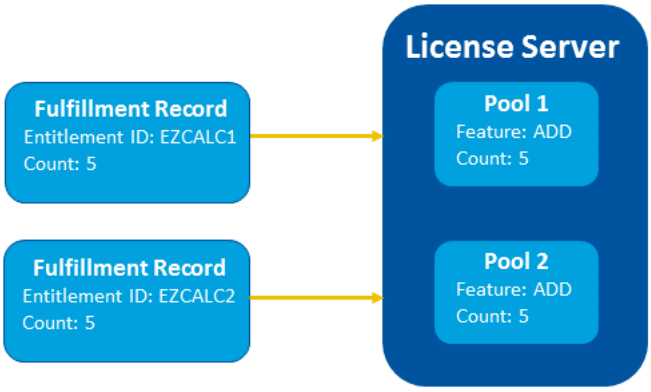


Figure 15-2: Pooling by entitlement ID is enabled.

Vendor Keywords

Here is a list of all available vendor keywords allowed in feature definition lines in fulfillment records. These keywords can appear in any order and must be placed between the `OVERDRAFT=` and `SIGN=` keywords.

- DUP_GROUP
- HOST_BASED
- ISSUED
- ISSUER
- MINIMUM
- NOTICE
- ONE_TS_OK
- SN
- sort
- START

- SUITE_DUP_GROUP
- TS_OK
- USER_BASED
- VENDOR_STRING

Character Limitations in Keyword Values

In places where a keyword value is a string surrounded with double quotes ("..."), the string can contain any characters except the following:

- Single or double quotes (within the surrounding double quotes)
- The backslash character sequence: \ (space+backslash+space)
- The double-backslash character sequence: \\ (space+backslash+backslash+space)

DUP_GROUP

`DUP_GROUP=NONE|SITE|[UHDV]`

The syntax is:

```
DUP_GROUP=NONE|SITE|[UHDV]
       U = DUP_USER
       H = DUP_HOST
       D = DUP_DISPLAY
        V = DUP_VENDOR_DEF
```

Any combination of `UHDV` is allowed, and the `DUP_MASK` is the `OR` of the combination. For example, `DUP_GROUP=UHD` means the duplicate grouping is (`DUP_USER | DUP_HOST | DUP_DISPLAY`), so a user on the same host and display will have additional uses of a feature and not consume additional licenses.

If `DUP_GROUP=` is specified in the feature line, this parameter overrides the *dup_group* parameter in the call to **lc_checkout**. If not specified in the license, the *dup_group* parameter from **lc_checkout** will be used.

HOST_BASED

`HOST_BASED[=n]`

If `HOST_BASED` appears, then licenses can be used only by hosts `INCLUDE`d for this feature in the options file. The purpose is to limit the use to a particular number of hosts, but allow the end user to determine which hosts. If `=n` is specified, then the number of hosts which can be `INCLUDE`d is limited to `n`. Otherwise, the limit is the `count` field of the feature line. If an `INCREMENT` appears where some licenses are `HOST_BASED` and some are not, the vendor daemon tracks these in separate license pools.

ISSUED

`ISSUED=`*`dd-mmm-yyyy`*

The date that the license was issued.

ISSUER

`ISSUER="string"`

The issuer, specified by `string`, of the license.

MINIMUM

`MINIMUM=n`

Optional field. If in **lc_checkout(**`...num_lic...`**)**, `num_lic` is less than n, then the server will checkout n licenses.

NOTICE

`NOTICE="..."`

Optional field. A field for intellectual property notices.

ONE_TS_OK

ONE_TS_OK

Optional field. Allows a user connected through a remote connection to another system (using Remote Desktop Connection or Terminal Services) to run the FlexEnabled application. This keyword is supported with node-locked uncounted licenses only. To allow the FlexEnabled application to run via the remote connection, you must define the license model as node-locked to the remote system and add the ONE_TS_OK keyword to the license.

The following example demonstrates this:

```
FEATURE f1 xyzd 1.00 1-jan-2020 uncounted HOSTID=ts_hostid \
        ONE_TS_OK SIGN=sign
FEATURE f2 xyzd 1.00 1-jan-2020 uncounted HOSTID=ts_hostid \
        ONE_TS_OK SIGN=sign
```

Without ONE_ TS_OK keyword on a feature definition line, the user is denied access to the FlexEnabled application. This keyword works similarly to TS_OK except, that it allows only one remote session to check out a feature at a time. It is supported on Windows operating systems.

Consider the scenario where a FlexEnabled application is installed on a system with its license locked to system's hostid. There are two features available, f1 and f2.

- The application runs successfully when accessed from that system directly, whether or not the feature definition line contains the ONE_TS_OK keyword. The license is checked out as though it was a node-locked license.
- If the feature definition line does not contain the ONE_TS_OK keyword and a user tries to run the application by connecting to the system through a remote connection, an error is returned.
- If the feature definition line contains the ONE_TS_OK keyword and one user runs the application through from the system directly while another user runs the application from a remote connection, both users are successful.
- If the feature definition line contains the ONE_TS_OK keyword and more than one user tries to run the application through a remote connection, the first user to check out f1 is successful. Subsequent requests for feature f1 return an error. However, another user can access the system through a remote connection and check out feature f2.

Important • *The limitation is on the number of times a feature is checked out. More than one remote session can run the application, but only one session can checkout a feature at a time.*

If both ONE_TS_OK and TS_OK are defined in a feature definition line, the ONE_TS_OK behavior takes precedence. This keyword does not limit users accessing the system using the Fast User Switching feature in Windows XP.

SN

`SN=serial_num`

Useful for differentiating otherwise identical `INCREMENT` lines. Its only use by FlexNet Publisher Licensing Toolkit is to be encrypted in the signature.

sort

`sort=nnn`

Optional keyword that sets the sort order. Features are sorted so that FEATURES and INCREMENTS with the lowest sort number are searched first for a matching license. Using this keyword in a feature definition line overrides the default sort setting for the sorting of features from trusted storage. For further details see *Programming Reference for License File–Based Licensing*.

START

`START=dd-mmm-yyyy`

Feature start date. If the license is used before this date, the checkout fails with the error `LM_TOOEARLY`.

SUITE_DUP_GROUP

`SUITE_DUP_GROUP=NONE|SITE|[UHDV]`

Similar to `DUP_GROUP`, but affects only the enabling `FEATURE` line for a package suite.

Note • *If* `SUITE_DUP_GROUP` *is not specified, the parent will have the same duplicate grouping as the components.*

`SUITE_DUP_GROUP` limits the total number of users of the package to the number of licenses, and allows the package to be shared among the users that have the `SUITE` checked out. For example

```
PACKAGE p ... COMPONENTS="A B C" OPTIONS=SUITE
FEATURE p ... 3 ... SUITE_DUP_GROUP=UHD
```

In this example, `SUITE_DUP_GROUP` limits the number of component users to three, and, separately, limits the number of uses of each component to three. This keyword is valid only with counted licenses.

Note • PACKAGE *lines can only be delivered in license files. For more information on* PACKAGE *lines, consult the Programming Reference for License File–Based Licensing.*

See Also
DUP_GROUP
Programming Reference for License File–Based Licensing for information on packages.

TS_OK

TS_OK

This feature is applicable only to feature definition lines held in the application's trusted storage. FlexNet Publisher Licensing Toolkit detects when the FlexEnabled application tries to checkout a license while it is running in a Windows Terminal Services guest session. To allow your FlexEnabled application to checkout a license while running in a Windows Terminal Services guest session, the license must include the TS_OK keyword. For example:

```
FEATURE f1 xyzd 1.00 1-jan-2020 1 OVERDRAFT=1 TS_OK SIGN=sign
```

Without the TS_OK keyword on a feature line, a FlexEnabled application running in a Windows Terminal Services guest session is denied use of the license.

Consider the scenario where a FlexEnabled application is installed on a Windows Terminal Services host system with its license.

- If the FlexEnabled application tries to checkout a license while running directly on the host system's session (that is, not in a guest system's session), the license is checked out as though it was a locally activated license.
- If the feature line does not contain the TS_OK keyword and the FlexEnabled application tries to checkout this license while running in a Windows Terminal Services guest session, an error is generated (License fails to checkout, -103).
- If the feature line contains the TS_OK keyword and the FlexEnabled application tries to checkout this license while running in one or more Windows Terminal Services guest sessions, the license is checked out and the application can run in each of the guest sessions. With TS_OK specified, FlexNet Publisher Licensing Toolkit does not limit the number of application instances that can checkout this license.

The TS_OK keyword is ignored if it appears in a feature line in the server's trusted storage.

This keyword applies to all virtual terminal technologies that register their sessions with Windows Terminal Services. Examples are Remote Desktop (formerly known as Windows Terminal Server) and Citrix® Metaframe®.

USER_BASED

USER_BASED[=*n*]

If USER_BASED appears, then licenses can only be used by users INCLUDEd for this feature in the options file. The purpose is to limit the use to a particular number of users, but allow the end user to determine which users. If =n is specified, then the number of users which can be INCLUDEd is limited to n. Otherwise, the limit is the count field of the feature line. If an INCREMENT appears where some licenses are USER_BASED and some are not, the vendor daemon tracks these in separate license pools.

VENDOR_STRING

`VENDOR_STRING="string"`

This keyword allows you to specify publisher-defined custom license data. If a checkout is conditional on the contents of the vendor string, use `LM_A_CHECKOUTFILTER` to check the contents of `string`. If the `VENDOR_STRING` is set, you will probably also need to set `ls_compare_vendor_*` in `lsvendor.c`.

The `VENDOR_STRING` is optionally configured as a pooling component, in which case, if the string differs in any way for two `INCREMENT` lines for the same feature name, they are pooled and counted separately.

For more information regarding the usage of the `VENDOR_STRING` keyword, see the following sections in the *Programming Reference for License File–Based Licensing*.

- **lc_auth_data**
- `LM_A_CHECKOUTFILTER`, `LM_A_CHECKOUTFILTER_EX`
- `ls_compare_vendor_on_increment` and `ls_compare_vendor_on_upgrade`

A

XML Schemas

This appendix provides details of all the external XML documents used by FlexNet Publisher Licensing Toolkit. These XML documents are all used for trusted storage–based licensing. For an introduction to trusted storage–based licensing, see Chapter 1, “Overview of Trusted Storage–Based Licensing.”.

Note • *Schemas updated for composite transactions can be found in* `<platform_dir>\publisher\schemas\Version5` *and* `Version6` *and for virtualization support in* `<platform_dir>\publisher\schemas\Version6`.

This appendix is arranged with the XML documents listed in alphabetical order. To enable you to identify the XML documents required for a particular aspect of activation, the following sections group the XML documents and give links to the required sections within the appendix.

Important • *Element names in XML files are case-sensitive. You must use the correct character case when creating XML files.*

The following table list the XML schemas in this chapter and describes where they would be needed.

Table A-1 • XML Schemas

Schema	Description
B_ActivationSpecificationRecord.xsd	This schema document defines the structure of an activation specification record (ASR) file that is used for ● Local activation ● Short-code activation
P2_ConfigKeyOut.xsd	This defines the Key File used by the manual response generator.
W1_CreateActivationResponseParameter.xsd	This defines the structure of the Activation Response and is required by the manual response generator.

Table A-1 • XML Schemas

Schema	Description
W2_CreateReturnResponseParameter.xsd	This defines the Return Response and is required by the manual response generator.
W3_CreateRepairResponseParameter.xsd	This defines the Repair Response and is required by the manual response generator.
W4C_CreateConfigResponseClientParameter.xsd	This defines the Trusted Configuration for client-side trusted storage and is required by the manual response generator.
W4S_CreateConfigResponseParameter.xsd	This defines the Trusted Configuration for server-side trusted storage and is required by the manual response generator.
W5_CreateFailureResponseParameter.xsd	This defines the failure response and is required by the manual response generator.
W6_CreateResponseCodeParameter.xsd	This defines the short-code response and is required by the manual response generator.
O_ConfigPhaseOne.xsd	This defines the XML file that is an input to tcencrypt. It is used to create the Trusted Configuration.
P1_ConfigPhaseTwo.xsd	This defines the XML file that is an output of tcencrypt.
I_FulfillmentRecord.xsd	This is referenced by B_ActivationSpecificationRecord.xsd.
M1_Dictionary.xsd	This is referenced by B_ActivationSpecificationRecord.xsd.
M2_NonEmptyString.xsd	This is referenced by B_ActivationSpecificationRecord.xsd.
M7_TrustedIdType.xsd	This is referenced by B_ActivationSpecificationRecord.xsd.
M5_ParameterHeaderType.xsd	This stores the version number of the schemas.
M5_VersionNumberType.xsd	This stores the version number of the schemas.
M6_ParameterHeaderType.xsd	This stores the version number of the schemas.
S_ActivationSettings.xsd	Defines the structure of `publisher.xml`.

Locating Schema Files in the Toolkit

Directories are provided within the toolkit that contain sets of schemas and example XML documents with the same version number:

```
<platform_dir>/publisher/schemas/Version1
<platform_dir>/publisher/schemas/Version2
```

```
<platform_dir>/publisher/xml/Version1
<platform_dir>/publisher/xml/Version2
```

Caution • Do not copy schemas from one version directory to another*. The schema names are independent of the version number so you will overwrite the existing schema. For correct operation of the manual response generator, you must ensure the integrity of the schema version directories.*

The Version 1 schema directory contains a full set of all schemas. Other directories, the Version 2 directory for example, only contain schemas that have been revised or added. Note that not all the schemas included in a Version 2 or later directory will have been changed: all schemas referenced by a revised schema are included in a version directory.

As changes are made to the available trusted storage–based licensing capabilities, the schemas may change. Elements may be added or removed from a schema and additional schemas may be defined. To ensure that all modules that use XML documents can correctly interpret a document's contents, a schema version is included in each schema that defines a complete document.

Normally this will be completely transparent to the toolkit user provided that schemas and modules from the latest version of the toolkit are consistently used. Backwards compatibility is provided with earlier versions of the toolkit. However, if later versions of the schemas are used with earlier version modules, then unpredictable results may occur.

This document describes the latest versions of the schemas, as these are the versions that are required to use all the functionality provided in this version of the toolkit.

When a document contains other documents, the version of the whole document may differ from the versions of its component parts. For example, a document conforming to version 2 of `W1_CreateActivationResponseParameter.xsd` may contain a fulfillment record conforming to version 1 of `I_FulfillmentRecord.xsd`.

Identifying Schema Versions

The schema version can be easily identified: there is a comment on the second line of each schema that indicates the version. This comment line is inserted in all schemas from version 2 onward. Schemas without this comment line, or a specified `VersionNumber`, are version 1.

Example of Schema Versions: Vendor Data

This section describes how schema versions have been used when adding a vendor data section to activation transaction documents. The vendor data section allows publishers to send information as part of an activation transaction. This data is completely under the control of the publisher: the only format constraints are that the data is sent as key-value pairs.

In version 2 of the schemas, the following element is added to the ResponseData element in `W1_CreateActivationResponseParameter`, `W2_CreateReturnResponseParameter`, and `W3_CreateRepairResponseParameter` schemas:

```
<xsd:element name="VendorDictionary" type="fnp:Dictionary" minOccurs="0" />
```

This adds an optional dictionary of key-value pairs to the responses sent to an activation utility. An activation utility uses the version number in the document to determine how to process a response document:

- If it contains no version information, then it is version 1 and does not contain a vendor data element.
- If it is version 2 or later, then it optionally may contain a vendor data element.

Functions are provided within the Activation API to access the vendor data when it is available. These functions check the version number and return an error indicating when a response document is version 1 and therefore does not contain a vendor data element.

B_ActivationSpecificationRecord.xsd

The activation specification record, ASR, is used to provide:

- **Local activation**—A *local activation* is the installation of license rights (that is, the addition of a fulfillment record to trusted storage) without a transaction with an activation server.
- **Short-code activation**—The use of request and response codes instead of XML documents for activation transactions. These are typically used for activation via a telephone.

Important • *Do not configure trial license capabilities in a short-code ASR. Flexera does not support configuring these two product features together in the same ASR file. For more information on creating trial licenses, see the section titled Chapter 4, "Implementing Trial Licenses and Trial Packs."*

Creating an Activation Specification Record

The toolkit includes a tool that creates a basic ASR that you can edit for your product. This is a command line tool, `asrgen`. A full description of how to use asrgen is given in Chapter 13, "Working with the Software Publisher Utilities." It is recommended that you create an ASR using `asrgen` and then modify it using the detailed schema information contained in this section and in I_FulfillmentRecord.xsd. After modification, the ASR must be signed using XMLSign.

The following procedure shows you how to create an ASR manually should this method be required.

Task ***To create an ASR without using asrgen***

1. Do either of the following:

- Create a `<ConfigDate>` element in the ASR similar to the following if the trusted section in which the ASR is installing the fulfillment record already exists:

```
<ConfigData>
<Id>1</Id>
<IsServer>true</IsServer>
</ConfigData>
```

  Also use this same element if the initial trusted section is being created via a prepped Trusted Configuration before the ASR is loaded. For information about setting up for deployment of a prepped Trusted Configuration, see the section Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage." (Best practice for creating an initial trusted section is to use this method.)

