



**InJoin™ Meta-Directory™
Connecting Proprietary Data
Servers using the JoinEngine
Perl Plug-In Guide**

Critical Path
USA 185 Berry Street (Suite 4700)
San Francisco, CA 94107, USA
Telephone : +(415) -659 3524
Fax: +(415) -659 0006
technical.support@cp.net

Critical Path
Ireland 42 - 47, Lower Mount Street,
Dublin 2, Ireland.
Telephone: +353 (1) 241 5001
Fax: +353 (1) 241 5170
technical.support@cp.net

Critical Path
Website <http://www.cp.net/>

InJoin™ Meta-Directory™ Connecting Proprietary Data Servers using the JoinEngine Perl Plug-in
Guide
January 2001

Copyright © 1995-2001 Critical Path, Inc. All rights reserved. No part of this publication may be reproduced, transmitted, transcribed, stored in a retrieval system, or translated into any language, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from Critical Path. Printed in Ireland. The information furnished herein is believed to be accurate and reliable. However, no responsibility is assumed by Critical Path for its use, nor for any infringements of patents or other rights of third parties resulting from its use. Critical Path and the Critical Path logo are trademarks of Critical Path, Inc.

Includes technology and information from the University of Michigan at Ann Arbor. Copyright © 1992-1996 Regents of the University of Michigan. All rights reserved.

Acrobat® Reader Copyright © 1987-2001 Adobe Systems Incorporated. All Rights reserved. Adobe and Acrobat are trademarks of Adobe Systems Incorporated, which may be registered in certain jurisdictions.

This product includes software that is copyright (c) 2001 by Sleepycat Software, Inc. All rights reserved.

Portions copyright 1991-2001 Compuware Corporation.

Microsoft and Windows NT are registered trademarks and Internet Explorer is a trademark of Microsoft Corporation.

ActiveState, ActivePerl, and PerlScript are trademarks of ActiveState Tool Corp. Commercial Support for ActivePerl is available through the PerlClinic at <http://www.PerlClinic.com>. Peer support resources for ActivePerl issues can be found at the ActiveState Web site under support at <http://www.activestate.com/support/>.

All other trademarks and registered trademarks are the property of their respective holders.

1	Overview	6
	Working with proprietary Data Servers	6
	Architectural Overview of the JoinEngine Perl Plug-In	7
	Operational Overview of JoinEngine	9
	Data Entries	9
	State Databases	11
	JoinEngine Multi-threading	13
2	Interfacing with Proprietary Data Servers	14
	Interfacing with proprietary Data Servers	14
	Project and Deployment requirements	14
	Data Propagation	15
	Interface APIs (C/C++, file)	17
	LDAP Schema Considerations	17
	Data Translation and Information Inference	17
	Proprietary Data Server Entry Indexing	18
3	Data Record Definition	20
	Data Record Definition	20
	Important Notes	22
	Sample Data Record	23
	Multi-Valued Attributes	23
	Overriding the JoinEngine default object-class	24
	Specifying binary attribute values	25
4	Perl Script Function Interface	28
	Overview of the Perl script function interface	28
	Package Constructor and Destructor	30
	Script Initialization Routines	30
	CV to MV synchronization routines	32
	MV to CV synchronization routines	35
5	Some Design Considerations	38
	Logging	38
	Scheduling	38
	Renaming entry index values	38
	Renaming entry LDAP DN values	39

6	Miscellaneous Issues	40
	Further Information on Perl	40
	ActiveState Tool Corporation's ActivePerl and InJoin Meta-Directory	41
	LDAP Schema Extensions	42
	Checklist	42
	Operations for the Proprietary Data Servers	44
	Interacting with the Perl script	44
	SAP and PeopleSoft Integration	45
	SAP	45
	PeopleSoft	49

Chapter 1

Overview

This chapter provides information on using the InJoin Meta-Directory to synchronize data between a proprietary Data Server and an LDAP compliant directory. A good understanding of the contents of this document should greatly assist those intending to write JoinEngine Plug-In Perl scripts.

To understand this manual, it is assumed that you have:

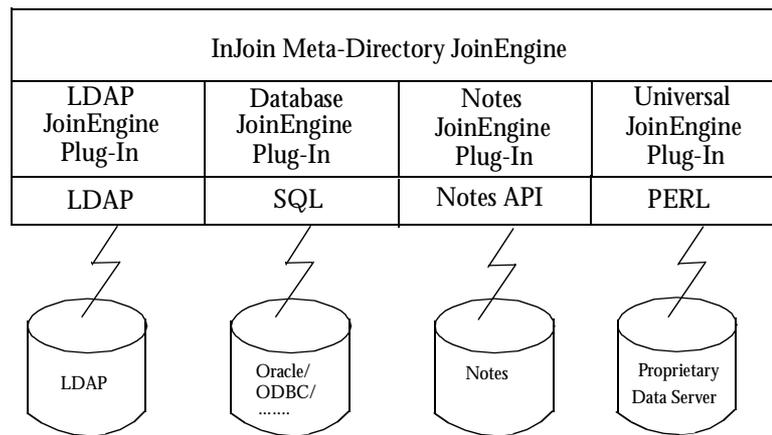
- Experience writing Perl scripts
- Experience using InJoin Meta-Directory
- Received training on InJoin Meta-Directory Release 3.x
- Full knowledge and understanding of the architecture of the deployed meta-directory

Working with proprietary Data Servers

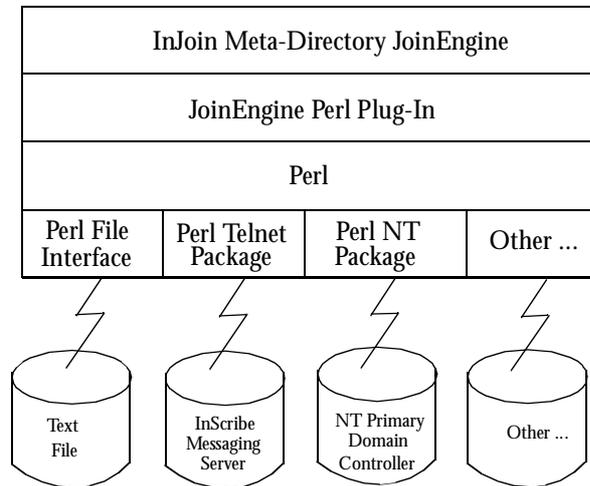
There is a wide variety of sources of information that can be included in a meta-directory. The possible data servers may be:

