TECHNICAL REFERENCE GUIDE: DIRECTORY SYSTEM AGENT

Published: November 2020
Version: 7.5.1

Technical Reference Guide: Directory System Agent

For ViewDS Release 7.5.1

Document Lifecycle

ViewDS may occasionally update documentation between software releases. Therefore, please visit www.viewds.com to ensure you have the PDF with most recent publication date.

November 2020

CONTENTS

Chapter 1
About this guide

This guide provides a technical reference for the ViewDS *Directory System Agent* (DSA). It includes information about configuration parameters and functionality. The functionality can be implemented using either the *Stream Directory User Agent* (Stream DUA) or ViewDS Management Agent. The Stream DUA, however, allows you to implement functionality beyond the scope of the ViewDS Management Agent.

This chapter describes:

- Who should read this guide
- Related documents
- How this guide is organized

Who should read this guide

Read this guide if you are responsible for administration of ViewDS and need to extend it beyond the functionality available through the ViewDS Management Agent.

Before using this guide, you should be familiar with the key concepts described in the *ViewDS Directory: Installation and Operation Guide*.

Related documents

The ViewDS document set includes the following:

- *ViewDS Directory: Installation and Operation Guide*
- *ViewDS Technical Reference Guide: Directory System Agent*
- *ViewDS Technical Reference Guide: User Interfaces*
- *ViewDS Access Sentinel: Installation and Reference Guide*
- *ViewDS Management Agent Help*
- *ViewDS Access Proxy Installation Guide*

How this guide is organized

Chapter 1: About this guide
Provides an overview of this guide.

Chapter 2: ViewDS tools
Provides an overview of the ViewDS tools including the ViewDS Management Agent, Stream DUA and DSA Controller.

Chapter 3: Configuring ViewDS
Provides details of the ViewDS configuration parameters.

Chapter 4: Defining schema
Describes schema concepts and the operational attributes that define schema.

Chapter 5: Indexes, extensions and word lists
Describes indexes and word lists (synonyms, noise words) which help optimize searches on a directory, along with attribute type extensions.

Chapter 6: Managing security
Describes how ViewDS authenticates users and controls their access to directory entries. The chapter also describes LDAP password management along with other miscellaneous aspects of ViewDS security.

Chapter 7: Replicating or distributing data
Provides an overview of X.500 distributed operations, and describes how to configure DSAs for distributed operations and replication.

Appendix A: Stream DUA
Describes the Stream DUA commands and notation.

Appendix B: Supported schema
Specifies the pre-defined schema supported by ViewDS.

Appendix C: OpenSSL and SSLeay licensing
Licensing requirements for distribution of OpenSSL and SSLeay.

Appendix D: Open XML SDK licensing
Licensing requirements for distribution of the Open XML SDK.

Chapter 2
ViewDS tools

This chapter describes the tools that allow you to manage a ViewDS *Directory System Agent* (DSA). It describes:

- ViewDS Management Agent
- Stream DUA
- DSA Controller
- Other ViewDS tools

ViewDS Management Agent

The ViewDS Management Agent is a Windows-based application that allows you to manage the status of DSAs and access their configuration parameters, log files, directory data, schema, knowledge and access controls.

The application runs on a different computer to that hosting the DSA. The *Remote Administration Service* (RAS) also resides on the DSA's host, and allows the DSA to be started, stopped and configured remotely.