  Include the **`<IsServer>true</IsServer>`** parameter only if the ASR is to be loaded on a license server. (For security purposes, license-server ASRs *require* deployment of prepped Trusted Configuration for creating the initial trusted section if necessary.)

  Skip to step 5.

- Create and include the Trusted Configuration information in the ASR itself. Use this method if you are creating the initial trusted section (and the prepped Trusted Storage feature is not implemented), creating an additional trusted section, or creating an ASR for deployment on a `notr` platform.

  Continue with step 2.

2. Create a Trusted Configuration input file using O_ConfigPhaseOne.xsd.

   Note • *If the trial is to run on a virtual machine, use the Version 6 schema* `O_ConfigPhaseOne.xsd`.

3. Encrypt the Trusted Configuration data in the file using tcencrypt.
4. Extract the `<ConfigData>` element from the `<ASRConfig>` element in the encrypted Trusted Configuration file (based on P1_ConfigPhaseTwo.xsd).
5. Create a fulfillment record (I_FulfillmentRecord.xsd).
6. Optionally, create a data dictionary (DataDictionary Element) that contains key-value pairs used by the activation utility when using the ASR.

   The data held in this data dictionary is secure because the signature in the ASR prevents tampering with the contents of the ASR. Thus rules for installing a trial license should be held in the ASR and not in an insecure application data file.

7. Create the ASR using the items created in the previous steps. See the next section Description of the XML Data Structure for information about formatting these items as elements in the ASR.
8. Add other required elements (for example, the Header and a blank Signature element).
9. Add the signature to the ASR using XMLSign. See xmlsign in Appendix 13, “Working with the Software Publisher Utilities.”

Description of the XML Data Structure

The following table describes the first level XML elements in an ASR. The format of some elements are described in subsequent subsections.

Table A-2 • ASR XML data structure

Element Name	Mandatory	Format	Description
Header	Yes	See ActSpecRecHeader Element	Data that identifies the publisher

Table A-2 • ASR XML data structure

Element Name	Mandatory	Format	Description
ConfigData	No	See ConfigData Element	The configuration data for the section of trusted storage where the fulfillment record in the ASR is stored. This data consists of *either* of the following: ● The `<ConfigData>` element extracted from the `<AsrConfig>` element in the `tcencrypt` output file (based on schema P1_ConfigPhaseTwo.xsd) ● The Trusted ID (and the `<IsServer>true</IsServer>` parameter when appropriate) if the trusted section already exists or if the activation utility uses the prepped Trusted Configuration to create an initial trusted section. (The `<IsServer>` parameter indicates that the ASR is for use on license servers only.) See ConfigData Element for more information.
DataDictionary	Yes	Dictionary of key-value pairs, see DataDictionary Element	Publisher data used when processing the ASR, for example, only allow one trial.
FulfillmentRecord	Yes	See I_FulfillmentRecord.xsd	The fulfillment record, which contains the license(s) that are written to trusted storage. See the note on fulfillment ID.
ShortCodeData	No	See ShortCodeData Element	Data used when generating and processing request and response codes.
Signature	Yes	Hexadecimal string	Ensures integrity of the ASR: it is checked before the ASR is processed to ensure that the ASR has not been altered.

Note • *The fulfillment ID within the fulfillment record identifies the fulfillment record. It is used if a fulfillment record is installed from an ASR and then is returned (for example, when a fulfillment record is installed from a short-code ASR and then returned using a programmatic transaction). In this scenario, the address of the server, for the return, is passed to the activation utility.*

ActSpecRecHeader Element

The header information identifies the ASR: a version and identity are required. You allocate these so that you can identify and manage your ASRs.

Table A-3 • ActSpecRecHeader Information

Element Name	Mandatory	Format	Description
Version	Yes	Positive integer	ASR version as defined and used by the publisher
ID	Yes	String	ASR identity as defined and used by the publisher

Table A-3 • ActSpecRecHeader Information

Element Name	Mandatory	Format	Description
PublisherId	Yes	Positive integer	Publisher ID allocated by Flexera.

ConfigData Element

The `<ConfigData>` element provides information about the trusted section in which the ASR stores the fulfillment record.

Information for an Existing Trusted Section

If the ASR is storing the fulfillment record in an existing trusted section (including an initial trusted section created by a prepped Trusted Configuration), you need to provide the Trusted ID. The **`<IsServer>`** parameter is required only if the ASR is for use on a license server. For more information about the Trusted ID, see the ID description in the next section.

```
<ConfigData>
<Id>1</Id>
<IsServer>true</IsServer>
</ConfigData>
```

Note • *When prepped Trusted Configuration is enabled, an activation utility automatically creates the initial trusted section (if it does not exist) before loading the ASR. For more information about prepped Trusted Configuration, see the section Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage."*

Information to Create a Trusted Section

If the ASR needs to create the trusted section in which to store the fulfillment record, extract the `<ConfigData>` element from the `<AsrConfig>` element in the encrypted, signed Trusted Configuration file (based on P1_ConfigPhaseTwo.xsd); and insert the `<ConfigData>` element in the ASR:

```
<ConfigData>
<ClientConfig>
<Id>1</Id>
<Revision>DefaultOne</Revision>
<RevisionType>ASR</RevisionType>
<Data>8E53FE2AD9680446F8D64C42B5AF3E73753BEBCDDBA8A52323A6352C4517CE1572043CD529B90FB413AD0D5F34E0B
178B453FBFCC7F5A0399A2A8EDCAE5D55EC90FBB81A716B45D85DF3C82AB95F0086EC9B04E951E25F4540AA1923C6299315
CCFEEF36BEB614B6A17B3B83CC593D0CA045E46071824B7FC44E239DBA5EE4D4F88DB5CFF3426BC14E250EDCBE7D8E3328D
DD385D8B1B294B7D37668FACF6D12AA9091DE10777D10561B836C641F391E01E709EA1987B7505AAADA98004E0B73E2B582
0F68ABB268C62C52829521BEB85AB77A13E8A33A258B232963275D5FFAC5D858A305266577DF42BEF1E92637CD7D6DCF19D
7B8665D41FA8E4973FA2064</Data>
</ClientConfig>
</ConfigData>
```

The Trusted Configuration information includes these parameters:

Table A-4 • ConfigClient Information

Parameter	Mandatory	Format	Description
ID	Yes	Positive integer	Trusted ID used to uniquely identify this trusted section of trusted storage. Values are in the range 0 through 255.
Revision	Yes	Alphanumeric string	Identifies the revision of this Trusted Configuration. Sample configuration, request, response, and ASR files in the toolkit can show one of these values: ● **DefaultOne**—Trusted Configuration generated by FlexNet Publisher 11.10 (and based on binding identities available with 11.10). ● **NumberOne**—Trusted Configuration generated by a pre-11.10 version of FlexNet Publisher (and based on binding identities available in this earlier version).
RevisionType	Yes	● SRV ● SPE ● ASR	Defines protection given to this client config. For an ASR this should be set to ASR so that updates from a server are preserved. **SRV**—Default that can be overwritten by a config that defines revision type SRV or SPE. **SPE**—Can be specified when running tcencrypt to ensure that the client config is not overwritten. This type can only be overwritten by a config of revision type SPE. **ASR**—Inserted by asrgen. This type can be overwritten by any type.
Data	Yes	Alphanumeric string	Encrypted Trusted Configuration.

Note • *The RevisionType is automatically set to* ASR *when an ASR is created using* asrgen*. This ensures that the Trusted Configuration specified in the ASR does not overwrite any existing configuration for this trusted section.*

DataDictionary Element

All data dictionaries contain a set of entries. The first element in the entry is the Key element that identifies the type of data in this entry. The second element of the entry, the Value element, contains the data for the entry. This type of entry is referred to here and elsewhere as a *key-value pair*.

Table A-5 • DataDictionary Information

Key	Required	Format of Value
PUBLISHER_DEFINED_CRITERION	No	• **LOCAL_ONCE**–Load ASR once only. This rule is implemented by the Activation API function **flxActCommonLicSpcAddASRs** • **LOCAL_ALWAYS**–Always load ASR. This rule is implemented by the Activation API function **flxActCommonLicSpcAddASRs** Default value when this key-value pair is not present is `LOCAL_ONCE`. You must define this as `LOCAL_ONCE` for trial anchors.
PUBLISHER_DEFINED_DURATION	No	A positive integer. The time in seconds for the duration of the license. When this is a positive value it is rounded up to the nearest day and used to set the duration of a fulfillment record when it is written to trusted storage. When this is zero, no change is made to the expiration date in the fulfillment record contained in the ASR when it is written to trusted storage Default value when this key-value pair is not present is zero.
PUBLISHER_TRIAL_ID	No	A number from 1 to 255 that uniquely identifies the trial anchor. This is called the Trial ID. If you specify a value that is outside of this numeric range, the process of loading the ASR into trusted storage will fail. Each publisher defines their own specific method for assigning Trial IDs which is dependant on the business requirements. If `LOCAL_ALWAYS` is defined as well as a Trial ID, then a trial anchor is not created. The trial will be anchored using a publisher anchor only. For more information about trial anchors, see Anchoring in Chapter 3, "Configuring Security for Trusted Storage."
PUBLISHER_TRIAL_RESET	No	A number that defines the operation of the trial anchor reset functions: • **flxActCommonResetTrialASRs** • **flxActCommonResetTrialASRFromBuffe**r A non-zero value allows repeated resets of the trial anchor defined by `PUBLISHER_TRIAL_ID`.

Table A-5 • DataDictionary Information

Key	Required	Format of Value
VIRTUALIZATION_POLICY	No	The value that determines whether the ASR is used to activate licenses on a virtual machine: ● VM_ONLY–Activate licenses on virtual machines only. ● PHYSICAL–Activate licenses on physical (non-virtual) machines only. If this key is omitted, the activation utility uses the ASR for activating licenses on either virtual or physical machines. However, the ASR establishes Trusted Configuration using default binding only.

ShortCodeData Element

This element is required when using the ASR to perform short-code activation, return, and repair transactions only. It is used to generate or interpret request and response codes used in short-code transactions. This element is not used when creating an ASR to perform local activation.

When creating an ASR file for short-code transactions, make sure that the fulfillment record sets the `fulfillmentType` key to the value of 2. This indicates that the fulfillment record will be added to trusted storage using a short-code transaction. For more information about the key-value pairs defined in a fulfillment record, see I_FulfillmentRecord.xsd.

Table A-6 • ShortCodeData Child Elements

Element Name	Required	Value	Description
SchemeId	Yes	String, one of: SCS_BASIC_42 SCS_COMPAT_1 SCS_COMPAT_2 SCS_COMPAT_3 SCS_COMPAT_4 SCS_COMPAT_5 SCS_COMPAT_6	Defines the scheme used to generate the request and response codes. The scheme defines the default properties of request and response codes including: the data items they contain; the way in which they are encrypted and signed; and the character set they use (numeric or alphanumeric). Overrides when defined, change the length and/ or the data contained in the code.
ShortCodeAlias	Yes	Positive integer either 2, 4, 6, or 8 digits long. Note that the value zero (0) is included so 2 digits allows you to define 100 aliases (0–99).	This is a unique value used to identify transactions using this ASR.

Table A-6 • ShortCodeData Child Elements

Element Name	Required	Value	Description
Overrides	No	N/A	Defines changes to the default format or data captured in the request or response code. See the sections ● Data That Can Be Overridden (or Defined) in Request Codes ● Data That Can Be Overridden (or Defined) in Response Codes for information about defining or overriding the data using this element.

Choosing a SchemeId

If you have not used short codes before, you should choose one of the SCS_COMPAT_* schemes. This is because they use UMN to identify the machine that initiates a request. See Local Activation in Chapter 10, "Building an Application Activation Utility for Single-Action Transactions," for a detailed explanation.

You have multiple schemes to choose from when defining the short-code ASR. Each scheme uses the same encryption, but generates different response signature lengths. The schemes SCS_COMPAT_1, SCS_COMPAT_2, SCS_COMPAT_3, SCS_COMPAT_4, SCS_COMPAT_5, SCS_COMPAT_6 capture the same default data.

The SCS_BASIC_42 and SCS_COMPAT_1 schemes generate the same signature length, but store different data. The following table describes the signatures generated for each scheme Id.

Table A-7 • Signatures in Short-Code Schemes

Scheme Id.	Generates a...
SCS_BASIC_42	42-bit Elliptic Curve Digital Signature Algorithm (ECDSA) Signature
SCS_COMPAT_1	42-bit Elliptic Curve Digital Signature Algorithm (ECDSA) Signature
SCS_COMPAT_2	142-bit Elliptic Curve Digital Signature Algorithm (ECDSA) Signature
SCS_COMPAT_3	162-bit Elliptic Curve Digital Signature Algorithm (ECDSA) Signature
SCS_COMPAT_4	182-bit Elliptic Curve Digital Signature Algorithm (ECDSA) Signature
SCS_COMPAT_5	202-bit Elliptic Curve Digital Signature Algorithm (ECDSA) Signature
SCS_COMPAT_6	282-bit Elliptic Curve Digital Signature Algorithm (ECDSA) Signature

Each trusted section can support only one signature strength. This means that once you have initialized and configured the trusted section (using the Trusted Configuration), all short-code transactions that used that trusted section must use that same signature strength. Each trusted section is identified by a unique TrustedId. For example, trusted section 1 could be configured for 42-bit signing and trusted section 2 can support 202-bit signing. Any ASR that contains data for trusted section 1 must use a 42-bit signature.

Make sure that the short-code ASR defines a Scheme Id that is supported by the trusted-section transaction key length. You configure the transaction key length using the TransactionKeyInfo element in the Trusted Configuration (defined by O_ConfigPhaseOne.xsd schema).

Default Short-Code Formats

The following sections describe the contents of request and response codes when no overrides have been defined, that is the *default short-code format*. Descriptions of all the default items in short codes are listed below and are referenced in the tables which describe the default short-code formats.

- **SCI_ERROR_ID**—This item is only used in error responses to indicate the reason for the error response. If you are using the Manual Response Generator to generate a short code, the value of this item must be added to the XML document used to create the response code. See W6_CreateResponseCodeParameter.xsd for details.
- **SCI_ORIGINAL_MACHINE_ID**—Machine ID is a generated value based on the binding. It does not identify a machine (a different machine identifier is generated for each trusted section). Also, when a binding break occurs it is recalculated. The original machine ID is the value of machine id when the fulfillment record was created.
- **SCI_NEW_MACHINE_ID**—The newly calculated machine ID. When there has been a binding break or the binding has changed within the allowed tolerance, this will differ from the original machine ID. This item is only used in repair requests.
- **SCI_REPAIR_SCOPE**—This item is only used in repair responses and indicates which fulfillment records are to be repaired. This item must not be overridden. The following table gives all possible values for this item.

Table A-8 • SCI_REPAIR_SCOPE Values

Value	Description
0	Repair only the fulfillment record specified in the request.
1	Repair all fulfillment records in the trusted section specified in the request.
2	Repair all fulfillment records for this publisher.

- **SCI_REQUEST_SEQUENCE_NO**—The request sequence number for this transaction. This number is incremented for each short-code transaction for any short-code alias. It can be used to determine when a request has been resent because a response has not been received.
- **SCI_TRUST_FLAGS**—This item is only used in repair requests and it indicates the reason for the repair. There are three trust flags that indicate the type of trusted storage break that has occurred: anchoring, binding and time. They are coded within a request code as 3 bits where the least significant bit is the anchoring trust flag, bit 2 is the binding trust flag and bit 3 is the time trust flag. The bit is set if the flag is trusted and not set if the flag is untrusted. The following table gives all possible values, in decimal, for this item.

Table A-9 • SCI_TRUST_FLAGS Values

Value	Indicates that the following flags are untrusted...
0	• Anchoring • Binding • Time
1	• Binding • Time
2	• Anchoring • Time
3	• Time
4	• Anchoring • Binding
5	• Binding
6	• Anchoring
7	• None - all trust flags set to trusted

- **SCI_TSSN**—Trusted Storage Serial Number is a generated value that is likely to be unique for a particular machine. This value is the same for all requests from a specific machine provided that trusted storage has not been recreated. If trusted storage is deleted and recreated, a new value is generated. This can be used to identify a particular trusted storage file. This can be used for example:
  - When you want to make licensing decisions for associated products that use different trusted Ids.
  - To identify the machine when the machine ID has changed but no binding break has occurred, as the binding is still within tolerance.
- **SCI_UMN1**—The value captured for the Type 1 Unique Machine Number (UMN) of the system that initiated the request.
- **SCI_UMN1_IS_VALID**—Indicates whether a valid value was obtained for UMN1.
- **SCI_UMN2**—The value captured for the Type 2 Unique Machine Number (UMN) of the system that initiated the request.
- **SCI_UMN2_IS_VALID**—Indicates whether a valid value was obtained for UMN2.

Default Request Code Formats

It is recommended that you use one of the SCS_COMPAT_* schemes that all have the same default format as defined in the following table:

Table A-10 • Default Short-Code Request Formats for SCS_COMPAT_* Scheme Ids

Item Name	Request Type	Format	Size
SCI_REQUEST_SEQUENCE_NO	Activation Return Repair	Decimal	4 characters
SCI_UMN1_IS_VALID	Activation Return Repair	Binary	1 bit
SCI_UMN1	Activation Return Repair	Hexadecimal	4 characters
SCI_UMN2_IS_VALID	Activation Return Repair	Binary	1 bit
SCI_UMN2	Activation Return Repair	Hexadecimal	4 characters
SCI_TRUST_FLAGS	Repair	Binary	3 bits

The default is for short-code requests to be numeric: all short-code requests are numeric unless the CodeBase override is used when short-code requests are generated as alphanumeric strings.