- Text file (many proprietary data servers offer text file import/export utilities)
- Database
- Remote server (accessible over ftp, telnet, and so on)
- Mail system directory (for example, cc:Mail)
- Operating system user registry (for example, NT domain controller or UNIX /etc/password file)

Architectural Overview of the JoinEngine Perl Plug-In

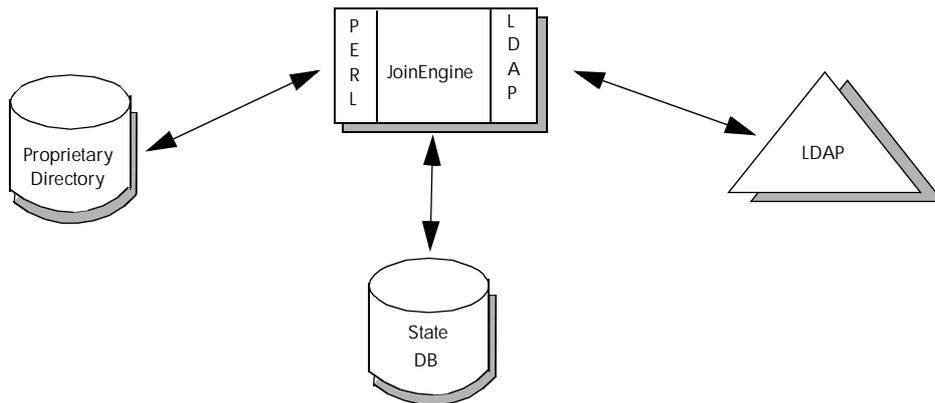


The JoinEngine server interfaces with a variety of different Data Server types using both standard and proprietary API and protocols. The LDAP protocol is used to communicate with directories such as InJoin Directory Server, Active Directory, and iPlanet/Netscape Directory Server. A combination of Open Database Connectivity (ODBC) and native DB access is used to communicate with databases such as Oracle. The Notes API is used to communicate with Notes Servers. In addition, a Perl API may be used to connect other Data Servers to the JoinEngine. The JoinEngine interfaces to a Perl script through a defined API. The Perl script interfaces with a Data Server using suitable interface techniques.



Perl is quite extensive and Perl modules exist for enabling communications with many Data Server types. Many of these modules are supplied with the binary distribution of Perl. Other modules may be downloaded from various Web sites. the diagram above illustrates some sample uses.

Operational Overview of JoinEngine



The diagram above provides a high-level overview of how the JoinEngine interfaces proprietary directories with LDAP compatible directories. A State Database is monitored by the JoinEngine to maintain information regarding the managed data.

Data Entries

Data entries generally have a number of attribute-value pairs. For example, some sample attributes from an NT user record are as follows:

```
objectclass=[L]top, person, organizationalPerson, ntuser
ntuserdomainid=JohnS
ntuserhomeDir=x:\users\johns
ntusercomment=localization engineer
description=Meta-Directory localization engineer
ntuserbadpwdcount=2
ntusernumlogons=45
ntuserlogonserver=PHOENIX
ntusercountrycode=121
ntuseruniqueid=johns
```

```
ntuserprimarygroupid=localeng
```

An important requirement on all data entries is that they can be uniquely and consistently identified, either by a single attribute or by a combination of a number of the entry's attributes. The entry's distinguished name (or GUID in the case of Microsoft's Active Directory) uniquely identifies each data entry in an LDAP directory. Unique names are important because the JoinEngine needs to correlate entries between synchronization cycles. Consistent index naming is required because the JoinEngine will expect the script to consult entries in proprietary data servers given the entry index value.

Data Entry Ownership

The JoinEngine is responsible for replicating and linking data entries between proprietary data servers and LDAP directories. For example, if an administrator adds a new data entry to a proprietary Data Server, the JoinEngine will replicate the entry to the LDAP directory creating a new equivalent LDAP directory data entry. For deleting entries, there are a number of possible scenarios?

- If the LDAP directory data entry is deleted, should the delete be propagated down to the proprietary directory, and the entry removed from the proprietary Data Server? Should the delete be flagged as an invalid operation and should the entry be re-added to the LDAP directory?
- What should happen if the original proprietary Data Server entry is deleted? Should the delete operation be honored or should an attempt be made to re-add the entry to the proprietary Data Server.

The answer is that both options are perfectly valid and the actual behavior depends on the administrator requirements. The behavior is managed on a per-entry basis by the JoinEngine as it maintains an entry ownership flag, and the ownership is assigned to either the LDAP directory or the proprietary Data Server. Ownership primarily bestows delete privileges. Ownership may also be used to control Attribute Flows. Only the owner may delete an entry. If the owner entry is deleted, the delete will be propagated to the adjacent system, either the LDAP directory or the connected directory. If a non-owner entry is deleted, the JoinEngine will attempt to re-add the deleted entry.

State Databases

The JoinEngine Perl Accessible Server State database serves a number of specific purposes. Every replicated data entry, whether it originated in the proprietary Data Server or the LDAP directory, is maintained in the State database. Each data entry has a single record in the State Database. Each State database entry in turn has a number of fields. The JoinEngine may be configured to maintain a State database for each connected proprietary Data Server. The following sections describe the structure and purpose of the State database.

State Record Contents

Either the [Proprietary Index] field or the [LDAP DN] field indexes each record. The following fields comprise each State database record:

[Proprietary Index]

The proprietary index uniquely identifies a data entry in the proprietary Data Server. Each proprietary data entry must be uniquely and consistently identifiable by an index value – the index is typically an attribute value (for example, UserID) or a Database record count.

[LDAP DN]

The LDAP DN is the distinguished name of the equivalent entry in the LDAP directory. It should be noted that in the case of Microsoft Active Directory, the entry GUID rather than the DN is used as the LDAP directory indexing attribute.