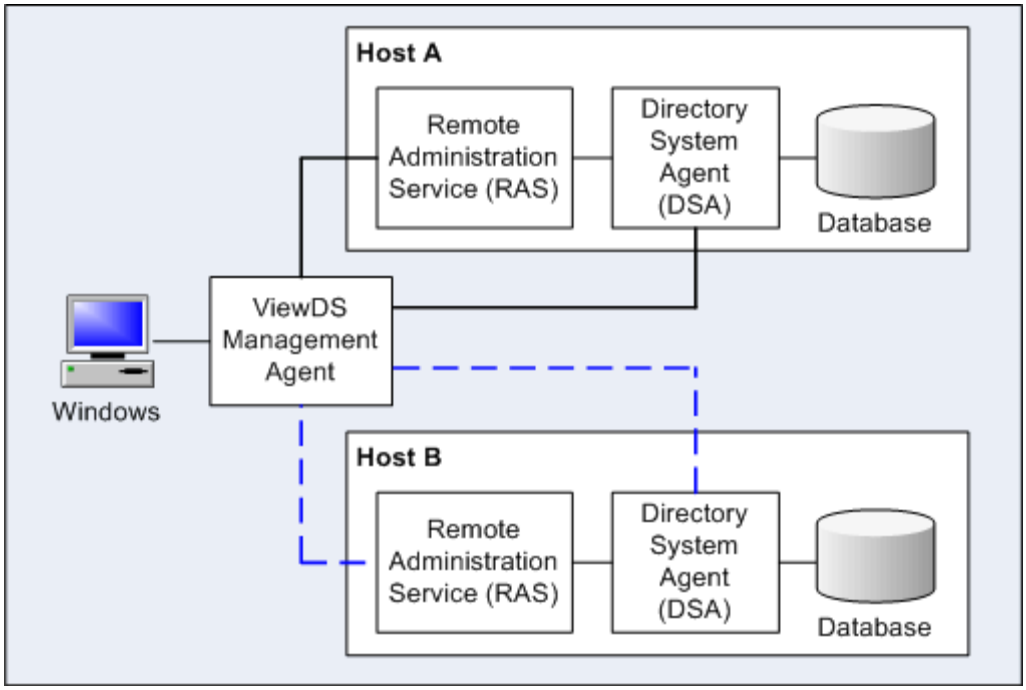


Figure 1: ViewDS Management Agent

Stream DUA

The *Stream Directory User Agent* (Stream DUA) is the main command-line tool for ViewDS. It allows you to configure schema, security and knowledge, and to bulk load and extract data. The Stream DUA commands can also process files of LDIF or ELDIF content records, and LDIF or ELDIF change records.

The Stream DUA is text-oriented and reads update and query requests from files specified on the command line or standard input stream. The requests correspond to the standard X.500, LDAP and XLDAP operations of the DSA interface together with extensions for directory administration. They are written in a structured, verbose language referred to as *Stream DUA format*.

Stream DUA format is used whenever data is extracted from the directory – for example, when a subtree is dumped, or when an update operation is logged. Because the extracted data is written in Stream DUA format, it can be reapplied to the directory (for example, to reload the directory or replay a series of update operations).

It is also useful to understand Stream DUA format because the update logs, query logs and dump files contain a series of Stream DUA commands.

Stream DUA commands

The Stream DUA commands permitted on an entry are insert, delete, rename, move, and modify.

Commands permitted on any part of the directory tree are search, compare, list, read, and dump. The dump and insert entry commands are particularly important as they allow the database to be transported from one host to another, across software versions or between different directory schemas.

For descriptions of the Stream DUA commands and input language, see *Appendix A*.

Synchronization

The Stream DUA's set command allows you to configure a DSA to synchronize with another directory. For more information, see *set-synchronization* on page 198.

Running Stream DUA

The following invokes the Stream DUA from the command line:

```
sdua [-AdimMUv8] [-a | -u username -p password] [-c commands]
     [-K key] [-l duaLocal] [-o dsaAddress] [-O ldapAddress]
     [-t vfhome] [-F ldif] { -L | -S | filename}
```

The options are described below.

`-A` Sets the `dontDereferenceAliases` bit of the `CommonArguments` sent with each operation.

`-i` Displays elapsed time of the DSA's processing of each operation.

`-d` Sets the `manageDSAIT` bit in the `CommonArguments` sent with each query or update operation. This prohibits chaining of operations and the generation of referrals. The entries which contain the knowledge required for chaining operations can be read and updated only when this bit is set.