Use the SCS_BASIC_42 scheme only if you need to do so for backwards compatibility. Its default request code formats are described in the following table:

Table A-11 • Default Short-Code Request Formats for SCS_BASIC_42 Scheme Ids

Item Name	Request Type	Format	Size
SCI_REQUEST_SEQUENCE_NO	Activation Return Repair	Decimal	4 characters

Table A-11 • Default Short-Code Request Formats for SCS_BASIC_42 Scheme Ids

Item Name	Request Type	Format	Size
SCI_TSSN	Activation Return Repair	Hexadecimal	4 characters
SCI_ORIGINAL_MACHINE_ID	Activation Return Repair	Hexadecimal	4 characters
SCI_TRUST_FLAGS	Repair	Binary	3 bits
SCI_NEW_MACHINE_ID	Repair	Hexadecimal	4 characters

Default Response Code Formats

The default items in response short codes are the same for all schemes and are listed in the following table:

Table A-12 • Default Contents of Response Short Codes

Item Name	Response Type	Format	Size
SCI_ERROR_ID	Error	Decimal	2 characters
SCI_REPAIR_SCOPE	Repair	Decimal	1 character

Default Short-Code Lengths

The length of a short code is determined by the following:

- The SchemeId defined in the short-code ASR.
- Whether the code is numeric or alphanumeric (see the CodeBase attribute).
- If you have defined a short-code override.
- The type of code (for example, activation request, return request).

Lengths are rounded up to the nearest 4 character increment. See the following table for the default short-code lengths.

Table A-13 • Default Short-Code Lengths

Scheme	Activation Request	Return Request	Repair Request	Responses
SCS_BASIC_42	28 numeric/ 20 alphanumeric		36 / 20	24 / 16

Table A-13 • Default Short-Code Lengths

Scheme	Activation Request	Return Request	Repair Request	Responses
SCS_COMPAT_1	28 numeric/ 20 alphanumeric		28 / 20	24 / 16
SCS_COMPAT_2				52 / 36
SCS_COMPAT_3				60 / 40
SCS_COMPAT_4				64 / 44
SCS_COMPAT_5				72 / 48
SCS_COMPAT_6				96 / 64

Using Short-Code Overrides

There are only a limited number of situations where you might want to override the data captured in a short code. These include:

- Changing the amount of space allotted in the short code to store a built-in property. For example, if you want to send 8 digits of the entire 40-digit UMN value, rather than 4 digits. Please note that this may change the length of the short code. See Increasing the Length of UMNs for an example.
- Adding custom data to the short-code message, see the section Adding Custom Data to a Short-Code Request in Chapter 12, Building an Activation Utility for Composite Transactions for an example.
- Enabling the specification of a license expiration date in a short-code response, see Specifying License Expiration in a Short-Code Activation for an example.

Predefined Override Items

The following override items are predefined and when used will be interpreted by the activation utility software as described.

- **MinimumCodeSize**–This sets the minimum number of characters of request or response codes. This may be required when codes are processed by other systems. This is intended to be used to make sure that codes of all types are the same length. Codes that are shorter are padded to this size.

  This may be required where the codes are recorded or processed by other systems or where a user interface uses fixed length fields to display codes.
- **CodeBase**-Defines whether request or response codes are generated as alphanumeric or numeric values. If you do not override the value, the default code type is numeric. If you enter an invalid value, the default DEC value is used.
- **PUBLISHER_DEFINED_EXPIRY_CDAY**-Used by an activation server to specify a license expiration date in a response code. This expiration date is used by the client to set the expiration date into the fulfillment record that is written to trusted storage on receipt of the response code. The expiration date is expressed as the number of days since 1st January 1970. A value of 0 (zero) is given the special meaning “permanent”: setting zero for this item in the response results in the expiration date in the fulfillment record added to trusted storage being set to “permanent. This override

can only be used with activation utilities created with version 11.6.1 or greater of the toolkit. When this override is specified the date specified in the response code is used to create a fulfillment record in trusted storage with this expiration date. When this override is not used, the expiration date defined in the fulfillment record within the ASR is used.

- **PUBLISHER_DEFINED_DURATION**-Used by an activation server to specify a license duration in a response code. This license duration is used to set the expiration date by adding it to the current date and writing the resultant expiration date into the fulfillment record that is written to trusted storage. The license duration is specified in seconds so this significantly increases the response code length. It is recommended that PUBLISHER_DEFINED_EXPIRY_CDAY is used instead.

Table A-14 • Predefined Override Items

Item Name	Request/ Response Type	Format	Defined in element
MinimumCodeSize	Request - Activation, Return, Repair Response - Activation, Return, Repair, Error	Decimal	Dictionary
CodeBase	Request - Activation Response - Activation, Return, Repair, Error	String	Dictionary
PUBLISHER_DEFINED_EXPIRY_CDAY	Response - Activation	Decimal	Item
PUBLISHER_DEFINED_DURATION	Response - Activation	Decimal	Item

Defining the RequestOverrides and ResponseOverrides Element

The RequestOverrides and ResponseOverrides element can contain two child elements that are used when overriding specific data items. They are the following:

- *Items* element: this element takes a list of Item elements.
- *Dictionary* element: this element takes a set of entries that each represent a Key-Value pair.

The following table shows the element structure of the Item element when used for defining overrides. The same structure is used when defining request and response overrides.

Table A-15 • Item Element Structure for Overrides

Element Name	Required	Value	Description
Group	Yes	One of:	The type of request code that this item is included in.
		REQ_GROUP_COMMON	REQ_GROUP_COMMON—All request codes.
		REQ_GROUP_ACTIVATION	REQ_GROUP_ACTIVATION—Activation request codes.
		REQ_GROUP_RETURN	REQ_GROUP_RETURN—Return request codes.
		REQ_GROUP_REPAIR	REQ_GROUP_REPAIR—Repair request codes.
		RSP_GROUP_COMMON	RSP_GROUP_COMMON—All response codes.
		RSP_GROUP_ACTIVATION	RSP_GROUP_ACTIVATION—Activation response codes.
		RSP_GROUP_RETURN	RSP_GROUP_RETURN—Return response codes.
		RSP_GROUP_REPAIR	RSP_GROUP_REPAIR—Repair response codes.
		RSP_GROUP_ERROR	RSP_GROUP_ERROR—Error response codes.
Name	Yes	The name of the data item to be overridden as a string.	Defines the name of the data item that is overridden. This can be: ● A data item included in the default format, see Default Short-Code Formats. In this case, the configuration defined by this entry replaces the configuration in the scheme. ● A predefined item, see Predefined Override Items. ● Custom data, see the section Using the Vendor Dictionary to Send Custom Data in Chapter 10, "Building an Application Activation Utility for Single-Action Transactions."
Type	Yes	One of:	Defines the characters used in the value generated.
		BITS	**BITS** – the value is created using the characters 0 – 9 and A – F.
		DEC	**DEC** – the value is created using the characters 0 – 9.
		HEX	**HEX** – the value is created using the characters 0 – 9 and A – F.
		ASCII96	**ASCII96** –- the value is created using characters in the range 0x20 to 0x7f
		UTF8	**UTF8** – the value is created using the characters in the range 0x00 to 0xFF.
Size	Yes	Positive integer	The size of this element. This is used in conjunction with the Type element. For example if Type is set to BITS and Size to 3, then this element will be 3 bits long in the request code. Values that are longer will be truncated.

Table A-15 • Item Element Structure for Overrides

Element Name	Required	Value	Description
Location	No	One of: BUILT_IN FULFILLMENT_DICTIONARY VENDOR_DICTIONARY DEFAULT	This defines where the data item will be stored. If it is not defined, then the location is determined using the rules explained under DEFAULT. **BUILT_IN**–Use when overriding the default items. Data is stored in a system-defined location. **FULFILLMENT_DICTIONARY**–Use when overriding an attribute of a fulfillment record. The value is written to the FulfillmentDictionary element in the fulfillment record. **VENDOR_DICTIONARY** – Use when defining custom data. Data is written to the VendorDictionary element. See the section Adding Custom Data to a Short-Code Request in Chapter 12, Building an Activation Utility for Composite Transactions **DEFAULT**– Use the default rules to determine where the data item is stored: Built-in data items are stored in the system-defined location. Data items that are not built-in are stored in the FulfillmentDictionary element if it exists. Otherwise, they are stored in the VendorDictionary element.

There are only two items that are added as overrides using the dictionary element. These are described in the following table:

Table A-16 • Override Items Using Dictionary Entries

Key Name	Value	Description
MinimumCodeSize	String Number of characters in decimal format	This sets the minimum length of a response codes and may be required where the response codes are recorded or processed by other systems.
CodeBase	One of: ALNUM32 DEC	Defines whether response codes are generated as alphanumeric (ALNUM32) or numeric (DEC). If you do not override the value, the default code type is numeric. If you enter an invalid value, the default DEC value is used.

Data That Can Be Overridden (or Defined) in Request Codes

Using the RequestOverrides element, you can either override existing data items or you can add custom data to a request code. When defining custom data, you choose the data item name. The following items can be overridden or added to the format of a request code. When a RequestOverrides is not defined for an item, the default values for the items are captured as described in Default Request Code Formats.

The following items in SCS_COMPAT_* requests can be overridden.

- **SCI_REQUEST_SEQUENCE_NO** - Define an override using an item element, selecting the appropriate settings for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT.
- **SCI_UMN1** - Define an override using an item element, selecting the appropriate setting for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT. You should configure an override if you either do not want to include UMN1 or you want to include a larger or smaller amount of data (for example, 2 characters rather than 4). The complete UMN is a 40 character hexadecimal number.
- **SCI_UMN2** - See SCI_UMN1 for details.
- **SCI_UMN1_IS_VALID** - Define an override using an item element, selecting the appropriate setting for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT. This item must only be overridden if SCI_UMN1 is removed from the code by setting an override with Size of zero.
- **SCI_UMN2_IS_VALID** -See SCI_UMN1_IS_VALID for details.
- **MinimumCodeSize** - Define an override using a dictionary element.
- **CodeBase** - Define an override using a dictionary element.

The following items in SCS_BASIC_42 requests can be overridden.

- **SCI_REQUEST_SEQUENCE_NO** - Define an override using an item element, selecting the appropriate settings for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT.
- **SCI_TSSN** - Define an override using an item element, selecting the appropriate setting for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT.
- **SCI_ORIGINAL_MACHINE_ID** - Define an override using an item element, selecting the appropriate setting for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT.
- **SCI_NEW_MACHINE_ID** - Define an override using an item element, selecting the appropriate setting for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT.
- **MinimumCodeSize** - Define an override using a **dictionary element.**
- **CodeBase** - Define an override using a dictionary element.

Caution • *The licensing toolkit allows for the creation of a short-code ASR conforming to SCS_BASIC_42 that specifies alphanumeric short codes.*

Example Request Code Overrides

Adding Custom Data to an Activation Request

This example enables the activation utility to send custom data in an activation request code. The data, Key = VENDOR_DATA and Value is a two decimal digit number, is added to the VendorDictionary element using the activation API function **flxActAppActivationVendorDataSet**.

```
<RequestOverrides>
   <Items>
      <Item>
         <Group>REQ_GROUP_ACTIVATION</Group>
```

```
            <Name>VENDOR_DATA</Name>
            <Type>DEC</Type>
            <Size>2</Size>
            <Location>VENDOR_DICTIONARY</Location>
         </Item>
      </Items>
   </RequestOverrides>
```

For a more complete example, see the section Adding Custom Data to a Short-Code Request in Chapter 12, Building an Activation Utility for Composite Transactions

Increasing the Length of UMNs

This example increases the size of the UMNs in all requests from the default of 4 hexadecimal characters to 8.

```
<RequestOverrides>
   <Items>
      <Item>
         <Group>REQ_GROUP_COMMON</Group>
         <Name>UMN1</Name>
         <Type>HEX</Type>
         <Size>8</Size>
         <Location>BUILT_IN</Location>
      </Item>
   </Items>
   <Items>
      <Item>
         <Group>REQ_GROUP_COMMON</Group>
         <Name>UMN2</Name>
         <Type>HEX</Type>
         <Size>8</Size>
         <Location>BUILT_IN</Location>
      </Item>
   </Items>
</RequestOverrides>
```

Using the Dictionary Element to Set Alphanumeric Codes

This example sets all request codes to be alphanumeric.

```
<RequestOverrides>
         <Dictionary>
            <Entries>
               <Entry>
                  <Key>CodeBase</Key>
                  <Value>ALNUM32</Value>
               </Entry>
            </Entries>
         </Dictionary>
</RequestOverrides>
```

Data That Can Be Overridden (or Defined) in Response Codes

Using the ResponseOverrides element, you can either override existing data items or your activation server can add custom data to a response code. When defining custom data, you choose the data item name. The following items can be overridden or added to the format of a response code. When a ResponseOverrides is not defined for an item, the default values for the items are captured as described in Default Response Code Formats.

The following items can be overridden or added to the response.

- **SCI_ERROR_ID** - Define an override using an item element, selecting the appropriate settings for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT. Generally, the only changes you should make to this is the size allotted for the value.
- **SCI_REPAIR_SCOPE** - Define an override using an item element, selecting the appropriate settings for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT.
- **PUBLISHER_DEFINED_EXPIRY_CDAY** - Define an override using an item element, selecting the appropriate settings for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT.
- **PUBLISHER_DEFINED_DURATION** - Define an override using an item element, selecting the appropriate settings for Group, Name, Type and Size; the Location element should be set to BUILT_IN or DEFAULT.
- **MinimumCodeSize** - Define an override using a dictionary element.
- **CodeBase** - Define an override using a dictionary element.

When you define a response code override, make sure your activation server is configured to add the correct items in the correct format to the response code. If you are using FlexNet Operations as your activation server, please consult the documentation; not all short-code functionality is supported by FlexNet Operations. You can use the responsegen API to build your own activation server.

Example Response Code Overrides

Adding Custom Data to the Response Code

This example enables the activation server to return custom data in an activation response code. The data is stored in the VendorDictionary element. Use the activation API functions **flxActCommonProdLicSpcVendorDataGetByKey** or **flxActCommonProdLicSpcVendorDataGetByIndex** to extract the custom data from the vendor dictionary.

```
<ResponseOverrides>
   <Items>
      <Item>
         <Group>RSP_GROUP_ACTIVATION</Group>
         <Name>VENDOR_DATA1</Name>
         <Type>DEC</Type>
         <Size>2</Size>
         <Location>VENDOR_DICTIONARY</Location>
      </Item>
   </Items>
</ResponseOverrides>
```

Specifying the Minimum Size of Response Codes

This example specifies a minimum size of 34 characters in response codes.

```
<ResponseOverrides>
          <Dictionary>
             <Entries>
                <Entry>
                   <Key>MinimumCodeSize</Key>
                   <Value>34</Value>
                </Entry>
             </Entries>
          </Dictionary>
</ResponseOverrides>
```

Specifying License Expiration in a Short-Code Activation

The following is a complete end-to-end example of how to use PUBLISHER_DEFINED_EXPIRY_CDAY to specify a license expiration in a short-code activation response. It assumes that the activation server is created using the responsegen API as this functionality was not available in FlexNet Operations when this document was written.

The steps in this example enable the publisher to specify an expiration date when a short-code activation request is received. The default behavior is to use the fixed expiration date written into the fulfillment record contained within the ASR.

Task ***To specify a license expiration in an short-code activation response:***

1. Create an ASR that contains a PUBLISHER_DEFINED_EXPIRY_CDAY response override:

   Add the following override element to the ASR:

   ```
   <ResponseOverrides>
      <Items>
         <Item>
            <Group>RSP_GROUP_ACTIVATION</Group>
            <Name>PUBLISHER_DEFINED_EXPIRY_CDAY</Name>
            <Type>DEC</Type>
            <Size>5</Size>
            <Location>FULFILLMENT_DICTIONARY</Location>
         </Item>
      </Items>
   </ResponseOverrides>
   ```

2. Install the ASR and the activation utility on the activation client.

3. Use the ASR and the application activation utility on the activation client to create a short-code activation request.

   You do not need to update the code in your application activation utility.

4. Process the short-code activation request on the license generator.

   You do not need to update the code in your license generation application that analyses the short-code activation request, as it has no additional data.

5. Create a short-code activation response on the license generator:

   Add the PUBLISHER_DEFINED_EXPIRY_CDAY to the short-code response using the **flxLicGenShortCodeResponseDataItemAdd** function as in the following example code which sets the expiration date at 1st January 2010 (30 x 365 + 10 x 366 + 1):

   ```
   flxLicGenShortCodeResponseCreate (theRequest, &theResponse);
   ```

```
flxLicGenShortCodeResponseDataItemAdd(theResponse,"PUBLISHER_DEFINED_EXPIRY_CDAY","14611");
```

To set ‘permanent’ into the fulfillment record set the value of PUBLISHER_DEFINED_EXPIRY_CDAY to zero as in the following example code:

```
flxLicGenShortCodeResponseCreate (theRequest, &theResponse);
flxLicGenShortCodeResponseDataItemAdd (theResponse, "PUBLISHER_DEFINED_EXPIRY_CDAY", "0");
```

6. Process the short-code activation response on the activation client:

   You do not have to update the code in your application activation utility, however it must be built with version 11.6.1 or later of the toolkit. The response code is input and processed in the normal way using **flxActShortCodeProcessResponse**.