[Contents Hash Values]

The contents hash values maintain a hash of the contents of the attributes that have been flowed in either direction between the proprietary Data Server entry and the LDAP directory entry. If no data attributes flow between from the proprietary Data Server, this hash value will be zero.

Data Entry Hash values

The JoinEngine maintains two hash values for each data entry it manages. The proprietary contents hash value is a hash of the data entry's attributes that have flowed from the proprietary Data Server to the LDAP directory. The directory contents hash value is a hash of the attributes that have flowed from the LDAP directory to the proprietary Data Server.

The purpose of the hash values is to allow the JoinEngine to quickly determine unchanged data entries. The hash values can also be used to determine data entries that have changed but where the data changes are not applicable. For example, if a new proprietary data entry is detected, it is replicated to the LDAP directory and a State database entry complete with the proprietary entry contents hash is added to the State database. Upon subsequent synchronization cycles, after reading the proprietary entry, the proprietary hash is recalculated. If it is the same as the hash value in the State database, no applicable changes have occurred and no further processing of the record is required. This technique provides optimal performance of change detection and updates as well as minimizing the load on the entire meta-directory system.

Purposes of the State database

The main purposes of the State database are:

- Maintaining hash values of both proprietary and LDAP data entry contents.
- Detection of new entries – if an entry is not present in the State database, it is assumed to be a new entry.
- Detection of deleted entries – The JoinEngine is responsible not only for adding and modifying data entries to Data Servers, but also for detecting entry deletions and propagating delete operations to the adjacent system. Many proprietary data servers do not provide incremental updates of changes to their data entries and instead provide the entire set of the data entries on each synchronization cycle. Some Data Servers do not provide incremental entry delete events. The JoinEngine detects deletes on such systems by comparing all entries in the Data Server against the State database. Any entries in the State database not in the Data Server are assumed to have been deleted.

Note. LDAP directories provide incremental data changes, including delete operations, and therefore this deductive method of determining deletes is not used for LDAP directories.

- Detection of modified entries - by maintaining hashes of the proprietary and directory contents, the JoinEngine can determine inapplicable entry changes and discard these changes without further processing.

JoinEngine Multi-threading

Each JoinEngine task consists of a number of independent threads and can read and write data concurrently. There may be some interleaving of the perl synchronization routine sets (see later). Typically, this does not present an issue with proprietary data servers, however, script writers need to be aware of this fact and make appropriate provisions in their scripts.

Plug-in scripts are always implemented as independent Perl packages and therefore operate within independent environment spaces. If two or more scripts are sharing external resources (for example, log files), proper provisions must be made within the scripts to support such resource sharing.

Chapter 2

Interfacing with Proprietary Data Servers

This chapter provides a discussion of the various issues associated with interfacing InJoin Meta-Directory with proprietary Data Servers.

Interfacing with proprietary Data Servers

When interfacing with proprietary Data Servers, careful consideration should be given to how best to interface given the various architectural and project constraints. Most directory systems use some form of file export mechanism and the obvious temptation is to use these files. The advantages of interfacing using the text file import/export mechanism is that it is workable, generally straightforward, and easily adaptable. The disadvantages include that it is often extremely inefficient and cumbersome. Also, functionality that may be available through direct API interfaces may not necessarily be available through file feeds. There is an abundance of Perl modules and packages already developed and freely available on the Internet that can offer more efficient interface options. Time spent examining various interfacing options will invariably result in significant long term gains. The following issues should be carefully considered when deciding on the interface mechanism.

Project and Deployment requirements

Various factors regarding the likely deployments are:

- Under what time constraints is the project operating?

- What relative importance is attached to the Data Server in relation to the entire project?
- Will the completed Data Server script be used as a once off deployment or will it be reused by both internal and external organizations in potential mass-deployment scenarios? Typically, for once-off deployments, custom scripts tailor-made for the scenario at hand is the most sensible approach. For mass-deployment scenarios, scripts should be more generic and more configurable.
- What is the expected lifetime of the script? JoinEngine scripts are often developed to assist organizations migrating from one directory system to another and the script's usefulness will cease once the migration is complete.
- Will the original script author be responsible for maintaining the script? If not, as is generally the case, adequate documentation and appropriate comments should be included with the script.

Data Propagation

Various issues regarding how and when data is to be propagated need consideration.

- Is unidirectional or bi-directional data flow required (that is, will data entries flow from the proprietary Data Server to the LDAP directory, flow from the LDAP directory to the proprietary Data Server, or flow in both directions)?
- What is the expected volume of data entries?
- What is the expected rate of data change?
- What is the expected data granularity level?
 - Entry level granularity implies that entire entries (and all contained attributes) are atomic and different attributes may not flow in opposite directions.
 - Attribute-level granularity supports mastering of separate entry attributes from either the proprietary or the LDAP directory.

For example, a proprietary e-mail address book may maintain various attributes about all registered users including the users e-mail address, e-mail privileges and various personal details such as job title, office and telephone number. It may be

appropriate for the e-mail system administrator to manage the e-mail specific attributes within the e-mail address book and for the HR manager to manage the users' personal details from a HR database and flow the personal attributes to the e-mail address book.

- What is the expected frequency of updates and what replication time-frame is required. For example is it sufficient to propagate all changes that occur during a 24-hour period at 12.00am, or must changes be replicated within a much smaller time window? Many user directories do not experience many changes on the typical daily basis, whereas directories maintaining information on inventory, for example, may change on an hourly basis.
- Data confidentiality, data security, and data integrity also need careful consideration.

Full/Incremental Data Loads

For data entry change detection within the proprietary Data Server, the ideal solution is for the proprietary Data Server to report incremental changes. For example, in the case of an e-mail directory containing 10,000 user entries, if the administrator modifies 50 records on average per day and if the e-mail directory can report which 50 entries have been modified (or added or deleted), the workload on the JoinEngine is limited to processing these 50 entries. The alternative is that a full synchronization is performed and all 10,000 entries are processed and checked for changes. Although the JoinEngine optimizes the delta detection to a considerable degree by the use of the State database, it still must process the entire 10,000 entries to some extent.

Many proprietary Data Servers now include mechanisms for change detection. If such mechanisms exist, it is advisable to avail of them.