`-m`	Suppresses reporting of successful operations. Only failed operations are reported when this option is specified.
`-M`	Behaves like `-m` except it suppresses messages about implicit binds.
`-U`	Prevents the Stream DUA from performing update operations. This is useful for checking the syntax of the input file without actually altering the database. Enquiry operations are still performed.
`-v`	Displays the Stream DUA version number.
`-8`	Causes the Stream DUA to conform to the X.500 1988 specifications as closely as possible.
`-a`	Sets the default bind credentials to none, allowing the user to bind anonymously by default.
`-u username`	Sets the default `username` that Stream DUA uses to bind to the DSA. The `username` is either the value of a `userName` attribute or the full Distinguished Name (DN) of an entry in Stream DUA notation.
`-p password`	Sets the default `password` that Stream DUA uses to bind to the DSA.
`-c commands`	Runs `commands` after Stream DUA finishes processing the `sdua.startup` file but before it processes commands from its file list. If no file list is supplied, Stream DUA exits after completing the commands. `commands` is a list of one or more Stream DUA commands separated by semicolons. The last command does not require a semicolon.
`-K key`	When processing files that update entries, specifies the files' encryption `key`. This must be supplied if the key is different from the current system key or Stream DUA is running on a remote machine.
`-l dualocal`	Uses `dualocal` as the OSI calling address (only relevant for an OSI connection) instead of the address specified by the configuration-file parameter `dualocal` (see page 41).
`-o dsaAddress`	Connects to the DSA at `dsaAddress` instead of the address defined by the configuration-file parameter `dsaaddress` (see page 41).
`-O ldapAddress`	Specifies the address of an LDAP server address. It has the same format as the `ldapaddress` option in the configuration file (see *Address parameters* on page 41). The value of the -u option should be an LDAP DN.

`-t vfhome`	Uses `vfhome` as the ViewDS root directory instead of the value set by the environment variable `${VFHOME}`.
`-F ldif`	Alters the way search results are presented. All LDAP search results are output as LDIF content records. The output for all other operations (including DAP search results) is either commented out or omitted so that the entire output file is a valid LDIF file.
`-L`	Causes Stream DUA to interpret any subsequent files named on the command line as LDIF or ELDIF files.
`-S`	Causes the Stream DUA to interpret any subsequent files named on the command line as Stream DUA format files.
`filename`	An input file of Stream DUA commands (for example, `dib.*`) or LDIF records or ELDIF records. Input files are processed in the order supplied. If no files are named on the command line, Stream DUA reads stdin.

Startup file

At startup, Stream DUA looks for the `sdua.startup` file in the `setup` directory. If the file exists, Stream DUA will process the file before its normal input.

The startup file is a good place to put a `set` command – for example, `set context`, `set base` or `set options`.

Sleep file

To reduce the load imposed by Stream DUA, specify a period for it to wait between executing each command. This is implemented by placing the `sdua.sleep` file in the current working directory (without the file, the delay is 0). The first line should contain the number of seconds that Stream DUA should ‘sleep’ between operations. The Stream DUA checks the content of the file every minute.

Interactive mode

Stream DUA operates in interactive mode if no command is entered at the command line. In interactive mode, Stream DUA displays the following to prompt for a command:

```
sdua>
```

It displays the following to prompt for parameters or values:

```
sdua>>
```

DSAIT management operations

Stream DUA supports the `manageDSAIT` service control. It modifies the semantics of an operation in the DSA so that:

- The root entry and all operational attributes can be accessed.
- Knowledge is not used but is treated as attribute information. The DSA operates as a stand-alone DSA which neither chains nor returns referrals.

The `mangeDSAIT` service control is required when reloading data from a database dump, or performing knowledge configuration (see *Replicating or distributing data* on page 147).

The service control can be set in one of the following ways:

- starting Stream DUA with the `-d` option (which also enables the DAP Admin Protocol).
- setting the `manageDSAIT` bit with the `set options` command or the `options` keyword.

Note that the `-d` option only sets the default value of `manageDSAIT`. If the `set options` command or the `options` keyword is used for other purposes, it will override this default and `manageDSAIT` must be set explicitly.

When the `manageDSAIT` bit is set, complete access is granted to all operational attributes (subject to access control) and the DSA is treated as a stand-alone DSA. The DSA does not chain or return referrals, but can return knowledge and manipulate schema as operational attributes. Used in this way, Stream DUA can set timestamps, change the `dseType` of an entry, modify schema, add or remove knowledge, and access the root entry, etc.

DAP Admin Protocol

Stream DUA normally binds to the directory using the standard X.500 Directory Access Protocol (DAP). In this mode, Stream DUA functions as a conformant X.500 DUA, and can be used to connect to a non-ViewDS DSA.