Sample ASRs in the Toolkit

The following example ASRs are included in the toolkit:

- Sample Trial Editions and Durations for FlexEnabled Clients
- Sample Trials Supporting Prepped Trusted Configuration for FlexEnabled Clients
- Sample Trials for FlexNet License Servers
- Sample Trials for FlexEnabled Clients Running in Virtual Environments
- Sample ASR to Demonstrate Trial Misuse on a FlexEnabled Client
- Sample ASR for Emergency Situations on FlexEnabled Clients
- Sample Short-Code ASRs

Sample Trial Editions and Durations for FlexEnabled Clients

The following are basic trial samples. They are found in subfolders of the same names under `<install_dir>/examples/ezcalc`.

- **1day_student**—A trial license with a duration of one day. The license includes four licensed features.
- **15day_engr**—A trial license with a duration of 15 days. The license includes five licensed features.
- **30day_acct**—A trial license with a duration of 30 days. The license includes five licensed features.

Sample Trials Supporting Prepped Trusted Configuration for FlexEnabled Clients

The following sample trials for FlexEnabled clients assume that prepped Trusted Configuration is enabled to create the initial trusted section in client-side trusted storage. For more information about prepped Trusted Configuration, see the section Deploying Trusted Configuration in Chapter 3, “Configuring Security for Trusted Storage.” These samples are found in `<install_dir>/examples/ezcalc/30day_acct_default_config`.

- **30day_acct_physical**—A trial license for an environment in which prepped Trusted Configuration is set up to create an initial trusted section on a physical FlexEnabled client machine.

- **30day_acct_virtual**—A trial license for an environment in which prepped Trusted Configuration is set up to create an initial trusted section on a virtual FlexEnabled client machine.

Note • *ASRs created to load licenses on a license server require the use of prepped Trusted Configuration. See the next section, Sample Trials for FlexNet License Servers, for samples.*

Sample Trials for FlexNet License Servers

The following sample ASRs activate and reset trial licenses on a FlexNet license server and assume that prepped Trusted Configuration is enabled.

Note • *For security purposes, ASRs that load licenses on a license server require that prepped Trusted Configuration be enabled to create the initial trusted section in server-side trusted storage. For more information about prepped Trusted Configuration, see the section Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage."*

These samples are found in `<install_dir>/examples/ezcalc/30day_trial_server`.

- **30day_trial_server**—A trial license for a FlexNet license server. Activations via an ASR on a license server require that prepped Trusted Configuration be deployed.
- **30day_trial_reset**—(For publisher use only) The file used to reset the trial for testing purposes. (Use `30day_trial_server` to reload the licenses.)

Sample Trials for FlexNet License Servers Running in Virtual Environments

The following sample ASRs assist in the evaluation of server-side activation on a virtual platform.

Note • *For security purposes, ASRs that load licenses on a license server require that prepped Trusted Configuration be enabled to create the initial trusted section in server-side trusted storage. (For more information about prepped Trusted Configuration, see the section Deploying Trusted Configuration in Chapter 3, "Configuring Security for Trusted Storage.") However, best practice is not to deploy virtualization-enabled server ASRs in production environments unless the license server is considered a trusted system by the publisher. It is somewhat easy to circumvent server ASR expirations in virtual environments.*

These samples are found in `<install_dir>/examples/ezcalc/30day_trial_server`.

- **30day_trial_server_VM**—A trial license for a FlexNet license server running on a virtual machine. Activations via an ASR on a license server require that prepped Trusted Configuration be deployed.
- **30day_trial_reset_VM**—(For publisher use only) The file used to reset the trial for testing purposes. (Use `30day_trial_server_VM` to reload the licenses.)

Sample Trials for FlexEnabled Clients Running in Virtual Environments

The following sample trials are used in an environment that includes both physical and virtual machines on which FlexEnabled clients run. These samples are found in `<install_dir>/examples/ezcalc/30day_acct_virtual`.

- **30day_acct_phy_vir**—A trial license for either a virtual or non-virtual (physical) machine. The license includes five licensed features and lasts 30 days. The ASR defines the `VIRTUALIZATION_POLICY` as **`PHYSICAL, VM_ONLY`** and uses default binding.
- **30day_acct_physical**—A trial license for a non-virtual (physical) machine only. The license includes five licensed features and lasts 30 days. The ASR defines the `VIRTUALIZATION_POLICY` as **`PHYSICAL`** and uses default binding.
- **30day_acct_virtual**—A trial license for a virtual machine only. The license includes five licensed features and lasts 30 days. The ASR defines the `VIRTUALIZATION_POLICY` as **`VM_ONLY`** and uses VMID-only binding.

Sample ASR to Demonstrate Trial Misuse on a FlexEnabled Client

The following sample ASR is used in a tutorial in FlexNet Publisher's *Getting Started with Trusted Storage–Based Licensing* to demonstrate an attempt to extend a FlexEnabled-client trial without authorization. This sample is found in `<install_dir>/examples/ezcalc/30day_acct_reset`.

- **30day_acct_reset**—(For publisher use only) The file used to reset and reload the trial for testing purposes.

Note • *For information about a sample ASR used to reset a license-server trial, see Sample Trials for FlexNet License Servers.*

Sample ASR for Emergency Situations on FlexEnabled Clients

This ASR is designed for situations when an activation server connection is not available and a temporary license is required on the FlexEnabled client. This sample differs from the other ASR samples in that the `fulfillmentType` attribute within the fulfillment record indicates that this ASR is for emergency use rather than for a trial. This attribute does not ensure that the ASR is used to obtain only in emergency situations; it simply describes how this ASR should be used.

This sample is found in `<install_dir>/examples/ezcalc/emergency`.

- **emergency**—An emergency license with a duration of 3 days. It includes four licensed features.

Sample Short-Code ASRs

The following sample ASRs demonstrate short-code transactions to activate licenses on a FlexEnabled client. These samples are found in `<install_dir>/<platform_dir>/publisher/xml.`

- **ShortCodeA**—A short-code ASR for the short-code alias 99. The minimum length of request and response codes is set to 24 characters. The default values of request and response code elements are used.
- **ShortCodeB**—A short-code ASR for the short-code alias 100. The minimum length of request and response codes is set to 24 characters. The default values of request and response code elements are used.
- **ShortCodeC**—A short-code ASR for the short-code alias 101. The minimum length of request and response codes is set to 24 characters. There is no trusted storage serial number in request codes (its length is set to zero). All other request and response code elements use the default values.
- **ShortCodeD**—A short=code ASR for the short-code alias 102. The minimum length of request and response codes is set to 24 characters. The default values of request and response code elements are used with one exception: the length of the trusted storage serial number is reduced from 4 to 2 hexadecimal characters.

- **ShortCodeE**—A short-code ASR for the short-code alias 103. The minimum length of request and response codes is set to 24 characters. The trusted storage serial number in request codes is reduced from 4 to 3 hexadecimal characters. The machine identity in request codes is reduced from 4 to 3 hexadecimal characters. All other request and response code elements use the default values.

See Also
I_FulfillmentRecord.xsd
M2_NonEmptyString.xsd

I_FulfillmentRecord.xsd

A fulfillment record contains details of license rights provided by a single activation. It is placed in trusted storage during a successful activation.

Creating a Fulfillment Record

There are two methods of creating a fulfillment record:

- Creating an ASR using `asrgen`.
- Writing an XML document based on the schema.

Using asrgen

The toolkit includes a tool that creates a basic ASR that contains a fulfillment record. This is a command line tool, asrgen. A full description of how to use asrgen is given in Chapter 13, "Working with the Software Publisher Utilities." You create an ASR using asrgen and then extract and modify the fulfillment record contained within it using the detailed schema information contained in this appendix.

Using the Schema

You can use the schema and the descriptions contained within this section to create a fulfillment record.

Task ***To create a fulfillment record***

1. Create a license file that contains feature definition lines and a SERVER line. Some information about creating this license file is given in `asrgen` as it is a required input parameter for asrgen.
2. Use `lmcrypt` to add signatures to the feature definition lines.
3. Create a fulfillment record using the `I_FulfillmentRecord.xml` schema, inserting the signed feature definition lines in the value element for the features key.
4. Create the required key-value pairs that define the license rights associated with the features included in the feature definition lines.

   See XML Structure of a Fulfillment Record for details that are specific to fulfillment records held in client-side trusted storage and server-side trusted storage.

XML Structure of a Fulfillment Record

Table A-17 shows the structure of a fulfillment record.

Table A-17 • Fulfillment Record Structure

Element Name	Required	Format	Description
VersionNumber	Yes	Numeric	Defines the format of the fulfillment record. For this release this must be 1.
FulfillmentId	Yes	Alphanumeric	A unique identity for this fulfillment.
FulfillmentDictionary	Yes	Dictionary of key-value pairs. See FulfillmentDictionary Data in Server-side Trusted Storage and FulfillmentDictionary Data in Client-side Trusted Storage for the dictionary entries and their formats.	The data that defines the license rights supplied by this fulfillment record.

FulfillmentDictionary Data in Client-side Trusted Storage

The following table gives details of each of the Key-Value pair entries that can be defined using the FulfillmentDictionary element in the fulfillment record for trusted storage on a FlexEnabled client. Key-Value pairs that are ignored in a fulfillment record on a FlexEnabled client are not included in the table.

Table A-18 • Fulfillment Record Description

Key Name	Required	Value Format	Description
flexFulfillment Version	No, see description	Numeric	Defines the format of the fulfillment dictionary. For this release this must be 1. If this entry is not present, a value of 0 (zero) is assumed.

Table A-18 • Fulfillment Record Description

Key Name	Required	Value Format	Description
fulfillmentType	No	Numeric	The type of "transaction" that placed this fulfillment record in trusted storage. The following are valid values: ● 0 = Unknown ● 1 = 'Trial' from ASR used for trial license, that is where PUBLISHER_DEFINED_ CRITERION is LOCAL_ONCE. ● 2 = From a short-code ASR, see ShortCodeData Element. ● 3 = 'Emergency' from ASR used for an 'emergency' license, that is when a connection to a server is not available. This will normally be a limited duration license. ● 4 = 'Served Activation' from a license server standard pool of activatable licenses ● 5 = 'Served overdraft Activation' from a license server overdraft pool of activatable licenses ● 6 = 'Publisher Activation' from a FlexNet Operations server ● 7 = Not used For usage information, see **flxActCommonProdLicSpcFulfillmentTypeGet**.
expiry	Yes	String	The license expiration date. This overrides the values in the feature definition lines. ● `permanent`—does not expire ● `dd-mmm-yyyy`—where *mmm* is the month as three letters

Table A-18 • Fulfillment Record Description

Key Name	Required	Value Format	Description
serverHostID	Yes	Any valid hostid	The SERVER line hostid used when signing the feature definition lines included in this fulfillment record. It is recommended that you use a value that is difficult to spoof, for example, a FlexNet ID dongle. Although this value comes from the SERVER line, it is not used to lock concurrent licenses to the system running the license server. when you sign a license file, the information on the SERVER line is embedded in all feature definition line signatures. This value must be captured in the fulfillment record so that the FlexEnabled application can authenticated these signatures. When defining the hostid make sure you select a type and value that cannot easily be recreated. This prevents someone from copying this license rights information into a license file to create additional unauthorized licenses.
servOrLocal	Yes	Numeric	Must be set to 2 to denote that this fulfillment record is used on a FlexEnabled client.
productId	Yes	Alphanumeric	Identifies the product that uses the license rights in this fulfillment record.
entitlementId	Yes	Alphanumeric	An identity that defines what license rights the user is entitled to.
availAct	Yes	Numeric	The number of activatable licenses available.
features	Yes	For details, see Chapter 15, "Defining Feature Definition Lines for a Fulfillment Record."	Feature definition lines. Each line gives license rights for a single feature. The feature definition lines must be signed using the hostid specified in ServerHostID.
actServerChain	No	Alphanumeric Format is: *Server_URL* [|*Server_URL*]	Activation server chain—the list of servers through which this fulfillment record has been supplied. The chain is in reverse order so that the first entry is the last server through which this fulfillment record passed.
fulfillmentId Chain	No	Alphanumeric Format is: *fulfillmentId* [|*fulfillmentId*]	Fulfillment identity chain—the list of fulfillment identities that have been used for this fulfillment record as it moved through the activation server chain. See actServerChain.

Table A-18 • Fulfillment Record Description

Key Name	Required	Value Format	Description
groupId	No	Numeric	This entry is inserted by a license server when it responds to an activation request. It indicates the license group from which this activation was taken, according to the following: ● 1 = activatable ● 2 = activatable overdraft ● 5 = hybrid ● 6 = hybrid overdraft

Note • *The following is additional information about fulfillment record key-value pairs:*

- **Key Names**—*Are case sensitive.*
- **flexFulfillmentVersion**—*The activation modules will reject a fulfillment record with a flexFulfillmentVersion that is greater than the version specified within the module.*
- **actServerChain**—*When creating a fulfillment record, put in the URL of the server that is supplying this fulfillment record. Then the fulfillment record can be returned to the server. When creating a fulfillment record that will be supplied by a manual response generator, put in text to indicate that this was the source of this fulfillment record.*
- **fulfillmentIdChain**—*When creating a fulfillment record, put in a value that indicates this is the source fulfillment identity. As the fulfillment is moved around by activation transactions a new fulfillment identity is created for each activation transaction. The sequence of these fulfillment identities is recorded in the fulfillment ID chain.*

FulfillmentDictionary Data in Server-side Trusted Storage

The following table gives details of each of the Key-Value pair entries for a fulfillment record that is held in server-side trusted storage on a license server. This data is used to serve licenses to other license servers or FlexEnabled clients.

Table A-19 • License Server Fulfillment Record Description

Key Name	Required	Value Format	Description
flexFulfillment Version	No, see description	Numeric	Defines the format of the fulfillment dictionary. For this release this must be 1. If this entry is not present, a value of 0 (zero) is assumed.
expiry	Yes	String	The license expiration date. This overrides the values in the feature definition lines. ● `permanent`—does not expire ● `dd-mmm-yyyy`—where *mmm* is the month as three letters
serverHostID	Yes	Any valid hostid	The SERVER line hostid used when signing the feature definition lines included in this fulfillment record. It is recommended that you use a value that is difficult to spoof, for example a FLEXid. Although this value comes from the SERVER line, it is not used to lock concurrent licenses to the system running the license server. when you sign a license file, the information on the SERVER line is embedded in all feature definition line signatures. This value must be captured in the fulfillment record so that the FlexEnabled application can authenticated these signatures. When defining the hostid make sure you select a type and value that cannot easily be recreated. This prevents someone from copying this license rights information into a license file to create additional unauthorized licenses.
servOrLocal	Yes	Numeric	Must be set to 1 to denote that this fulfillment record is used on a license server.
productId	Yes	Alphanumeric	Identifies the product that uses the license rights in this fulfillment record.
entitlementId	Yes	Alphanumeric	An identity that defines the license rights contained in this fulfillment record. It is used in requests to the server that can be served using this fulfillment record.
suiteId	No	Alphanumeric	Identifies the suite that this product has been packaged into.

Table A-19 • License Server Fulfillment Record Description

Key Name	Required	Value Format	Description
maxAct	No	Integer. Default of 0 if this key-value pair is not specified.	The maximum value allowed for availAct. Set equal to availAct to limit fraudulent attempts to add extra licenses.
availAct	No	Integer. Default of 0 if this key-value pair is not specified	The number of activatable licenses available. See note about license group counts.
maxActOd	No	Integer. Default of 0 if this key-value pair is not specified	The maximum value allowed for availActOd. Set equal to availActOd to limit fraudulent attempts to add extra licenses.
availActOd	No	Integer. Default of 0 if this key-value pair is not specified	The number of overdraft activatable licenses available. See note about license group counts.
maxHyb	No	Integer. Default of 0 if this key-value pair is not specified	The maximum value allowed for avalHyb. Set equal to availHyb to limit fraudulent attempts to add extra licenses.
availHyb	No	Integer. Default of 0 if this key-value pair is not specified	The number of hybrid licenses available. See note about license group counts.
maxHybOd	No	Integer. Default of 0 if this key-value pair is not specified	The maximum value allowed for availHybOd. Set equal to availHybOd to limit fraudulent attempts to add extra licenses.
availHybOd	No	Integer. Default of 0 if this key-value pair is not specified	The number of overdraft hybrid licenses available. See note about license group counts.