The performance of the script reading and writing data to the proprietary Data Server and hosted system should be considered. Some directories may slow down considerably after an administrator directory connection is opened.

Interface APIs (C/C++, file)

Many proprietary Data Servers may be accessed through vendor-supplied APIs. Many directories also offer text file import and export options. The Perl language may be extended by building Perl packages upon vendor-supplied C and C++ APIs using the standard Perl XS and SWIG processes. For further information, please refer to the Perl man pages (`perlxs`, `perlxtut`, `perlcall`) or relevant sections within Perl reference books (*Advanced Perl Programming*, Chapter 18, is a useful starting point).

LDAP Schema Considerations

The LDAP directory's schema may not include object classes and attributes required to host replicated data entries from the proprietary Data Server. If this is the case, it may be necessary to define new object classes and new attributes within the LDAP directory's schema. LDAP schema modifications are generally a tedious exercise prone to repeat iterations regardless of which LDAP directory in use. It is therefore advisable to attempt to use common object classes and attributes if possible and define appropriate mappings between the proprietary attributes and the LDAP attributes. If this is not feasible, it is advisable to formally define schema extensions even if the Data Server is only for use within an organization.

It is also typically not advisable to overload the use of standard attributes because this can potentially create interoperability problems with other software at a later date. For example, it is not advisable to insert an `employee_security_id` into an `inetorgperson` `description` attribute.

Data Translation and Information Inference

Depending on the layout of proprietary data records it may be necessary to parse attribute values and infer additional information from the source entry. For example, a proprietary Data Server may host user names within a single common name attribute. When replicating the user entry into an LDAP `inetorgperson` entry, as `surname` is a mandatory attribute, it may be necessary to parse the common name attribute into the name components (typically given name, initials, surname, gener-

ation). Also some information can be inferred from other information. For example, a country attribute can generally be inferred from a city (for example, if city is New York, country can be inferred to be USA).

Proprietary Data Server Entry Indexing

As already discussed in the architectural description, a fundamental requirement of the JoinEngine is that all entries can be uniquely identified by either a single attribute or a combination of a number of attributes.

Chapter 3

Data Record Definition

This chapter provides a formal definition of the Perl data record format. The related topics discussed include multi-valued attributes, overriding the default object-class, specifying binary attribute values, and deleting attributes.

Data Record Definition

The following is the formal language definition of a Perl data record. The definition may seem somewhat terse but most of the terseness relates specifically to the formal definition of rfc-822 mailboxes and is therefore not of general concern. A number of examples are supplied at the end of this section and demonstrate various aspects of how records should be used.

```
#
# <record> ::= <index-field>
#           1*<field>
# <index-field> ::= "index" "=" 1*<word>
# <field> ::= /
#           "rfc822"           "=" <mailbox>
#           "alias"           "=" <mailbox>
#           "commonname"      "=" <common-name>
#           "distinguishedname" "=" <distinguished-name>
#           "relativedistinguishedname" "=" <distinguished-name>
#           "operation"        "=" "add" | "modify" | "delete"
#           <user-defined-param> "=" [<type>] <parameter-value>
#
# <mailbox> ::= <local-part> "@" <domain>
# <local-part> ::= <dot-string> | <quoted-string>
# <dot-string> ::= <string> | <string> "." <dot-string>
# <quoted-string> ::= "" <qtext> ""
# <string> ::= <char> | <char> <string>
```

```

# <char> ::= <c> | "\" <x>
# <qtext> ::= "\" <x> | "\" <x> <qtext> | <q> | <q> <qtext>
# <c> ::= 0..127 less <special>, <sp>
# <special> ::= "<" | ">" | "(" | ")"" | "[" | "]" | "\" | "," |
# ";" | ":" | "@" | "" | 0..31 | 127
# ; same as rfc-821 except "."
# <sp> ::= space character (decimal 32)
# <q> ::= 0..127 less <CR>, <LF>, <"> (quote), <\> (backslash)
# <x> ::= 0..127
#
# <common-name> ::= 1*<word>
# <distinguished-name> ::= <dn-attr> "=" <dn-value> *("," <dn-attr> "="
<dn-value>)
# <dn-attr> ::= 1*<key-char>
# <key-char> ::= <a-z, A-Z, 0-9, "-" and ".">
# <dn-value> ::= <unquoted-dn-value> | <quoted-dn-value>
# <unquoted-dn-value> ::= 1*<dn-string>
# <dn-string> ::= <dn-char> | <dn-char> <dn-string>
# <dn-char> ::= <cdn> | "\" <xdn>
# <cdn> ::= 0..255 less """, ";"
# <quoted-dn-value> ::= "" <qdntext> ""
# <qdntext> ::= "\" <xdn> | "\" <xdn> <qdntext> | <qdn> | <qdn> <qdntext>
# <qdn> ::= 0..255 less <CR>, <LF>, <"> (quote), <\> (backslash)
# <xdn> ::= 0..255
# <user-defined-param> ::= 1*<word>
# <parameter-value> ::= <string-value> | <string-list-value> | <integer-value>
# <type> ::= "[" "S" | "L" | "N" "]" ; S=String,L=String-list,N=integer
# <string-value> ::= 1*(word | encoded word)
# <encoded-word> ::= "=?" charset "?" encoding "?" encoded-text "!="
# <charset> ::= ("win32" | "latin1" | "bin")
# <encoding> ::= ("B")
# <string-list-value> ::= 1#string-value
# <integer-value> ::= number
# <word> ::= *(<qtext>/<quoted-pair>)
# <qtext> ::= < any CHAR except ">, <=> >
# <quoted-pair> ::= "\" CHAR
# <CHAR> ::= <any character> ; 0-255
# <number> ::= *<num-char>
# <num-char> ::= <'0'-'9' >
#