The *DAP Admin Protocol* is a superset of the standard X.500 DAP. It supports standard DAP operations plus operations that correspond to the following:

- Stream DUA commands: add, dump, empty, fill, remove, save, checkpoint, verify, dumpDIT, fillDIB, saveDB, synonym, verifyDIB, word, and checkpointLogs.
- DSA Controller commands: initialiseDBM, emptyDIB, openDBM, closeDBM, exitServer, readStatus, writeStatus, bindList and server.

Stream DUA automatically binds to the DSA using the DAP Admin Protocol when one of the additional operations is invoked. If Stream DUA is currently bound to the DSA using DAP Protocol, it will unbind and then bind using the DAP Admin Protocol.

The default bind protocol for Stream DUA can be controlled using the set command (see page 196).

Strong Authentication

Stream DUA can connect to the directory using DAP strong authentication to establish its credentials. This may only be done using the `bind` command (see page 180).

DSA Controller

The *DSA Controller* is a command-line tool that allows you to modify a DSA's operational parameters while it is running. It is only available on the ViewDS's host, and should be used when the ViewDS Management Agent is unavailable.

NOTE: If you use this tool to stop a DSA that was started through the ViewDS Management Agent, then the RAS will restart the DSA immediately.

As well as allowing you to modify the DSA's operational parameters, the DSA Controller also allows you to:

- open, close or empty the database
- list the current DSA users
- view the DSA status
- terminate the DSA

NOTE: The *DSA utility* allows you to start and stop the DSA (see page 14).

Running the DSA Controller

The DSA must be running before its operational parameters can be modified through the DSA Controller. They can also, however, be modified through the *ViewDS Fast Load* tool (see page 11) if the DSA is not running.

If necessary, the DSA can be started with the `-c` option so that it starts with the default values for its operational parameters. The following invokes the DSA Controller from the command line:

```
dsac [-t vfhome] [-g] [-o dsaAddress] [-l localAddress] [-v]
     [command ...]
```

If a command is included, the DSA Controller executes the command and exits. Without a command, the DSA Controller runs in interactive mode and reads commands subsequently entered at the command line. You can exit interactive mode using Control-D or the `quite` command.

The DSA operational parameters that can be modified through the DSA Controller are stored in the *Runtime settings file* (see page 17). For descriptions of the *DSA* , see page 18.

Options

`-t vfhome`	Uses `vfhome` as the ViewDS root directory instead of the value set by the environment variable `${VFHOME}`.
`-g`	Runs with the super-administrator identity. This option has been maintained for backward compatibility only – it is the default behaviour of the DSA Controller.
`-o dsaAddress`	Uses `dsaAddress` instead of the address specified by the configuration-file parameter `dsaaddress` (see page 38).
`-l localAddress`	Connects to `localAddress` instead of the address specified by the configuration-file parameter `dualocal` (see page 38). This is only relevant for an OSI connection.
`-v`	Prints the ViewDS version and exits.

Commands

The commands can be abbreviated to their first letter unless stated otherwise below.

`close`	Closes the database. The DSA continues to run in the same state as if launched with the `-r` switch. The database must be closed to change certain parameters or to put its files into a consistent state before a backup.
`display`	Displays status. The DSA's status includes its operational state; whether the database is open or closed; a report on each of its `dot` threads (new, idle or processing and, if processing, the state and type of operation and the protocol being used); and the values of all parameters that can be set.
`empty`	Empties the database. Removes every entry from the database except for an empty root entry. **NOTE:** Use this command with caution. *There is no abbreviation for this command*.
`help`	Lists the available commands.
`init [size]`	Initializes the *safe* file (see *safe* on page 21). `size` specifies the default size in megabytes of the *safe* file. If `size` is unspecified, the default in the operational parameters file (see page 17) is used. If the file is missing, the value 4 is used. **NOTE:** The database must be closed before this command can be used.
`open`	Opens the database. This command is necessary after a `close` command or if the DSA was started without opening the database (using the DSA Controller `-r` switch).
`quit`	Exits the tool.
`reset`	Sets all current *DSA* (see page 18) to the values in the operational parameters file (see page 17). Use this command to restore settings after making a temporary change. This command will not change the value of `cache` if the database is open. **NOTE:** The size of the *safe* file cannot be reset with this command.
`set params...`	Sets the listed *DSA* (see page 18). Each `param` is a parameter assignment of the form `name=value`. Each `name` must be a DSA parameter; each `value` must be a non-negative number or one of the strings `off`, `on`, `false`, or `true`. **NOTE:** The values are not saved to the operational parameters file (see page 17).