Table A-19 • License Server Fulfillment Record Description

Key Name	Required	Value Format	Description
maxOdDuration	No	Integer.	The maximum duration in seconds for an overdraft license provided by activation. Licenses can be provided by activation from activatable or hybrid counts. See note.
maxConc	No	Integer. Default of 0 if this key-value pair is not specified	The maximum value allowed for availConc. Set equal to availConc to limit fraudulent attempts to add extra licenses.
availConc	No	Integer. Default of 0 if this key-value pair is not specified	The number of concurrent licenses available. See note about license group counts.
maxConcOd	No	Integer Default of 0 if this key-value pair is not specified	The maximum value allowed for availConcOd. Set equal to availConcOd to limit fraudulent attempts to add extra licenses.
availConcOd	No	Integer Default of 0 if this key-value pair is not specified	The number of overdraft concurrent licenses available. See note about license group counts.
hops	No	Integer Default of 0 if this key-value pair is not specified	How many additional entries can be added to the activation server chain. That is how many times this fulfillment can be transferred. See note.
actServerChain	Yes	Alphanumeric Format is: *Server_URL*[|*Server_URL*]	Activation server chain—the list of servers through which this fulfillment record has been supplied. The chain is in reverse order so that the first entry is the last server through which this fulfillment record passed. See note.
fulfillmentIdChain	Yes	Alphanumeric Format is: *fulfillmentId*[|*fulfillmentId*]tcb1	Fulfillment identity chain—the list of fulfillment identities that have been used for this fulfillment record as it moved through the activation server chain. See actServerChain and note.

Table A-19 • License Server Fulfillment Record Description

Key Name	Required	Value Format	Description
repairsToServe	No	Integer Default of 0 if this key-value pair is not specified	Number of repairs that this server performs for activations supplied from this fulfillment record. See note.
features	Yes	For details, see Chapter 15, "Defining Feature Definition Lines for a Fulfillment Record."	Feature definition lines. Each line gives license rights for a single feature. The feature definition lines must be signed using the hostid specified in ServerHostID.

Note • *The following is additional information about fulfillment record key-value pairs:*

1. **Key Names**—*Are case sensitive.*
2. **flexFulfillmentVersion**—*The license server will reject a fulfillment record with a flexFulfillmentVersion that is greater than the latest version it has information about.*
3. **License group counts**—*Are described in "Activation on License Servers" in the section in Fulfillment Records on License Server. Although none of the key-value pairs that define license group counts are mandatory, there must be at least one entry with a positive value.*
4. **maxOdDuration**—*Values calculated using this value (current time + maxOdDuration) are rounded up to the next whole number of days. For details see Activation on License Servers.*
5. **hops**—*Are only used when transferring fulfillment record between license servers. The hops element is not required to use a fulfillment record to supply activatable licenses to a FlexEnabled client.*
6. **actServerChain**—*When creating a fulfillment record, put in the URL of the server that is supplying this fulfillment record. Then the fulfillment record can be returned to the server. When creating a fulfillment record that will be supplied by a manual response generator, put in text to indicate that this was the source of this fulfillment record.*
7. **fulfillmentIdChain**—*When creating a fulfillment record, put in a value that indicates this is the source fulfillment identity. As the fulfillment is moved around by activation transactions a new fulfillment identity is created for each activation transaction. The sequence of these fulfillment identities is recorded in the fulfillment ID chain.*
8. **repairsToServe**—*This count can also be transferred, in part or whole, to another server when the fulfillment is transferred.*

Example Fulfillment Record

Example fulfillment records are provided in the toolkit within ASRs. For a list and description of these ASRs, see Sample ASRs in the Toolkit in B_ActivationSpecificationRecord.xsd

Fulfillment Record for a Trial License

The following fulfillment record is extracted from the `1day_student.asr` example ASR included in the toolkit.

```
<FulfillmentRecord>
   <FulfillmentRecordHeader>
      <VersionNumber>1</VersionNumber>
   </FulfillmentRecordHeader>
   <FulfillmentRecordData>
      <FulfillmentId>FIL-EZCALC-S1</FulfillmentId>
         <FulfillmentDictionary>
            <Entries>
               <Entry>
                  <Key>fulfillmentType</Key>
                  <Value>1</Value>
               </Entry>
               <Entry>
                  <Key>expiry</Key>
                  <Value>14-dec-2018</Value>
               </Entry>
               <Entry>
                  <Key>serverHostID</Key>
                  <Value>FLEXID=9-a6300155</Value>
               </Entry>
               <Entry>
                  <Key>servOrLocal</Key>
                  <Value>2</Value>
               </Entry>
               <Entry>
                  <Key>productId</Key>
                  <Value>EZCALC-S</Value>
               </Entry>
               <Entry>
                  <Key>entitlementId</Key>
                  <Value>ENTL-EZCALC-S1</Value>
               </Entry>
               <Entry>
                  <Key>availAct</Key>
                  <Value>1</Value>
               </Entry>
               <Entry>
                  <Key>features</Key>
                  <Value>INCREMENT ADD demo 1.0 31-dec-2020 1 SIGN="00ED 3E87 8BF0 4261 F9B3 \
                     64CE 960A FB00 37A1 44C3 5A57 3B38 1134 B3A2 E422"
                      INCREMENT SUBTRACT demo 1.0 31-dec-2020 1 SIGN="0057 2675 F0F7 6B0F 9CC0 \
                    8E37 7F64 C000 6BF5 9A9E 4B98 D8C1 241C BD02 7DAC"
                      INCREMENT MULTIPLY demo 1.0 31-dec-2020 1 SIGN="00B5 6771 ADD9 990B B2B4 \
                    13B3 BE64 8600 553E 439B 7348 014C 40E4 B858 B643"
                      INCREMENT DIVIDE demo 1.0 31-dec-2020 1 SIGN="00D5 6E90 4493 943C 2D0A \
                    D6D7 FBC1 F700 9B36 9211 642F 4D49 F089 C871 9AC0"
                  </Value>
               </Entry>
            </Entries>
         </FulfillmentDictionary>
   </FulfillmentRecordData>
</FulfillmentRecord>
```

The key elements in this fulfillment record are:

- `<FulfillmentId>FIL-EZCALC-S1</FulfillmentId>`—Identifies this fulfillment record as `FIL-EZCALC-S1`.

- `<Key>fulfillmentType<Key> <Value>1</Value>`—This fulfillment record is for a 'trial' license, that is it is installed from an ASR using local activation and provides a temporary license that allows the end user to trial the product.
- `<Key>expiry</Key> <Value>14-dec-2018</Value>`—The licenses for all the features included within this fulfillment record expire on December 14, 2018. This overrides the expiration date in the activated features lines. For example, feature `ADD` expires on December 14, 2018 rather than December 31, 2020.
- `<Key>serverHostID</Key> <Value>FLEXID=9-a6300155</Value>`—The SERVER line used when signing the feature definition lines included a hostid of `FLEXID=9-a6300155`.
- `<Key>servOrLocal</Key> <Value>2</Value>`—This fulfillment record is for use on a FlexEnabled client.
- `<Key>productId</Key> <Value>EZCALC-S</Value>`—Identifies that this fulfillment record is for product `EZCALC-S`.
- `<Key>entitlementId</Key> <Value>ENTL-EZCALC-S1</Value>`—The license rights acquired using this fulfillment record within an ASR are identified using `ENTL-EZCALC-S1`.
- `<Key>features</Key> <Value> </Value>`—Details of the features that are licensed by this fulfillment record. Four features are licensed: `ADD`, `SUBTRACT`, `MULTIPLY`, and `DIVIDE`. Some of the feature definition line positional fields are not used as they would be in a feature line in a license file. So the feature definition line:

  `INCREMENT ADD demo 1.0 31-dec-2020 1 SIGN="00ED ....”`

  when in this fulfillment record results in an uncounted license for the `ADD` feature that expires on the 14th December 2018 (the value defined by the expiry property).

Fulfillment Record for a Concurrent License

The following is a typical fulfillment record that provides concurrent licenses. It illustrates a fulfillment record that is in use, meaning that it has been transferred between servers and has been used to serve activatable licenses.

```
<FulfillmentRecord>
   <FulfillmentRecordHeader>
      <VersionNumber>1</VersionNumber>
   </FulfillmentRecordHeader>
   <FulfillmentRecordData>
      <FulfillmentId>FIL-145-USB1-2</FulfillmentId>
         <FulfillmentDictionary>
            <Entries>
               <Entry>
                  <Key>expiry</Key>
                  <Value>14-dec-2018</Value>
               </Entry>
               <Entry>
                  <Key>actServerChain</Key>
                  <Value>40000@myserver|Manual</Value>
               </Entry>
               <Entry>
                  <Key>fulfillmentIdChain</Key>
                  <Value>FIL-145-USB1|FIL-145</Value>
               </Entry>
               <Entry>
                  <Key>serverHostID</Key>
                  <Value>FLEXID=9-a6300155</Value>
               </Entry>
               <Entry>
```

```
                <Key>servOrLocal</Key>
                <Value>1</Value>
            </Entry>
            <Entry>
                <Key>productId</Key>
                <Value>EZCALC-A</Value>
            </Entry>
            <Entry>
                <Key>entitlementId</Key>
                <Value>ENTL-EZCALC-AS1</Value>
            </Entry>
            <Entry>
                <Key>hops</Key>
                <Value>2</Value>
            </Entry>
            <Entry>
                <Key>repairsToServe</Key>
                <Value>5</Value>
            </Entry>
            <Entry>
                <Key>maxAct</Key>
                <Value>10</Value>
            </Entry>
            <Entry>
                <Key>availAct</Key>
                <Value>5</Value>
            </Entry>
                <Key>maxHyb</Key>
                <Value>10</Value>
            </Entry>
            <Entry>
                <Key>availHyb</Key>
                <Value>10</Value>
            </Entry>
            <Entry>
                <Key>features</Key>
                <Value>INCREMENT ADD demo 1.0 31-dec-2020 1 SIGN="00ED 3E87 8BF0 4261 F9B3 \
              64CE 960A FB00 37A1 44C3 5A57 3B38 1134 B3A2 E422"
                INCREMENT SUBTRACT demo 1.0 31-dec-2020 1 SIGN="0057 2675 F0F7 6B0F 9CC0 8\
              E37 7F64 C000 6BF5 9A9E 4B98 D8C1 241C BD02 7DAC"
                INCREMENT MULTIPLY demo 1.0 31-dec-2020 1 SIGN="00B5 6771 ADD9 990B B2B4 \
              13B3 BE64 8600 553E 439B 7348 014C 40E4 B858 B643"
                INCREMENT DIVIDE demo 1.0 31-dec-2020 1 SIGN="00D5 6E90 4493 943C 2D0A \
               D6D7 FBC1 F700 9B36 9211 642F 4D49 F089 C871 9AC0"
                INCREMENT AVG demo 1.0 31-dec-2020 1 SIGN="00F0 AF8F 1444 1C97 69CC BBD9 \
                66D5 0A00 AC61 125D F5B8 C947 4A1C DA5C 78E4"
                </Value>
            </Entry>
        </Entries>
    </FulfillmentDictionary>
  </FulfillmentRecordData>
</FulfillmentRecord>
```

The key elements in this fulfillment record are:

- `<FulfillmentId>FIL-E1</FulfillmentId>`—Identifies this fulfillment record as `FIL-E1`.

- `<Key>expiry</Key> <Value>14-dec-2018</Value>`—The licenses for all the features included within this fulfillment record expire on December 14, 2018. This overrides the expiration date in the activated features lines. For example, feature `ADD` expires on December 14, 2018 rather than December 31, 2020.
- `<Key>serverHostID</Key> <Value>FLEXID=9-a6300155</Value>`—The SERVER line used when signing the feature definition lines included a hostid of `FLEXID=9-a6300155`.
- `<Key>servOrLocal</Key> <Value>1</Value>`—This fulfillment record is for use on a license server.
- `<Key>productId</Key> <Value>EZCALC-A</Value>`—Identifies that this fulfillment record is for product `EZCALC-A`.
- `<Key>entitlementId</Key> <Value>ENTL-EZCALC-AS1</Value>`—The license rights in this fulfillment record are identified using `ENTL-EZCALC-AS1`. This entitlement ID must be used in activation requests to the license server. The license server identifies the fulfillment record that matches an activation request using this value.
- `<FulfillmentId>FIL-145-USB1-2</FulfillmentId>`
  `<Key>actServerChain</Key> <Value>40000@server.com|Manual</Value>`
  `<Key>fulfillmentIdChain</Key> <Value>FIL-145-USB1|FIL-145</Value>`—This fulfillment record was originally supplied from a manual response generator where it was identified as `FIL-145` within the activation response XML document. It was used to generate an activation in response to a request from server `40000@smyerver`. The fulfillment identity `FIL-145-USB1` was inserted manually to identify this activation transaction. As the result of an activation request from another license server, the license rights in this fulfillment record (`FIL-145_USB1-2`) were transferred and the entries for `actServerChain` and `fulfillmentIdChain` were updated.
- `<Key>hops</Key> <Value>2</Value>`—Two transfers between license servers are allowed for this fulfillment record. For details, see the section FulfillmentDictionary Data in Server-side Trusted Storage and refer to Chapter 2, “Transactions to a License Server.”
- `<Key>maxAct</Key> <Value>10</Value>`—The license server does not increase the value of `availAct` to greater than 10.
- `<Key>availAct</Key> <Value>5</Value>`—This fulfillment record can provide five activations to FlexEnabled clients or these can be transferred to another server. From this and the value of `10` for `maxAct`, it can be concluded that `five` activations have been served from this fulfillment record. For details, see the section FulfillmentDictionary Data in Server-side Trusted Storage and refer to Chapter 2, “Transactions to a License Server.”
- `<Key>maxHyb</Key> <Value>10</Value>`—The license server does not increase the value of `availHyb` to greater than `10`.
- `<Key>availHyb</Key> <Value>10</Value>`—The current hybrid license group count is `10`. These are all available firstly for serving as concurrent licenses. As the counts within the feature lines are all `1`, then this fulfillment record adds a count of `10` to the concurrent pool for each feature. For details, see the section FulfillmentDictionary Data in Server-side Trusted Storage and refer to Chapter 2, “Transactions to a License Server.”
- `<Key>features</Key> <Value> </Value>`—Details of the features that are licensed by this fulfillment record. Five features are licensed: `ADD`, `SUBTRACT`, `MULTIPLY`, `DIVIDE`, and `AVG`.

M1_Dictionary.xsd

This schema document is used to define the structure of all data dictionaries.

See Also
M2_NonEmptyString.xsd

M2_NonEmptyString.xsd

This schema document is referenced by other schemas.

M5_ParameterHeaderType.xsd

This schema document is referenced by other schemas. It defines the schema version number.

See Also
M6_ParameterHeaderType.xsd
Identifying Schema Versions

M5_VersionNumberType.xsd

This schema document is referenced by other schemas. It defines the schema version number as a header type.

See Also
Identifying Schema Versions

M6_ParameterHeaderType.xsd

This schema document is referenced by other schemas. It defines a header type that references the schema version number.

See Also
M5_ParameterHeaderType.xsd
Identifying Schema Versions

M7_TrustedIdType.xsd

This schema document is referenced by other schemas. It defines a type for the trusted ID.

M8_UniqueMachineNumbersType.xsd

This schema document is referenced by other schemas. It defines a type for the UMN origin data.

M9_PlatformType.xsd

This schema document is referenced by other schemas. It defines a type for the platform data.

M10_ConfigTypes.xsd

This schema document is referenced by other schemas. It defines types for configurations sent in a response by a server, or for configurations sent in an ASR.

M11_TrustFlags.xsd

M12_BreakInfo.xsd

M13_RightsId.xsd

M14_ResponseActionReference.xsd

M15_ResponseDenyReason_NotUsed.xsd

M16_TypeValueComment.xsd

O_ConfigPhaseOne.xsd

This schema document describes the structure of the XML file used as input to `tcencrypt`. It defines a publisher specific Trusted Configuration and contains the following information:

- The seeds for creating transaction keys, used to secure the contents of activation transactions that place data in this trusted section
- Anchors to be used
- Binding used to lock the trusted storage to the system
- Optionally, windback tolerance—the time difference that results in windback being reported.

It is used to create other documents that are used in activation transactions that result in the creation of trusted storage. For a description of the data required in this document, see Description. For a description of how this document is used, see Usage.

Description

The document is structured so that data used to identify the Trusted Configuration is listed first, followed by the seeds used to create a unique transaction key for activation transactions that use this trusted section of trusted storage. Separate configurations are defined for server-side trusted storage (on license servers) and client-side trusted storage (on FlexEnabled clients). Normally the highest level of anchoring and binding is used for server systems.

Table A-20 • Trusted Configuration Structure

Element Name	Required	Format	Description
PublisherId	Yes	Integer	Flexera supplied publisher ID
Id	Yes	M7_TrustedIdType.xsd Integer in range 0 to 255	Trusted ID used to identify this trusted section. This must be unique for this publisher.
Revision	Yes	Alphanumeric string	Identifies the version of this Trusted Configuration. Sample configuration, request, response, and ASR files in the toolkit can show one of these values: ● **DefaultOne**—Trusted Configuration generated by FlexNet Publisher 11.10 (and based on binding identities available with 11.10). ● **NumberOne**—Trusted Configuration generated by a pre-11.10 version of FlexNet Publisher (and based on binding identities available in this earlier version).
Seeds	Yes	A set of 8 entries: each entry is a hexidecimal string of eight characters	The set of eight seeds selected by the publisher to create a unique transaction key.