```

Important Notes

- 1 `<index>` is the only mandatory field and must uniquely identify each record within the proprietary system. The index field is used by the JoinEngine to track record modifications and deletions.
- 2 The only value types supported are single and multi-valued strings and single valued integer fields. Value syntaxes are interpreted from the directory schema file. All data is expected in the local system code-page and will be encoded into UTF8 by the JoinEngine. Likewise all directory data will be translated to the local code-page before supplying it to the Perl script.
- 3 `<operation>` is an optional field but if specified instructs the JoinEngine on a particular action to take. The operation is normally self-determined by the JoinEngine.
- 4 `<user-defined-param>` may identify a configured attribute within the attribute flow configuration table or a directory attribute field name.
- 5 Multi-valued attributes must be specified within a single attribute definition.
- 6 Unless object-class is explicitly specified, all attribute composition and structure and naming rules must comply with the default NPLEXOrgPerson object class, which is a structural object class derived from inetOrgPerson and organizationalPerson. Attribute definitions and structure and naming rules must be obeyed. The NPLEXOrgPerson object class may be extended as required provided the schema is properly updated in the directory.
- 7 `<distinguished-name>`, if specified must be conform to RFC1779, "A string representation of Distinguished Names", S. Kille.
- 8 Field order is unimportant.
- 9 "relativedistinguishedname" is specified as the relative DN of the entry to view's root DN. If both "distinguishedname" and "relativedistinguishedname" parameters are set, one value will be arbitrarily selected.
- 10 Binary attribute value support: Binary attributes are supported over the Perl stack by adopting text encoding techniques from MIME and in particular RFC2047. Any attribute value containing binary data may be encoded into either base64 or quoted-printable data. There are no length restrictions on a value.

An encoded word is defined as follows:

```
"=?" charset "?" encoding "?" encoded-text "=?"
```

Where:

"charset" may be either "win", "latin1" or "bin"

"encoding" must be "b" (base64 encoding) and the encoded-text must be base64 encoded according to algorithm definitions from RFC2045.

Sample Data Record

The following is a very simple record for the user "John Smith".

```
@_[$count++] = "relativedistinguishedname=cn=John
Smith,ou=Marketing";
@_[$count++] = "sn=Smith;";
@_[$count++] = "givenname=John;";
```

A few points to note:

- objectclass is not specified and the default "person,organizationalPerson,inetorgperson,NPLEXOrgPerson,NPLEXNotesPerson,NPLEXExchangePerson" is therefore assumed.
- The "relativedistinguishedname" value will be prepended to the MetaView's parent DN. For example, if this is "o=ACME,c=US", the entry DN will be "cn=John Smith,ou=Marketing,o=ACME,c=US".
- In reality, attribute values are unlikely to be hard-coded as above and it is considerably more likely that values will be first loaded into Perl scalars, for example,


```
@_[$count++] = "sn=$surname";
```

 However, in the interest of simplicity, this section uses hard-coded values.

Multi-Valued Attributes

To append a multi-valued description field to this record, the following line could be added:

```
@_[$count++] = "description=[L]Mid-West Regional  
Manager,Associate Vice President";
```

The “[L]” indicates that there is a list of attribute values.

Note that the order of attribute values is irrelevant because LDAP directories do not order attribute values. LDAP directories often order attribute values when returning the result of an LDAP search – typically alphabetical or reverse-alphabetical order.

```
@_[$count++] = "description=[L]Associate Vice  
President,Mid-West Regional Manager";
```

Back-slash characters (“\”) should be used to precede naturally occurring commas in a value. Back-slash characters naturally occurring in a value must also be preceded with back-slash characters.

For example, to append a third value to the description attribute of “Emp-ID: 0Y113\AD, 003011”, the new description line becomes:

```
@_[$count++] = "description=[L]Mid-West Regional  
Manager,Associate Vice President, Emp-ID: 0Y113\\AD\  
003011";
```

Overriding the JoinEngine default object-class

If you wish to overwrite the default object-class, specify a multi-valued attribute, for example,

```
@_[$count++] =  
"objectclass=[L]person,organizationalPerson,inetorgper  
son";
```

Superfluous white spaces are to be avoided, as the JoinEngine will not strip them from data record values and will instead assume that they are part of the value specification.

For example, assume the following line is specified

```
@_[$count++] = "objectclass=[L]person,  
organizationalPerson, inetorgperson";
```

The JoinEngine will attempt to add an LDAP entry with an object-class specification containing the three values “person”, “organizationalPerson” and “inetorgperson”. That will fail because the LDAP directory will not recognize these values as legitimate object-classes.

Specifying binary attribute values

The Perl stack automatically truncates all attribute-value specifications at the first ASCII ‘\0’ character. Certain ASCII control characters also will result in unexpected behavior and therefore it is advisable that for attributes that may hold binary data, binary data values should be encoded.

Assume, for example, that the proprietary Data Server maintains JPEG images of all users and it is required to load these values into an LDAP directory. The raw jpeg image data must be encoded using the base64 encoding algorithm and supplied as follows:

```
@_[$count++] = "jpegphoto==?bin?b?0E333asss..ddP3=";
```

Where “0E333asss..ddP3” is the base64 encoded data – the other characters are delimiters as defined above.

Any attribute value may be binary encoded but it typically only makes sense when values contain non-text characters or ASCII control characters that will not pass over the Perl stack intact.

Also, binary and non-binary values may be mixed in a multi-valued attribute specification

```
@_[$count++] =  
"sn=[L]DeCharles,=?bin?b?0D487376FFHJUSCUKD..33=";
```

Deleting Attributes

To request deletion of an LDAP entry attribute, it is necessary to supply an empty attribute. The JoinEngine will not automatically delete attributes. For example, the following record entry requests the JoinEngine to delete the description attribute if it exists in the LDAP entry. If no description attribute exists, the JoinEngine ignores the request.

```
@_[$count++] = "description=";
```

It is therefore advisable when writing scripts to always supply all possible attributes with each record. Supplying empty attributes will not cause any negative effects if no attribute exists in the LDAP directory entry.

Chapter 4

Perl Script Function Interface

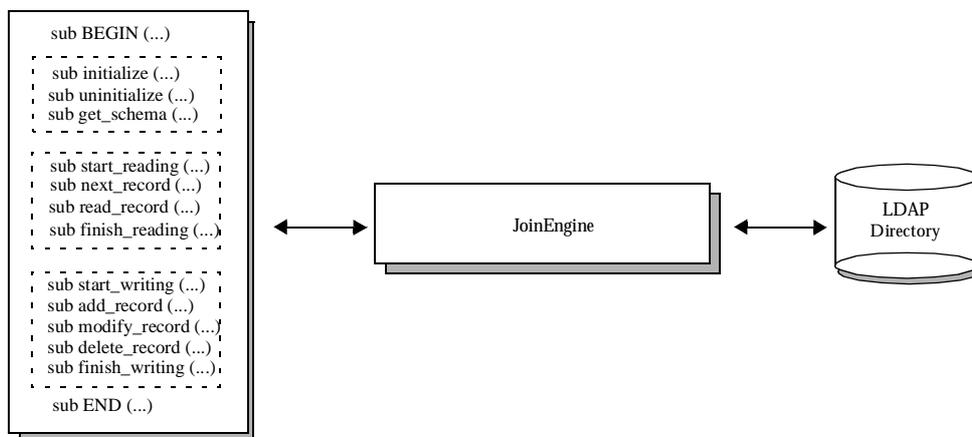
This chapter discusses various issues related to JoinEngine Perl script function interfaces, including package constructor and destructor, script initialization routines, ConnectorView (CV) to MetaView (MV) synchronization routines, and directory to CV synchronization routines.

The entries that the InJoin Meta-Directory JoinEngine manages are in either a MetaView or a ConnectorView. The MetaView is where the unified meta-directory is stored. A ConnectorView is where entries in external data repositories are stored. An entry in an external data repository is viewed by the InJoin Meta-Directory JoinEngine as being part of a ConnectorView. ConnectorViews are composed of entries held in a directory accessible by the LDAP protocol (such as the Critical Path InJoin Directory Server) or a database (such as Oracle, Sybase, DB2, or Microsoft SQL Server), or other sources such as Lotus Notes, Microsoft Exchange, or other data sources accessed by customized Perl scripts.

Overview of the Perl script function interface

Each JoinEngine script must comply with a pre-determined Perl format. There are 12 interface functions, including three script initialization functions, four functions that are responsible for reading records from the proprietary Data Server, and five functions that are responsible for writing data entries to the proprietary Data Server. The purpose of each function is described below and the relevant inputs and outputs are detailed.

Note. The interface may be extended between incremental versions of the JoinEngine. For the latest correct definition, refer in all cases to the `template.pl` file that is supplied with every release. The information contained in this chapter is more for informational and educational purposes rather than a formal reference point.



There are four overall categories of routines:

- Package Constructor/Destructor routines
- Script initialization routines
- CV to MV Synchronization routines
- MV to CV Synchronization routines

The routines belonging to each category are explained in detail below.

Package Constructor and Destructor

Perl supports two special subroutine definitions that act as package constructor and destructor routines. The “BEGIN” routine is invoked when the Perl script is parsed (that is, when the JoinEngine process is starting up). The “END” routine is invoked when the Perl script has been successfully parsed. These routines are invoked only once. These routines are useful for performing certain types of script operations such as initializing data blocks, opening/closing log files, verifying Data Server connections are functioning correctly.

sub BEGIN()

Purpose: All one-time initialization should be performed in this function. There are implicit Perl restrictions on what may be performed within this routine (for example, other package functions may not be invoked), which must be kept in mind when writing this routine. Please refer to the appropriate Perl reference guides for full discussion on this routine.

sub END()

Purpose: All one-time termination should be performed in this function. Please refer to the appropriate Perl reference guides for full discussion on this routine.

Note. END () is invoked when the script has been successfully parsed by the JoinEngine and not when the JoinEngine is being shut down.

Script Initialization Routines

sub initialize()

Purpose: This routine accepts configuration parameters and performs other initialization tasks. Script configuration parameters are supplied through this function.

Input Parameters:

```
@_ = * (<user-defined-param>="<parameter-value>")
```

Output Parameters: NULL

sub uninitialized()

Purpose: This routine is called when all operations have completed.

Input Parameters: NULL

Output Parameters: NULL

sub get_capabilities()

Purpose: This function informs the JoinEngine of the capabilities of the script.

Input Parameters: NULL

Output Parameters: @rec -array of script capability strings

* "E_DAReadNext" - does the script support traversing

* "E_DAReadInTraverse" - when traversing, will the entire record or just the entry name be returned

"E_DARead" - does the script support direct indexed entry reads

* "E_DAAdd" - does the script support the addition of new entries

* "E_DAModify" - does the script support the modification of existing entries

"E_DADelete" - does the script support the deletion of existing entries

* "E_DAAssignsName" - will the script create the entry's index or is the JoinEngine responsible for generating the entry index via the DN mapping rules

* "E_DASchema_GeneratedByPlugIn" - is the get_schema subroutine implemented

"E_DASupportsIncrementalRefresh" - does the script support incremental refresh (that is, only return entries that have been modified) or will it return all records on each synchronization cycle

(*) indicates default configuration

sub get_schema()

Purpose: This function retrieves the schema supported by the script, which is used to set up attribute-mapping tables.

Input Parameters: NULL

Output Parameters: @rec -return an array of schema in the form:

```
<oc-spec>          ::= <object-class> <attr-spec> *( <attr-spec> )
<attr-spec>        ::= <attr-prefix> <attr-name>
<attr-prefix>      ::= *( <attr-prefix-id> )
<attr-prefix-id>   ::= ( '*' ! '+' | '&' | '!' );
                   '*' indicate not mandatory
                   '+' indicate multi-valued
                   '&' indicate binary
                   '!' indicate no-user-modification permitted
<object-class>     ::= [ '*' ] <oc-name>;
                   '*' indicates a non-structural objectclass
<attr-name>       ::= LDAPString; per RFC-2251
<oc-name>         ::= LDAPString; per RFC-2251
```

Example: @_[0] = *sysuser sysusername !sysid *sysaddr

This example defines a non-structural object class named 'sysuser' with a mandatory attribute 'sysusername', a non-modifiable attribute 'sysid' and an optional 'sysaddr' attribute

CV to MV synchronization routines

These routines are responsible for retrieving information from the proprietary Data Server. According to the configured CV to directory synchronization schedule, the JoinEngine will periodically re-synchronize data entries from the proprietary Data Server to the LDAP directory.