Table A-20 • Trusted Configuration Structure

Element Name	Required	Format	Description
TransactionKeyInfo	Yes	Set as either: Long_1 Short_21 Short_22 Short_23 Short_24 Short_25 Short_26	The length of the transaction key created by tcencrypt. Long_1 creates a long transaction key for use with XML requests and responses. Short_21, Short_22, Short_23, Short_24, Short_25, and Short_26 create a short transaction key for use with request and response short codes. When creating a Trusted Configuration for both XML and short-code transactions, both types of keys can be specified. When using short-code transactions, the value must correlate to the signature strength used in the ASR. See Transaction Key Length for more information about this dependency.
Server	Yes	See Platforms, Anchoring, Binding and Timing for the formats of these elements.	The anchoring, binding and windback tolerance settings for this trusted section when it is created on the license server.
Client	Yes	See Platforms, Anchoring, Binding and Timing for the formats of these elements.	The anchoring, binding and windback tolerance settings for this trusted section when it is created on a FlexEnabled client.

Transaction Key Length

The TransactionKeyInfo element defines the lengths of the transaction key created by tcencrypt. The following XML example demonstrates how this is defined.

```
<TransactionKeyInfo>
    <KeysRequired>Long_1</KeysRequired>
    <KeysRequired>Short_21</KeysRequired>
</TransactionKeyInfo>
```

In this example,

"Long_1" specifies the transaction keys for XML request and response messages.

"Short_21" specifies the transaction keys for short-code request and response messages.

When using short-code transactions, the value defined here must be compatible with the signature strength defined in the ASR. The second value in this example, Short_21, must correlate to the Scheme Id used by ASRs for this trusted section.

The following table describes the transaction key types that you can define and the Short-Code Scheme Id that each supports. For more information about defining Scheme Ids in an ASR, see ShortCodeData Element.

Table A-21 • Defining TransactionKeyInfo to Match Short-Code Scheme Id

TransactionKeyInfo	Short-Code Scheme Id
Short_21	SCS_BASIC_42
Short_21	SCS_COMPAT_1
Short_22	SCS_COMPAT_2
Short_23	SCS_COMPAT_3
Short_24	SCS_COMPAT_4
Short_25	SCS_COMPAT_5
Short_26	SCS_COMPAT_6

Platforms

The anchoring, binding and windback tolerance settings can be applied to all operating systems or can be separately specified for individual operating systems. The following table lists the strings that must be used to identify the platforms. These strings are not case sensitive. They are used in the <Type> element both when setting anchors to be disabled and when setting binding identities to be disabled.

Table A-22 • Platform Information

Setting	Description
default	Apply these settings to all platforms not individually specified
windows	Windows OS
solaris	Sun Solaris
linux	Linux
mac	OS X
HPUX	HP-UX
AIX	AIX

Anchoring

This section is used to change the default configuration for both Publisher Anchors and Trial Anchors. This element specifies only those anchoring technologies that are *disabled*, meaning that they are not to be used. For more information about Anchoring, see the section titled Anchoring in the Chapter 3, "Configuring Security for Trusted Storage."

The following strings are used to identify the anchoring technologies to be disabled:

Table A-23 • Anchor-Identifying Strings

String	Description
TrackZero	Hardware-based anchor (Windows only)
Registry	Registry-based anchor (Windows only)
FileBased	An anchor based on a file (Not available on Windows)
OSXFileBased	An anchor based on the file system (OS X only)

Binding

Binding is used to prevent the license exploitation that occurs when licenses are copied to another system. This is implemented by storing identities obtained from the system in trusted storage. Each time trusted storage is accessed these *binding identities* are checked and compared with the values held in trusted storage. The binding identities are operating-system dependent.

Table A-24 • Binding Identities

Binding Identity	Description	Environment	Deal breaker
SYSTEM	Information obtained from the system	Physical	No
HARDDISK	Hard disk or file system information	Physical	Yes if *both* the `HARDDISK` and `TPMID` binding identities are changed. No if TPM is not used.
DISPLAY	Display or graphics card information	Physical	No
BIOS	(Available on Windows only) Manufacturer name and version	Physical	No
CPU	Information about the CPU(s)	Physical	No
MEMORY	RAM size	Physical	No
ETHERNET	MAC address	Virtual and Physical	Yes if virtual.

Table A-24 • Binding Identities

Binding Identity	Description	Environment	Deal breaker
GENID	A hash of the virtual machine Generation ID	Virtual	Yes
VMID	A hash of the universally unique ID (UUID) used to bind to virtual machines. ***Note*** • *The UUID is the same value as the* `VM_UUID` *hostid described in the Programming Reference for License File–Based Licensing.*	Virtual	Yes
TPMID	A hash of the TPM hostid.	Physical	Yes if *both* the `HARDDISK` and `TPMID` binding identities are changed.

FLEXIBLE Binding

The default for FLEXIBLE binding is to use all the built-in bindings that are available on an operating system. However, users who are not using FlexNet Operations can specify the `<Disable>` or `<Enable>` element along with `<HID>` to identify explicitly which binding identities to use or not to use for a given operating system:

- Use `<Disable>` to specify the binding identities you do not want to include in the bind. All identities except for those you list for the `<HID>` element are used in the bind.
- Use `<Enable>` to specify the identities you want to include in the bind. Only the identities you list for the `<HID>` element are used in the bind.

Note • *The* `<Enable>` *element is recommended for use with only the VMID identity since VMID-only binding is the preferred method for binding trusted storage to a virtual machine. For all other trust-storage binding setups (for physical machines),* `<Enable>` *is not recommended since the binding should use as many identities as possible.*

Use the binding identities listed in Table A-24 as strings to specify that each binding identity be disabled or enabled.

Note • *You cannot specify both* `<Enable>` *and* `<Disable>` *within a given flexible binding configuration.*

The following example shows a binding configuration in which default binding is used for all platforms except Windows. On the Windows platform, all available binding identities are used except for display or graphic-card information.

```
<Binding>
  <Platform>
    <Type>DEFAULT</Type>
```

```
    <Mode>FLEXIBLE</Mode>
  </Platform>

  <Platform>
    <Type>windows</Type>
    <Mode>FLEXIBLE</Mode>
    <Disable>
      <HID>DISPLAY</HID>
    </Disable>
  </Platform>
</Binding>
```

Timing

The Timing element enables the windback tolerance to be specified. *Windback* is when the system clock has been wound back to an earlier date and time to try to obtain an illegal extension to the license. It is detected by placing a time-stamp in trusted storage when trusted storage is accessed. When a time-limited license is used, the current clock time is compared with the time-stamp in the fulfillment record. When the current time is earlier than the time-stamp, windback may have occurred. The *windback tolerance* is the difference between these times that is acceptable. When the current time is earlier than the time-stamp by a greater amount than the windback tolerance, windback is reported.

When windback tolerance is not specified (the Trusted Configuration file does not contain a Timing element), the default value of 24 hours is used.

Some windback tolerance is normally allowed so that resetting the clock, for example when moving between time zones, does not trigger windback detection.

Windback tolerance can be set for the following:

- Platform—A different windback tolerance can be specified per platform.
- FlexEnabled clients—A different windback tolerance can be specified per trusted section of client-side trusted storage.
- License servers—A different windback tolerance can be specified per trusted section of server-side trusted storage.

The lowest level elements of the schema that define the setting for windback tolerance are described in the following table.

Table A-25 • Windback Tolerance Configuration Elements

Element Name	Mandatory	Format	Description
Units	Yes if Timing element included	String with one value: Hours	The units for Value. This currently can only be set as hours.
Value	Yes if Timing element included	Integer in range 0 to 1193046	The windback tolerance in hours. Setting a value of zero hours results in windback being detected when the current time is earlier than the time-stamp by any detectable amount.

Usage

The Trusted Configuration file is not used directly. It must be processed and used in the following way:

- Create a Trusted Configuration file based on this schema.
- Encrypt the Trusted Configuration file using `tcencrypt`. The Trusted Configuration file is required as input to `tcencrypt`.
- Use `xmlsign` to add signatures to the `P1_ConfigPhaseTwo` document output by `tcencrypt`.
- When you create an ASR using `asrgen`, use the output file from `tcencrypt` as input to `asrgen`.
- When you create an ASR without using `asrgen`, extract the `<ConfigData>` element from the `<AsrConfig>` element in the signed `P1_ConfigPhaseTwo` document; include the `<ConfigData>` element in the ASR (B_ActivationSpecificationRecord.xsd).
- When you want to use the Trusted Configuration as the *prepped* Trusted Configuration deployed by the activation utility to create an initial trusted section, you must point to this file in the `preptool` configuration file used to prepare the activation utility. For more information, see the chapter "First Build of the FlexNet Publisher Toolkit" in the *Development Environment Guide*.
- When you create response files for the Manual Response Generator that contain Trusted Configurations, extract the relevant sections from the signed and include these sections in the Create Config Response document. To help you determine which elements to extract and then include in the response, see the sample documents W4C_CreateConfigResponseClientParameter.xsd and W4S_CreateConfigResponseParameter.xsd. For a description of the Trusted Configuration output file, see P1_ConfigPhaseTwo.xsd.
- When you create key files for the Manual Response Generator, tcencrypt outputs the required file, P2_ConfigKeyOut.xsd.

Example Contents of the Input Trusted Configuration File

The following example is the configuration for Trusted Section 1 of publisher 256's trusted storage. It can be used both for transactions that use XML documents and for transactions that use request and response codes. The settings for servers include the following:

- Anchoring is set to the default (maximum security)
- Binding is set to the default (all available bindings are used)

The settings for FlexEnabled clients are:

- The track zero anchor is disabled
- The binding is set on Windows platforms to exclude the binding to the display information
- Default binding settings are used for all other operating systems

```
<?xml version="1.0" encoding="utf-8" ?>
<TrustedConfigInformation xmlns="http://www.macrovision.com/fnp/2004/11/trusted">
    <PublisherId>256</PublisherId>
    <Id>1</Id>
    <Revision>NumberOne</Revision>
    <Seeds>
        <SeedPart>129C6257</SeedPart>
        <SeedPart>94724A4C</SeedPart>
```

```
        <SeedPart>A2C5E526</SeedPart>
        <SeedPart>A6AD539B</SeedPart>
        <SeedPart>39C3EAA0</SeedPart>
        <SeedPart>195B47EA</SeedPart>
        <SeedPart>AA11C2B3</SeedPart>
        <SeedPart>A4E93C2C</SeedPart>
    </Seeds>
    <TransactionKeyInfo>
        <KeysRequired>Long_1</KeysRequired>
        <KeysRequired>Short_21</KeysRequired>
    </TransactionKeyInfo>
    <Server>
        <Anchoring>
        </Anchoring>
        <Binding>
            <Platform>
                <Type>DEFAULT</Type>
                <Mode>FLEXIBLE</Mode>
            </Platform>
        </Binding>
    </Server>
    <Client>
        <Anchoring>
             <Platform>
                 <Type>windows</Type>
                 <Disable>
                     <Anchor>TrackZero</Anchor>
                 </Disable>
             </Platform>
        </Anchoring>
        <Binding>
            <Platform>
                <Type>DEFAULT</Type>
                <Mode>FLEXIBLE</Mode>
            </Platform>
            <Platform>
                <Type>windows</Type>
                <Mode>FLEXIBLE</Mode>
                <Disable>
                    <HID>DISPLAY</HID>
                </Disable>
            </Platform>
        </Binding>
    </Client>
</TrustedConfigInformation>
```

Examples Shipped in Toolkit

The FlexNet Publisher Licensing Toolkit contains the following examples of the `tcencrypt` input file for trusted storage configuration:

```
<platform_dir>\publisher\DemoTrustedConfigIn.xml
```

See Also
P1_ConfigPhaseTwo.xsd
P2_ConfigKeyOut.xsd

P1_ConfigPhaseTwo.xsd

This document describes the XML structure of one of the output files generated by `tcencrypt`. Normally more processing is performed on it rather than it being used directly. For a description of the input XML file to `tcencrypt`, see O_ConfigPhaseOne.xsd. For a description of the process that generates this file, see Usage in O_ConfigPhaseOne.xsd.

Examples Shipped in Toolkit

The FlexNet Publisher Licensing Toolkit contains the following examples of the `tcencrypt` output file for trusted storage configuration.

For Virtual and Non-Virtual Platforms

The toolkit contains the following examples:

- `<platform_dir>\publisher\DemoTrustedConfigOut.xml`
- `<platform_dir>\publisher\xml\Version6\O_ConfigPhaseTwo.xml`

RevisionType

This the RevisionType element in this file defines how the trusted storage configuration is used when it is included in a config response to update an existing trusted section.

The priority for updating a Trusted Configuration is:

- SPE—Highest priority, can only be updated by a configuration of RevisionType SPE
- SRV—Can be updated by configurations of RevisionType SPE or SRV
- ASR—Lowest priority, can be updated by configurations of RevisionType SPE, SRV, or ASR

The `RevisionType` is set to SPE by specifying the `-s` switch when using `tcencrypt`.

The `RevisionType` is set to SRV when using `tcencrypt` without the `-s` switch.

The `RevisionType` is set to ASR when an ASR is created using `asrgen`. This ensures that the Trusted Configuration specified in an ASR does not overwrite any existing configuration for this trusted section that has been set using a transaction.

P2_ConfigKeyOut.xsd

This file describes the structure of one of the output files from `tcencrypt`. It is one of the XML documents required when using the manual response generator. For details, see XML Documents Required.

The FlexNet Publisher Licensing Toolkit contains an example key file located in the `<platform_dir>\publisher\xml` folder (`DemoKeyout.xml`).

S_ActivationSettings.xsd

This document defines the structure of the `publisher.xml` file that contains security data required for all applications that use trusted storage. This document is a publisher specific file that is created by Flexera and used within the toolkit. The seeds within the document must be supplied by you and provide publisher specific security.

This file also defines the structure of `sdt_publisher.xml`, which contains the security data required for applications that both use license-file licensing (and were built using a FlexNet Publisher trusted-storage licensing toolkit) and contain SDTs.

See Also
preptool
Implementing Secure Data Types (SDTs) in the *Development Environment Guide*

W1_CreateActivationResponseParameter.xsd

This schema document defines the structure of XML file containing the activation response parameter data required when generating an activation response. It is required as input to the manual response generator when generating activation responses.

Description

This document contains the input required when generating a response to an activation request. It contains the following information:

- **Version number**—identifies the format of the activation response created using this document.
- **Trusted ID**—identifies the trusted section into which the fulfillment record is to be placed.
- **Fulfillment record**—contains the license rights that are to be placed into trusted storage.
- **Vendor data**—optional, contains publisher specific data sent from the publisher's activation server.

The following table shows the top level structure of the create activation response document. The format and description columns indicate the format of lowest level elements that are the contents of the top level elements.

Table A-26 • Create Activation Response Structure

Element Name	Mandatory	Format	Description
Header	Yes	M6_ParameterHeaderType.xsd	Version number: identifies the document version. The activation response document created using this document will be this version.
ConfigData	Yes	M7_TrustedIdType.xsd	Trusted ID: identifies the trusted section. This is a positive integer from 0–255.
ResponseData	Yes	See ResponseData for details of the format.	Fulfillment record that contains the license rights to be placed into trusted storage. Optionally, publisher (vendor) data sent during the activation transaction.

ResponseData

This element contains two elements the first of which is mandatory and the second optional:

- **FulfillmentRecord**—the fulfillment record that contains the license rights to be placed into trusted storage by an activation response that uses this document as input. This element is mandatory.
- **VendorDictionary**—optional data added to the activation response. This is for use by the activation utility. It is formatted as key-value pairs with no constraints on its contents. You should ensure that any data added is in a format known to the activation utility.

See Also
Response Files
I_FulfillmentRecord.xsd
Using the Vendor Dictionary to Send Custom Data

W2_CreateReturnResponseParameter.xsd

This schema document defines the structure of XML file containing the return response parameter data required when generating a return response. It is required as input to the manual response generator when generating return responses.

Description

This document contains the input required when generating a response to a return request. It contains the following information:

- *Version number*—identifies the format of the return response created using this document.
- *Vendor data*—optional, contains publisher specific data sent from the publisher's activation server.

The following table shows the top level structure of the create return response document. The format and description columns indicate the format of lowest level elements that are the contents of the top level elements.

Table A-27 • Create Activation Response Structure

Element Name	Mandatory	Format	Description
Header	Yes	M6_ParameterHeaderType.xsd	Version number: identifies the document version. The activation response document created using this document will be this version.
ResponseData	No	VendorDictionary—key-value pairs	Publisher (vendor) data sent in the return response.

See Also
Response Files
Using the Vendor Dictionary to Send Custom Data in Application Activation Utility
Using the Vendor Dictionary for Custom Data in Server Activation Utility

W3_CreateRepairResponseParameter.xsd

This schema document defines the structure of XML file containing the repair response parameter data required when generating a repair response. It is required as input to the manual response generator when generating repair responses.

The following schema and description references version 2 of this document. The toolkit also contains the version 1 schema, which has a different format. For information on versions, see Identifying Schema Versions.

Description

This document contains the input required when generating a response to a return request. It contains the following information:

- **Version number**—identifies the format of the repair response created using this document.
- **Vendor data**—optional, contains publisher-specific data sent from the publisher's activation server.

The following table shows the top level structure of the create repair response document. The format and description columns indicate the format of lowest level elements that are the contents of the top level elements.

Table A-28 • Create Activation Response Structure

Element Name	Mandatory	Format	Description
Header	Yes	M6_ParameterHeaderType.xsd	Version number: identifies the document version. The activation response document created using this document will be this version.
ResponseData	No	VendorDictionary—key-value pairs	Publisher (vendor) data sent in the repair response.

See Also
Response Files
Using the Vendor Dictionary to Send Custom Data in Application Activation Utility
Using the Vendor Dictionary for Custom Data in Server Activation Utility

W4C_CreateConfigResponseClientParameter.xsd

This schema document defines the structure of the XML file used to create a Trusted Configuration response sent to a client. It contains client Trusted Configuration created in an XML document that conforms to the schema O_ConfigPhaseOne.xsd.

The XML file that this schema defines is required as input to the manual response generator when a new client Trusted Configuration is required or the existing Trusted Configuration requires updating.

This document is structured so that there can be multiple client Trusted Configurations. The structure of a Trusted Configuration, ClientConfig, is described in Table A-29. It is extracted from the signed and encrypted version of a Trusted Configuration source document. See O_ConfigPhaseOne.xsd for information about how to create the source document from which the ClientConfig structures can be cut.

Table A-29 • ClientConfig Structure

Element Name	Mandatory	Format	Description
Id	Yes	Integer	Trusted ID used to uniquely identify this trusted section. The value is in the range of 0–255.
Revision	Yes	Alphanumeric	Identifies the revision of this Trusted Configuration.
RevisionType	Yes	• SPE = cannot be overwritten by a client config of revision type SRV • SRV = can be overwritten by client configs of revision type SRV or SPE	Defines protection given to this client config. SRV is the default. SPE can be specified when running tcencrypt to ensure that the client config is not overwritten.
Data	Yes	Alphanumeric	Encrypted version of the anchoring and binding settings for this trusted ID when created on a FlexEnabled client.
Signature	Yes	Hexadecimal	Ensures integrity of this structure. It is generated from all the other elements of this structure.

See Also
Response Files
O_ConfigPhaseOne.xsd
P1_ConfigPhaseTwo.xsd
M2_NonEmptyString.xsd

W4S_CreateConfigResponseParameter.xsd

This document is used to create a Trusted Configuration response sent to a server. It contains server and client Trusted Configurations created using a document that conforms to the schema O_ConfigPhaseOne.xsd.

The XML file that this schema defines is required as input to the manual response generator when a new server Trusted Configuration is required or the existing Trusted Configuration requires updating.

This document is structured so that there can be multiple Trusted Configurations. The top-level structure of a Trusted Configuration, ConfigData, is described in the following table (given here for reference only). You do not create it directly: it is extracted from the signed and encrypted version of a Trusted Configuration source document. See O_ConfigPhaseOne.xsd for information about how to create the source document used to create the signed and encrypted Trusted Configurations. The ConfigData structure is cut from the resulting Trusted Configuration file after signatures have been added to it using xmlsign.

ConfigData Element

The following table describes the ConfigData element:

Table A-30 • ConfigData Structure

Element Name	Mandatory	Format	Description
PublisherId	Yes	Integer	Flexera supplied publisher ID
ServerConfig	No	See ServerConfig Element	Encrypted trusted storage configuration for use on a server.
ClientConfig	Yes	See ClientConfig Element	Encrypted trusted storage configuration for use on a FlexEnabled client.
Signature	Yes	Hexadecimal string	Ensures integrity of this structure. It is generated from all the other elements of this structure.

ServerConfig Element

The following table describes the ServerConfig element:

Table A-31 • ServerConfig Structure

Element Name	Mandatory	Format	Description
Id	Yes	Integer in range 0–255	Trusted ID used to uniquely identify this of trusted storage.
Revision	Yes	Alphanumeric string	Identifies the revision of this Trusted Configuration.
RevisionType	Yes	• SPE = cannot be overwritten by a server config of revision type SRV • SRV = can be overwritten by server configs of revision type SRV or SPE	Defines protection given to this server config. SRV is the default. SPE can be specified when running tcencrypt to ensure that the server config is not overwritten.
Data	Yes	Alphanumeric string	Encrypted version of the anchoring and binding settings for this trusted ID when created on a license server.

ClientConfig Element

The following table describes the ClientConfig element:

Table A-32 • ClientConfig Structure

Element Name	Mandatory	Format	Description
Id	Yes	Integer in range 0–255	Trusted ID used to uniquely identify this trusted section.
Revision	Yes	Alphanumeric string	Identifies the revision of this Trusted Configuration.
RevisionType	Yes	• SPE = cannot be overwritten by a client config of revision type SRV • SRV = can be overwritten by client configs of revision type SRV or SPE	Defines protection given to this client config. SRV is the default. SPE can be specified when running tcencrypt to ensure that the client config is not overwritten.
Data	Yes	Alphanumeric string	Encrypted version of the anchoring and binding settings for this trusted ID when created on a FlexEnabled client.
Signature	Yes	Hexadecimal string	Ensures integrity of this structure. It is generated from all the other elements of this structure.

See Also
Response Files
O_ConfigPhaseOne.xsd
P1_ConfigPhaseTwo.xsd
M2_NonEmptyString.xsd

W5_CreateFailureResponseParameter.xsd

This schema document describes the XML file that contains parameter data required when generating a failure response. It is required as input to the manual response generator when generating a failure response.

This document contains the input required when generating a failure response to any type of request. It contains the following information.

- The reason for the failure:
  - The failure was due to an error in processing the request.
  - The request was denied because it did not conform to the business rules.
- Optionally, an error code.

Table A-33 shows the lowest level of the structure of the failure response input document at the level where the input data is held.

Table A-33 • Failure Response Structure

Element Name	Mandatory	Format	Description
ResponseReason	Yes	• ERROR = failure due to an error in processing • DENIED = request denied	Reason that a failure response is being returned.
ErrorCode	No	Alphanumeric	Error code that is publisher defined.

See Also
Response Files
flxActCommonHandleGetLastResponseError
flxActCommonHandleGetLastResponseErrorString

W6_CreateResponseCodeParameter.xsd

This schema document defines the XML file that contains parameter data required when generating a response code. It is required as input to the manual response generator when generating a response code.

The XML file contains the input required when generating a response code. It contains the following information.

- The type of response code
- The values for variable and additional items included in the response code

The following table shows the structure of the document.

Table A-34 • Response Code Input Data Structure

Element Name	Mandatory	Format	Description
ResponseType	Yes	• ACTIVATION • RETURN • REPAIR • ERROR	Defines the type of response code. ACTIVATION, RETURN, and REPAIR indicate that the corresponding function is to be completed. An ERROR response code indicates that an error has occurred or the transaction is denied.
ResponseItem	No	See Response Item table	Defines a value for an item in the response code.

Table A-35 • Response Item

Element Name	Mandatory	Format	Description
Name	Yes	String The name of the item	These names are defined in the ASR. See SchemeId in the ShortCodeData Element section and Data That Can Be Overridden (or Defined) in Response Codes.
Value	Yes	String	The value for this item as inserted in the response code.

Note • *For activation and repair transactions, response items that are added to the default response code format using ResponseOverrides are inserted into the copy of the fulfillment record in trusted storage.*

See Also
ShortCodeData Element
Response Files

W7_CreateCompositeResponseParameter.xsd

This schema combines all response types into one schema for composite transactions. View this schema in `<platform_dir>\publisher\schemas\Version6`.

Glossary

Term	Meaning
activatable license	A license right in trusted storage that can be activated to a FlexEnabled client. See also license group.
activated feature line	See feature definition line.
activation	In the FlexNet Publisher Licensing Toolkit, this is a process used to load license rights into trusted storage. Different activation types include: ● programmatic activation ● manual activation ● short-code activation ● local activation In FlexNet Operations, this is a process used to generate license rights in either trusted storage and in license files.
Activation API	Functions available in the FlexNet Publisher Licensing Toolkit libraries that enable you to initiate transactions and manage license rights in trusted storage. The Activation API is used to develop the functionality in an activation utility.

Term	Meaning
activation client	The system that works with an activation server to receive license rights. This system can be either a license server or a FlexEnabled client. The activation client initiates an activation, repair, or return transaction with an activation server, using an activation utility. The following transaction types can be initiated by an activation client: ● activation transaction ● return transaction ● repair transaction
activation request	A message sent by an activation utility to an activation server that initiates an activation transaction.
activation response	A message sent from an activation server to an activation client, in response to an activation request. This message defines the license rights.
activation server	A system that responds to requests for license rights from an activation client. The activation server can be one of the following: ● a license server ● FlexNet Operations ● in a test environment, the Manual Response Generator (responsegen.exe) An activation server can also perform repair and return transactions.
activation specification record (ASR)	A signed XML file used with local activation and short-code activation that contains information about license rights, configuration settings for the publisher's trusted storage segment, and additional information required by the activation transaction.
activation transaction	A sequence of activation request and activation response messages, and related actions that result in license rights being loaded into trusted storage.
activation utility	A utility that initiates an activation transaction, repair transaction, or return transaction by sending a request to an activation server. This utility also processes the response received from the activation server. It uses the Activation API to perform these capabilities. It is written by the software publisher to meet specific business process requirements. This application can be either separate from or embedded in the FlexEnabled application.
anchor	A link between the system and the trusted storage that is used to detect whether trusted storage has been tampered with, deleted, or restored. Anchors are also used to ensure that information in trusted storage about trial licenses persists beyond the expiration date.
ASR file	See activation specification record (ASR).
binding	A capability used with trusted storage where one or more properties of a system, combined together into a unique identifier, are used to ensure that license rights held in trusted storage are not copied to another system.

Term	Meaning
borrowing	A capability available in FlexEnabled applications that enables a user to temporarily obtain a license in such a way that the license can be used locally for a limited period of time while the user is disconnected from the local network.
bulk entitlement	An entitlement created for an order of multiple copies of a single product or suite. Bulk entitlements are used for sales through reseller channels.
checkin	The transaction that a FlexEnabled application performs to return a license after it has been checked out. This operation is not the same as an activation transaction. FlexEnabled applications must perform a checkout and checkin transaction to use a license.
checkout	The transaction that a FlexEnabled application performs to request a license. This transaction is different than activating a license. FlexEnabled applications must perform a checkout and checkin transaction to use a license.
client system	A general term used to define any system that interacts with a server in a client-server relationship. When using the FlexNet Publisher Licensing Toolkit there are two types of clients: ● activation client ● FlexEnabled client
composite transaction	Available in FlexNet Publisher 11.8, composite transactions allow multiple actions to be combined into a single transaction with FlexNet Operations. Also called multi-action or V2 transactions. From FlexNet Publisher 11.14.0, composite transactions are also supported for all license servers.
concurrent license	A license that can be shared among users by allowing each user to check out and then return the license. These licenses are served to FlexEnabled applications from a license server. If all licenses are being used, an additional user cannot run the FlexEnabled application until one of the other users check in their license. When one user finishes using the license, another user can begin using it. Also called floating license or served license rights.
config response	A response to any request from an activation client that contains information about how the publisher's trusted storage segment must be configured (called the trusted storage segment configuration). This response is processed by the activation utility and results in a segment of trusted storage being created or re-configured for that publisher.
consumer	An individual or small business who purchases a license for their individual use. Compare with enterprise.
counted	A type of license right or license model that defines a multiple quantity of licenses and allows concurrent use of the licenses. This term is taken from the fact that the license server counts the number of licenses that are being used.
debug log file	A file used by the license server to record status and error messages that are useful for debugging the license server. Each license server can have one or more of these files.

Term	Meaning
demo license	See trial license.
duplicate grouping	A capability in the FlexNet Publisher Licensing Toolkit used to uniquely identify a single client (or client checkout request) for the purpose of counting the number of licenses that are used. Duplicate grouping defines a set of rules under which multiple checkout requests can share the same license(s).
embedded activation	See local activation.
encryption seed	A set of values defined by a software publisher that are used when creating a publisher-specific vendor daemon.
end-user	A person who uses a FlexEnabled application. This role is not the same as a license administrator.
end-user system	See client system.
end-user license administrator	See license administrator.
enterprise	An organization that purchases licenses on behalf of a number of individual users. Compare with consumer.
entitlement	A representation of the license model and product that a customer has bought. It is a definition only. The customer will need to fulfill the entitlement so that the license rights can be populated to license files or trusted storage. In FlexNet Operations, there are two types of entitlements: ● bulk entitlement ● simple entitlement Each entitlement is uniquely identified by an entitlement ID.
entitlement ID	In the FlexNet Publisher Licensing Toolkit, an attribute in trusted storage that stores a value used to identify the customer's entitlement. In FlexNet Operations, a value that uniquely identifies the entitlement.
expiring license	License rights granted to a customer for a limited duration. When the license rights have expired, the licenses can no longer be checked out.

Term	Meaning
feature	In the context of a software application, this is a single unit of capability. In the FlexNet Publisher Licensing Toolkit, this is an identifier used to associate a license model to a unit of capability in a software application. The software publisher defines which features in their software application map to a single feature in the FlexNet Publisher Licensing Toolkit. For example, a feature in the FlexNet Publisher Licensing Toolkit could represent any of the following: ● The entire software application. ● A subset of menu items. ● A specific set of data available from the software application. ● A process in the software application. The software publisher also defines the license model for each FlexNet Publisher Licensing Toolkit feature. This gives software publishers the ability to define a different license model for each feature in the software application.
feature bundle	A collection of features in FlexNet Operations. It exists primarily to enable modularity and re-use.
feature definition line	An entry in a license file or fulfillment record that describes a feature. Each line begins with the keyword FEATURE, INCREMENT, or UPGRADE. In license files, feature definition lines are used with SERVER and VENDOR lines to define the license model. In trusted storage, feature definition lines are used with the fulfillment record properties to define the license model.
feature line	See feature definition line.
file-based activation	See manual activation.
FlexEnabled application	A software application that uses the FlexNet Publisher Licensing Toolkit to implement its license models.
FlexEnabled client	A client system where a FlexEnabled application is installed.
Flexible API	Functions available in the FlexNet Publisher Licensing Toolkit libraries that allow FlexEnabled clients to access license rights.
FlexNet Agent	A utility that runs on a license server and allows communication between the license server and other FlexNet products.
FlexNet Manager	This is a license server management and license management decision support solution in the FlexNet product family that allows companies to centrally manage license servers and license usage across the enterprise.
FlexNet Operations	A product in the FlexNet family of products that manages entitlements and generates license rights.

Term	Meaning
floating license	A license that can be shared among users and is served to FlexEnabled applications from a license server. When a user finishes using the license, another user can begin using it. Also called concurrent license or served license rights.
fulfill	The action of getting license rights.
fulfillment record	In the FlexNet Publisher Licensing Toolkit, a data structure specific to trusted storage that defines license rights.
heartbeat	A type of message sent periodically between two systems (for example, a FlexEnabled application and a license server) to identify whether each is running and able to communicate with the other.
hop count	In the FlexNet Publisher Licensing Toolkit, the number of times that license rights in a single fulfillment record have been transferred from one license server to another.
hostid	In the FlexNet Publisher Licensing Toolkit, the value of a specific system attribute that uniquely identifies the host. For example, if IP address is a hostid type, then the hostid might be 129.87.33.101. Hostids are used in node-locked license.
hostid type	A type of attribute that is available to uniquely identify a system. For example, disk serial number, IP address, or MAC address.
hybrid license	In the FlexNet Publisher Licensing Toolkit, a type of license right in the license group on trusted storage that can function as either a concurrent license or as an activatable license.
internet activation	See programmatic activation.
keyboard activation	See short-code activation.
license administrator	A person in an enterprise who is responsible for installing the FlexEnabled application and administering the licenses and license servers.
license file	A text file, usually with the `.lic` extension, that contains license certificates from one or more publishers.
license-file list	See license search path.
license generator	A general term used to describe any mechanism a software publisher uses to generate license rights for their customers. The following FlexNet capabilities and products can function as a license generator: ● FlexNet Operations: generates license rights for trusted storage and license files. ● lmcrypt: generates license rights for license files.

Term	Meaning
license group	A mechanism used to describe licenses rights held in server-side trusted storage. These groups are not used with license rights held in license files. ● **activatable**—A license right in trusted storage that can be activated to a FlexEnabled client. ● **concurrent**—license rights in this group can be shared among users by having a FlexEnabled client to check out, and then return the license. ● **hybrid**—these license rights have the same characteristics as both the activatable license group and concurrent license group. They can be activated to client-side trusted storage on a FlexEnabled client or checked out by a FlexEnabled client. Although you see these group names used to describe license rights in client-side trusted storage, the capabilities of the license rights do not change. Once license rights are in client-side trusted storage, you cannot transfer, serve, or activate them to another system. The FlexEnabled application can only check out and check in these licenses. The group names are used when a license right is returned to the license server, so that it is assigned to the correct license group on the license server.
license model	The set of characteristics that determine the conditions under which (for example, how, when, where, and by whom) a FlexEnabled application can be used.
license rights	Information in trusted storage or license files that defines the license model and controls user access to a specific FlexEnabled application.
license search path	A property that the FlexEnabled application uses to locate where license rights are stored. Sometimes known as a license-file list.
license server	A set of files from the FlexNet Publisher Licensing Toolkit, that work together to manage and serve licenses to FlexEnabled applications. A license server is required to support a concurrent license model. The license server consists of a license server manager, one or more vendor daemons, zero or more options files, zero or more debug logs, and, optionally, a report log. These components are available in the FlexNet Publisher Licensing Toolkit. If the license server supports license rights held in trusted storage, it also includes an activation utility. It may, optionally, act as an activation server.
license server system	See license server.
license server manager (lmadmin or lmgrd)	A license server manager `lmadmin` or `lmgrd` executable that forwards connections from a FlexEnabled application to the correct vendor daemon. Because the vendor daemons are responsible for authenticating requests and serving licenses, a single license server manager can be used with multiple vendor daemons on the license server.