In all cases, input and output arguments are supplied through an array of strings. Within each routine, supplied arguments may be read from the @_ array.

sub start_reading()

Purpose: The purpose of this routine is to perform any necessary CV initialization, such as opening files or sockets, initializing counter variables etc. This routine is invoked once per CV to directory synchronization cycle. If initialization fails, call `die()` and the synchronization cycle will be aborted. To operate in incremental mode (that is, propagating only deltas and no automatic deletions) return the string "mode=incremental".

Note. A failure to call `die()` when initialization fails in full-synch mode may result in the JoinEngine deleting all previously loaded entries from the LDAP view. This is because as each `next_record` fails, the JoinEngine will assume the record has been deleted.

Input Parameters: NULL or `@_[0] = "Resynch=Refresh"`

If a "refresh" re-synchronization has been requested, the input record will contain a single string indicating that a refresh of all records has been requested. If the script always operates in 'full synch mode' it can ignore this flag, as it will always return all records. If the script operates in "incremental" mode, it should if capable of doing so return all entries from the proprietary Data Server. If the script is not capable of retrieving all records, it should call `die()` and thereby abort the synchronization cycle – failure to do this will result in the JoinEngine interpreting all un-returned entries as having been deleted.

Output Parameters: NULL or `@_[0] = "mode=incremental"`

If nothing is returned, the JoinEngine will assume the script is running in full synchronization mode. If the script is operating in 'incremental' mode, the script should return the string "mode=incremental".

sub next_record()

Purpose: Called once per CV record. This routine should build the record structure and return the record to the JoinEngine. If there are no further records, call `die()`. If you wish to abort a cycle during a run, for example, if an error is encountered communicating with the CV system, return 'index=abort', in which case the JoinEngine will abort the cycle. An additional reason code, in the form of an 'AbortReason' attribute, will be logged to the JoinEngine's log file.

Note. Calling `die()` indicates to the JoinEngine that all records have been read. In full synchronization mode, the JoinEngine will assume any records that have not been returned have been deleted from the host system and will consequently delete the corresponding LDAP records. If the host system becomes unavailable for whatever reason, return an `'index=abort'` rather than calling `die()`.

Input Parameters: NULL

No input parameters are supplied to the `'next_record()'` routine.

Output Parameters: `@_ entry record`

The entry data record must comply with format defined in the previous section.

sub read_record()

Purpose: If the host system supports indexed entry reads, this routine is invoked by JoinEngine to request individual data entries. If there is a communication problem with the host system, return `"index=abort"`, in which case the JoinEngine will abort the cycle. An additional reason code may be supplied in the `"AbortReason"` attribute.

Input Parameters: `@_ = "index="<index>`

Output Parameters: `@rec`

sub finish_reading()

Purpose: Called at end of CV to MV synchronization cycle and all files or sockets opened during `start_reading()` should be closed

Input Parameters: NULL

No input parameters are supplied.

Output Parameters: NULL

No returned parameters are interpreted by the JoinEngine.

MV to CV synchronization routines

sub start_writing()

Purpose: Called at the start of directory to CV synchronization cycle and should perform whatever initialization is required to accept deltas from the directory. If initialization fails for any reason, call `die()` and the current synch cycle will be aborted.

Input Parameters: NULL

No input parameters are supplied.

Output Parameters: NULL

No returned parameters are interpreted by the JoinEngine.

sub add_record()

Purpose: Adds a directory owned entry to the proprietary Data Server. This routine takes as source a complete record and must return a locally assigned index value. The JoinEngine uses this to create a link to the originating LDAP directory entry for the purposes of tracking entry modifications and deletions. Specify an index value of "discard" if the record is not required.

Input Parameters: `@_="operation=add"`

`"distinguishedname="<distinguishedname>`

`*(<user-defined-param>="["<type>] <parameter-value>)`

Output Parameters: `@_[0] = "index" "=" 1*word; unique local index string`

`@_[0]="index=jsmith@cp.net"`

sub modify_record()

Purpose: Modify an entry within the proprietary Data Server. The entire entry is supplied and it is the responsibility of the supplied routine to determine what attributes have changed if an index is returned, the State database will be updated with the new index

Input Parameters: @_="operation=modify"

"index=" <index-field>

"distinguishedname="<distinguishedname>

*(<user-defined-param>="["<type>] <parameter-value>)

Output Parameters: NULL or \$_[0] = "index" "="
<index-field>; unique local index string

sub delete_record()

Purpose: Delete a directory-owned entry from the proprietary Data Server.

Input Parameters: @_="operation=modify"

"index=" <index-field>

"distinguishedname="<distinguishedname>

Output Parameters: NULL

No returned parameters are interpreted by the JoinEngine.

finish_writing()

Purpose: Perform any user required cleanup operations following a series of write requests

Input Parameters: NULL

No input parameters are supplied.

Output Parameters: NULL

No returned parameters are interpreted by the JoinEngine.

Chapter 5

Some Design Considerations

This chapter discusses various design considerations, not discussed elsewhere, that should be considered.

Logging

The JoinEngine writes its log files into a configurable directory from where they may be viewed remotely through the Critical Path Management Center. Filenames must conform with MDS log filename conventions.

Scheduling

Each JoinEngine has two separate synchronization schedule specifications for controlling both of the synchronization cycles (proprietary Data Server to directory and directory to local proprietary Data Server).

Renaming entry index values

There are two separate index-renaming cases, based upon entry ownership, which need consideration.

- **LDAP Directory Owned Entries:**
The first case is if the entry is directory-owned, and a directory 'modify' operation results in the index being renamed in the proprietary Data Server. In this case, the "modify_record()" Perl subroutine should return the new index to the JoinEngine (for example, @_[0] = "index=<new-index>". The JoinEngine database is updated accordingly.

-
- **Proprietary Directory Owned Entries:**
Index renames of locally owned entries are treated as entry delete and re-add operations. If a locally owned entry has been renamed in the proprietary Data Server, when the renamed entry is supplied to the JoinEngine (through Perl subroutine `'next_record()'`), the JoinEngine will treat the entry as a new entry and will delete the old entry.

Renaming entry LDAP DN values

Again there are two cases to consider when LDAP entry DNs are renamed.