Term	Meaning
license sharing	A capability available in the FlexNet Publisher Licensing Toolkit where multiple license requests from the same user, host, or display results in only one license being used. Also known as duplicate grouping.
licensing client	See FlexEnabled client.
lmadmin	The license server manager introduced in version 11.6 which includes a Web-based GUI.
lmgrd	A command-line-based license server manager.
local license rights	License rights that are stored on the same system where the FlexEnabled application is running, rather than on a license server.
local activation	A type of activation where license rights are loaded using an ASR file. The ASR file must reside with the FlexEnabled application. This does not require user intervention.
machine ID	A value, generated using attributes of the system, that is used to bind trusted storage to that system.
maintenance	A maintenance represents a customer's right to obtain updates and upgrades to a particular licensed product or suite for a specified duration. Used in FlexNet Operations.
manual activation	An type of activation where license rights are transmitted from an activation server to another system using a set of request and response messages that are formatted as XML. These messages are saved to and read from physical XML files. An activation utility loads the license rights from an XML file into trusted storage.
Manual Response Generator (responsegen.exe)	A tool included in the FlexNet Publisher Licensing Toolkit that enables software publishers to test the activation process offline using XML request and response messages stored as files.
node-locked license	A license that can be used only when the FlexEnabled application is run on a specific system or by a specific user (as defined by the hostid).
options file	A configuration file available on the license server that license administrators can use to either change license server behavior or implement certain licensing policies. Each software publisher has its own separate options file that works with a specific vendor daemon.
orderable	Anything that can be purchased by a customer. Orderables can include suites, products, documentation, and maintenances.
package	In the FlexNet Publisher Licensing Toolkit, a set of features that can be grouped and licensed together.
part number	A unique identifier, typically correlated with an order system, assigned by a publisher to a product.

Term	Meaning
permanent license	License rights granted to a customer that do not expire.
product	Anything that can be sold by a publisher and can be licensed.
product ID	In the FlexNet Publisher Licensing Toolkit, one of the values that uniquely identifies the license rights in trusted storage. See also entitlement ID.
programmatic activation	A type of activation where license rights are transmitted from one system to another across a network using a set of request and response messages that are formatted as XML. This process is the identical to manual activation, except that it does not generate a physical XML file. An activation utility loads the license rights from the XML message into trusted storage. This is an automated process and does not require user intervention.
publisher ID	A value used by trusted storage to uniquely identify the software publisher's logical and physical licensing components. Flexera distributes this identifier to software publishers. It is used to prepare certain FlexNet Publisher Licensing Toolkit files that access trusted storage. This value is not the same as the vendor daemon.
quorum	A condition specific to three-server redundancy that is met when at least two of the three license servers are running and can communicate with each other using heartbeats.
rehost	The process of reissuing a license for a different system. This process requires software publishers to perform certain activities to redefine the license rights so that they will work properly on the new system.
repair request	A message sent by an activation utility to an activation server that initiates a repair transaction. The activation server provides the information that repairs the license rights in trusted storage. An activation utility is an application that initiates the repair transaction.
repair response	A message sent from an activation server to an activation client, in response to a repair request. This message contains information that is used to the repair the license rights.
repair transaction	A sequence of repair request and repair response messages, and the related actions that result in license rights being repaired in trusted storage.
report log file	A file that runs on a license server that contains data about the features used by a single vendor daemon. Report logs are encrypted and cannot be read by a person, but are used by FlexNet Manager to produce reports.
responsegen	See Manual Response Generator (responsegen.exe).
return request	A message sent by an activation utility to an activation server that initiates a return transaction. The activation server receives the license rights returned by an activation client. An activation utility is an application that initiates the return transaction.
return response	A message sent from an activation server to an activation client, in response to a return request.

Term	Meaning
return transaction	A sequence of return request and return response messages, and the related actions, that result in license rights being returned from trusted storage to the activation server.
served license rights	License rights that are stored on a license server and are checked out when needed by FlexEnabled applications that run on client systems. These license rights are used for concurrent licensing.
short code	A multi-character code used with a short-code activation transaction, a repair transaction, or a return transaction.
short-code activation	A type of activation where license rights are loaded from an ASR file using an activation transaction that requires a specific code (called a short code). Often the user enters this code into a prompt from the application. This is also is used for licensing via automated telephony.
signature	A secure multi-character string added to the license certificate that ensures it has not been modified.
simple entitlement	An entitlement for a single, specific customer.
single-action transaction	This is the default transaction type for FlexNet Publisher versions 11.7 and earlier. Each transaction can include only a single action. Also known as legacy or V1 transactions. From FlexNet Publisher version 11.14, single actions are deprecated.
suite	A set products grouped together that are functionally complementary and associated with one or more license models.
supersede	A capability in the FlexNet Publisher Licensing Toolkit, used to replace the characteristics of old license rights with new characteristics.
three-server redundancy	A capability in the FlexNet Publisher Licensing Toolkit that enables license administrators to configure three license servers in a failover cluster.
triad	A set of three license servers which operate together to support three-server redundancy.
trial license	License rights granted to a user for a limited duration so that the user can evaluate the software application.
trusted storage	A tamper-proof area on the system that stores license rights. This storage location is different than license files and requires a different mechanism for loading and managing license rights. Compare with license file.
trusted configuration	A set of attributes that define the security configuration of a segment of trusted storage. Each software publisher can configure their segments of trusted storage differently. This configuration defines the settings for binding, anchoring, and windback detection.
trusted configuration file	A file that stores the attributes that define a software publisher's trusted storage segment configuration.

Term	Meaning
trusted storage configuration	See trusted configuration.
uncounted	A license right attribute that allows an unlimited number of concurrent uses of a FlexEnabled application. Compare with counted.
unserved license rights	These are license rights that are not served from a license server. These license rights are located with the FlexEnabled application. Also known as local license rights.
upgrade	A process where a user installs a newer version of a FlexEnabled application and also updates the license rights so that they can use this newer version.
upsell	The process of transitioning from a product of lesser functionality to one of greater functionality in the same product line.
vendor daemon	One of the files that is a part of the license server. This executable is customized and built by the software publisher using files in the FlexNet Publisher Licensing Toolkit. The vendor daemon is responsible for communicating with the FlexEnabled application and issuing licenses.
Vendor Certificate Generator	A publisher-built component, that is a part of FlexNet Operations, that creates license certificates or signed feature lines in fulfillment records. This functions similarly to lmcrypt.
vendor name	A value used to uniquely identify a software publisher's logical and physical licensing components. It is used to prepare certain files in the FlexNet Publisher Licensing Toolkit that manage license rights in both trusted storage and license files. Flexera distributes this identifier to software publishers. This value is not the same as the publisher ID.
windback	A condition where the system clock has been set to an earlier date or time. One reason users change the system clock is to extend the license duration.
windback detection	A capability in the FlexNet Publisher Licensing Toolkit that detects whether a user has reset the system clock. This prevents a user from extending the duration of the license.
windback tolerance	An acceptable amount of windback to a system clock that won't trigger windback detection. A certain tolerance is normally allowed to account for time zone differences or Daylight Saving Time.

Index

A

activatable license 33, 34, 278
activated feature line, See feature definition line
activation 278
    local 83
    process summary 29
    short code 287
    using license server 83
Activation API 92, 113, 278
activation application 279
Activation Borrow API 86
activation client 279
activation library 20
activation request 191, 279
activation response 279
    schema 185
activation server 249, 279
activation specification record 20, 97, 118, 159, 164, 174, 175, 220, 223, 279
    creating 178
    editing 182
activation transaction 279
    between FlexEnabled client and activation server 82
    between license servers 30
    general, introduction 14
    short codes 84, 158
    with manual response generator 106
    with Operations server 110
alphanumeric short codes 235
anchor 279
anchoring 41, 231, 264
application activation utility 18
    for single-action transactions 92
asrgen 178
    functions 182
    license template 180
    Trusted Configuration 181
    usage 179

B

binding 279
    FLEXIBLE 265
    trusted-storage security feature 40
    XML schema 264
BORROW
    replacing with activation borrow API 86
    return of borrowed license 86
borrowing 280
bulk entitlement 280

C

checkin 280
checkin callback, vendor-defined 67
checkin filter, vendor-defined 67
checkout 280
checkout filter, vendor-defined 67
chroot and ls_do_checkroot 74
client system 280
CodeBase 235, 240
composite action
    response schema 185
composite request 190
composite transaction 280
concurrent license 33, 34, 280
config response 280
config response (client)
    schema 185

config response (server)
schema 185
consumer 280
continuation 211
counted license 280
custom data
response codes 241

D

daemon, vendor 288
debug log file 24, 280
Deduction Records 130
demo license, See trial license
duplicate grouping 281

E

embedded activation, See local activation
encryption seed 281
end user 281
end-user license administrator, See license administrator
end-user machine, See client system
enterprise 281
entitlement 281, 284
entitlement ID 281
expiring license 281

F

failure response
schema 185
feature 282
feature bundle 282
feature definition line 85, 164, 181, 209, 211, 256, 278, 282
comment lines 211
concurrent licenses 210
DUP_GROUP 215
HOST_BASED 215
ISSUED 215
ISSUER 216
License Pools 212
LM_A_CHECKOUTFILTER 219
MINIMUM 216
NOTICE 216
PLATFORMS 217
Signing 164
SN 217
START 217
SUITE_DUP_GROUP 217
syntax 210
TS_OK 218
USER_BASED 218
Vendor Keywords 214
VENDOR_STRING 219
FEATURE/INCREMENT line, See feature definition line
file
license 283
options 285
report log 286
file-based activation, See manual activation
FlexEnabled application 282
FlexEnabled client 282
Flexible API 50, 93, 112, 114, 137, 161, 282
FLEXIBLE binding 265
FlexNet Agent 282
FlexNet Licensing Service 95, 116, 144
installing (Linux) 58
installing (OS X) 60
installing (Windows) 58
FlexNet Operations 282
FlexNet Operations Server 110
floating license 283
fulfill 283
fulfillment record 283
asrgen 182
creating 246
deleting 102
examples 254
in activation response 36
on FlexEnabled client 85
overview 22
returning to license server 103, 122
returning to manual response generator 107
structure 247
viewing 98
Fulfillment type 248, 256

H

heartbeats 283
hop count 283
Hostid 181
hostid 283
hostid type 283
hybrid license 33, 34, 283

I

installanchorservice.exe 60
installing
FlexNet Licensing Service (Linux) 58
FlexNet Licensing Service (OS X) 60
FlexNet Licensing Service (Windows) 58
internet activation, See programmatic activation

K

key files 186
keyboard activation, See short-code activation

L

lc_vsend()
  and ls_vendor_msg 73
license
  counted 280
  served 287
  unserved 288
license administrator 283
license expiry, in short code transactions 242
license file 283
license files, generating 164
license generator 283
license group 284
license groups 34
license model 284
license pools
  by entitlement ID 213
  forming 212
  managing counts 35
license rights 284
  local 285
license search path 284
license server 27, 114, 284
  debug log 24
  report log 24
license server manager 284
  introduction 23
license server manager, definition 284
license server system 284
license sharing 285
license template 180
license types 33
  activatable 33, 34
  concurrent 33, 34
  hybrid 33, 34
license-file list, See license search path
licenses
  limiting mobility 31, 253, 254, 258
  pooling 213
  transferring between servers 30
licensing client 285
lmadmin 23, 285
lmcrypt 164
lmflex.c 51
lmgrd 23, 285
lmutil 24
local activation 97, 223, 285
local license rights 285
log file
  debug 280
  report 286
ls_a_check_baddate 65
ls_a_license_case_sensitive 74
LS_ATTR_CHECKOUT_DATA 69
ls_attr.h 68, 69
ls_checkout() and ls_outfilter 69, 71
ls_compare_vendor_on_increment 65
ls_compare_vendor_on_upgrade 65
ls_daemon_periodic 66
ls_do_checkroot 74
ls_dump_send_data 74
ls_entitlement_based_pooling 74
ls_get_attr(), and ls_outfilter 68, 69
ls_get_status_is_master 66
ls_incallback 67
ls_infilter 67
ls_min_lmremove 67
ls_minimum_user_timeout 67
ls_outfilter 67
ls_show_vendor_def 70
ls_use_all_feature_lines 76
ls_user_init1 71
ls_user_init2 71
ls_user_init3 71
ls_user_lockfile 76
ls_user_lockfile2 77
ls_vendor_msg 73
lsvendor.c 63

M

machineid 285
manual activation 285
Manual Response Generator 285
manual response generator
  using command-line interface 199
  using on Windows 188
manual response generator, See also responsegen 184
MinimumCodeSize 235
mobile licensing 86
model, license 284

N

newlink GlossaryC 280
newlink Name 237

O

options file 25, 285
overdraft 211

overdraft condition 211

P

package 285
part number 285
permanent license 235, 243, 286
pooling, See license pools 213
preptool 38, 116, 144, 166
    configuration files 169
    usage 167
product 286
programmatic activation 286
publisher id 286
PUBLISHER_DEFINED_DURATION 236
PUBLISHER_DEFINED_EXPIRY_CDAY 235, 242

Q

quorum 286

R

regular entitlement, See simple entitlement
rehost 286
Remote Desktop support 218
repair request 194, 286
    analyzing 195
repair response 286
    schema 185
repair transaction
    from license server 104
    from Operations server 112
    general, introduction 14
    initiated by application activation utility, description 83
    initiated by server activation utility, description 32
    using manual response generator 108
repairing, trusted storage 108
report log file 24, 286
response codes
    custom data
        241
response files 185
responsegen 286
    API usage example 207
    automated use 203
    command-line syntax 199
    errors 201
    operator 202
responsegenUI.exe 188
return request 193, 286
return response 286
    key file 186
    schema 185
return transaction 14, 287
    description 32, 84
    initiated by application activation utility, introduction 79
    initiated by server activation utility, introduction 27
    to a FlexNet Operations server 119
    to license server 103, 122
    to manual response generator 107, 110, 125
    to Operations server 110
returning
    fulfillment record to license server 103

S

SCI_ERROR_ID 231
SCI_NEW_MACHINE_ID 231
SCI_ORIGINAL_MACHINE_ID 231
SCI_REPAIR_SCOPE 231
SCI_REQUEST_SEQUENCE_NO 231
SCI_TRUST_FLAGS 231
SCI_TSSN 232
SCI_UMN1_IS_VALID 232
SCI_UMN2 232
SCI_UMN2_IS_VALID 232
SDTs
    In multiple executables 171
served license rights 287
server
    license 284
server activation utility 19
    for composite transactions 135
    for single-action transactions 113
short code 287
short code transactions 84
short code transactions, license expiry 242
short-code activation 223, 287
short-code request 196
short-code response
    schema 185
signature 287
signing XML documents 175
simple entitlement 287
single-action transaction 287
SOAP module 22
suite 287
supersede 287

T

tcencrypt 174
    usage 174
tcencyrpt 186
Terminal Server support 218
three-server redundancy 287

TIMEOUT
setting 67
transaction
repair, See repair transaction 32, 83
return, See return transaction 27, 79
transactions
with license server 102
transfer transaction
introduction 14
transferring licenses 30
triad 287
trial license 287
Trial Pack ID XML Snippet 49
Trial Packs 47
TRIAL_PACK_IDS Entry 49
TRIAL_PACK_SIGNATURE Entry 49
Trust Flags 231
trust flags 196
trusted configuration file 287
Trusted ID Configuration 22, 106, 108, 109, 125, 159, 164, 174, 175, 178, 181, 182, 186, 193, 221, 226, 260, 261, 269, 272, 273
trusted sections 193
trusted storage 287
cleaning up 176
components 18
definition 21
repair 32, 83
security 38
trusted storage configuration 288
Trusted STorage Serial Number 196
tsreset 176
usage 177
TSSN 196
type, hostid 283

U

uncounted license
uncounted 288
uninstallanchorservice.exe 61
Unique Machine Number 162
ASR overrides 163
unserved license rights 288
unserved licenses 288
upgrade 288
upsell 288
utilities
asrgen 178
lmcrypt 164
preptool 166
tcencrypt 174
tsreset 176
XMLSign 175

V

Vendor Certificate Generator 288
vendor daemon 288
introduction 24
vendor name 288
VENDOR_STRING
and ls_compare_vendor_on_* 66
vendor-defined checkin callback 67
vendor-defined checkin filter 67
vendor-defined checkout filter 67

W

windback 288
windback detection 288
activating 39
definition 38
windback tolerance 288
configuring 39
definition 39

X

XML documents, signing 175
XMLSign 175