- **LDAP Directory Owned Entries:**
If an entry is renamed in an LDAP directory, the rename operation will be supplied to the script. The JoinEngine will in turn process the rename operation by updating the State database with the new DN value.
- **Proprietary Directory Owned Entries:**
Locally owned entries may invoke a rename operation by supplying an alternative DN value when returning individual records from the Perl `'next_record()'` subroutine.

Note. Index values may not be changed.

Chapter 6

Miscellaneous Issues

This chapter provides information on various topics, including a list of reference books for learning Perl, a description of the relationship between ActiveState Tool Corporation's Active Perl and InJoin Meta-Directory, a discussion on LDAP schema extensions, and a checklist for producing Perl scripts.

Further Information on Perl

The following website is useful for locating downloadable installation images of Perl and various Perl packages.

- <http://www.activestate.com/ActivePerl/>

CPAN (Comprehensive Perl Archive Network) acts as the central distribution point for Perl. The entire Perl source code is available at this site. In addition, some binary installable versions are also available for some platforms.

The standard Perl `man` pages that are supplied with the standard Perl distribution are extensive and cover all relevant aspects of the language.

- <http://www.ora.com/>
- <http://perl.oreilly.com/>

For reference books on Perl, the O'Reilly Perl books are highly recommended. More information can be retrieved from the above web locations.

For novice programmers wishing to learn about Perl either of the following books is recommended:

- “Learning Perl”, O’Reilly, 2nd edition, 1997, Randal L. Schwartz, Tom Christiansen, and Larry Wall
- “Learning Perl on Win32 Systems”, O’Reilly, 1997, Randal L. Schwartz, Erik Olson, and Tom Christiansen

For more experienced programmers wishing to get to grips with the Perl language and obtain information about the standard Perl library or particular Perl packages, the following books are recommended:

- “Programming Perl”, 2nd edition, O’Reilly, 1996, Larry Wall, Tom Christiansen, and Randal L. Schwartz
- “Advanced Perl Programming”, O’Reilly, 1997, Siriam Srinivasan
- “Perl Cookbook”, O’Reilly, 1998, Tom Christiansen and Nathan Torkington
- “Perl for System Administration”, O’Reilly, 2000, David Bank-Endelman

ActiveState Tool Corporation’s ActivePerl and InJoin Meta-Directory

InJoin Meta-Directory relies on a Perl script that you supply to interact with a non-LDAP compliant data source.

The JoinEngine makes no assumptions about the format of the data source as long as the information is accessible from a user-supplied Perl script. Given the capabilities of Perl, the data source may be, for example, the NT registry or an Access database, and may be manipulated through the Perl script before being passed on to the JoinEngine.

ActivePerl is the latest Perl binary distribution from ActiveState and replaces what was previously distributed as Perl for Win32. The latest release of ActivePerl, as well as other professional tools for Perl developers, are available from the ActiveState web site at <http://www.ActiveState.com>.

The ActiveState Repository has a large collection of modules and extensions in binary packages that are easy to install and use. To view and install these packages, use the Perl Package Manager (PPM) that is included with ActivePerl.

LDAP Schema Extensions

You should avoid extending the LDAP schema unless absolutely necessary. Most supplied LDAP schemas come with a range of object classes and attributes, which often prove to be suitable (or at least usable) to most applications. The reasons for avoiding modifying schema are as follows:

- It is generally a time-consuming activity, prone to trial-and-error iterations
- Different LDAP directories from different vendors have different methods for modifying schemas and conflicts sometimes occur
- The schema will need to be updated on all directory servers that may potentially host data
- It may be necessary to register for a proper valid Object-identifier base id to avoid conflicts with other potential schema enhancements

In some cases, modification of the LDAP schema is unavoidable. If it is necessary to modify the schema, please refer to the relevant LDAP directory administrator guide for details on updating LDAP directory schemas. Also, it is advisable to define the schema enhancements in a formal document to make it easier to maintain or enhance the schema in the future.

Checklist

The following list provides a general guide to the typical steps required to develop Perl scripts.

- Understand Perl – If you have not had a lot of experience with the Perl programming language, acquire the Perl reference books and familiarize yourself with the basics of the language.
- Examine possible interfaces to the proprietary Data Server remembering that file interfaces are generally not the best interface mechanism. Other considerations include:

- Full/Incremental synchronization modes – if incremental mode is possible, it should be used
- Attribute Flow granularity requirements
- Index attribute – what attribute (or combination of attributes) uniquely and consistently identifies entries in host directories
- Potential system unavailability/downtimes – does the script need to cater for such eventualities. It may be advisable for the script to generate warning messages for the administrator?
- Schema definition – if new object classes and attribute definitions are required, properly define the extensions and update the LDAP directory's schema
- Select appropriate distinguished name mapping algorithm and LDAP directory layout.
- Write the script.
- Test the script in various likely deployment scenarios. Using separate custom-written Perl test scripts is often the best approach to testing basic functionality. End-to-end testing should also cover all basic functionality and volume/stress loads.

Also when testing the script, test basic functionality with small data loads as problems are typically just as likely to occur in the first 100 record sample as in a 50,000 record sample.
- Carefully decide the script configuration parameters and fully document all possible options.
- Document the script completely and identify and explain key design decisions and any assumptions made when writing the script.

The ActiveState Repository has a large collection of modules and extensions in binary packages that are easy to install and use. To view and install these packages, use the Perl Package Manager (PPM) which is included with ActivePerl.

Appendix A

Operations for the Proprietary Data Servers

This chapter contains three sections on:

- SAP and PeopleSoft integration, in which Perl scripts are generated to provide connectivity between InJoin Meta-Directory and SAP or PeopleSoft.
- Perl scripts, which the JoinEngine uses to interact with external data sources.

Interacting with the Perl script

A Perl script template (*Template.pl*) that describes how to create a Perl script to suit your needs is installed to the following location:

```
C:\Program Files\CriticalPath\Manager\scripts\UniversalParser
```

Note. This template can also serve as a Universal Parsing script when used in conjunction with a configuration file and a data file. For detailed information about the Universal Parsing Script, please see “Configuring the Universal Parsing Script,” which is located at
C:\Program Files\CriticalPath\Manager\scripts\UniversalParser\univconn.doc.