`setwrite params...`	Sets and writes the listed *DSA* (see page 18). This command is the same as the `set params...` command, except the new values are written into the operational parameters file (see page 17). They take effect on the DSA immediately and after it is restarted. The abbreviation for this command is `sw`.
`terminate`	Closes the database and forces the DSA to exit. The DSA Controller also exits and cannot be restarted until the DSA has been restarted.
`userlist`	Displays the users connected to the DSA.
`write params...`	Writes the listed *DSA* (see page 18). This command is the same as the `setwrite params...` command, except the new values only take effect when the DSA is restarted.

DSA Controller examples

Example DSA Controller commands are shown below.

- Open the database:
  `dsac open`
- Close the database:
  `dsac close`
- Terminate the DSA:
  `dsac terminate`
- List users currently connected to the DSA:
  `dsac userlist`

Other ViewDS tools

ViewDS also has the following tools:

- ViewDS Fast Load (vfload)
- Remote Administration Service
- Billing statistics
- Database backup
- DSA utility
- Printing DUA
- SNMP proxy agent
- smerge – sort update log files

ViewDS Fast Load (vfload)

This utility provides the same commands as the Steam DUA. However, unlike the Stream DUA, you must first stop the DSA before applying a ViewDS Fast Load command.

The benefit of this utility is that it is much faster than the Stream DUA when loading dump files or LDIF content records into a database. Its remaining commands are useful for testing and for recovering from problems, but are rarely used in an operational environment.

When a database is loaded by ViewDS Fast Load, there is no protection against fatal errors because it disables all database recovery mechanisms. If a crash or fatal error occurs, the database must be rebuilt from scratch.

When loading a large database, you may prefer to prepare a script that loads a number of `dib.*` files, copies the `ddm.*` files to safe storage, and then continues. In the event of a fatal error, you can resume the load from the last checkpoint rather than restarting from the beginning.

The schema-checking level for ViewDS Fast Load is set by the `schemachecking` parameter in the configuration file (see 48). For information about schema checking, see page 60.

By default, the utility runs with the super-administrator identity and with DSA information tree management enabled (see *DSAIT management operations* on page 7). A script can override this behaviour by including commands such as `bind` and `set`, or by using the `options` keyword.

Synopsis

```
vfload [-AimMrUv] [-a | -u username –p password ] [-c commands]
       [-K key] [-t vfhome] [-F ldif] { -L | -S | filename}
```

Options

All options are the same as those described for the Stream DUA (see page 4), except for the following:

`-r` Causes `vfload` to start with the DSA's database closed.

Remote Administration Service

The Remote Administration Service (RAS) – the `rasrv` process – allows the ViewDS Management Agent to start, stop and configure a DSA remotely. These services can also be invoked from the RAS command-line tool.

Synopsis

Windows platforms only:

```
ras [ -t vfhome ] [ -n windows-service-name ] [-s | -i | -u ]
```

All platforms (including Windows):

```
ras [ -t vfhome ] [-l seconds] [ add servicename [ servicepath
    ] | list | remove servicename | status | stop | servicename
    { status | dsa configure | dsa start [ closed ] | dsa stop
    | dsa unconfigure | license install licensefile | show
    configuration } ]
```

Options

Windows platforms only.

`-n windows-service-name`	Uses the identified service name instead of the default service name when interacting with the Windows service manager.
`-s`	Starts the RAS without using the Windows service manager.
`-i`	Installs the RAS as a Windows service.
`-u`	Uninstalls the RAS as a Windows service.

All platforms:

`-t vfhome`	Uses `vfhome` as the ViewDS root directory instead of the value set by the environment variable `${VFHOME}`.
`-l seconds`	Allows you to override the `rastimeout` parameter set in the ViewDS configuration file (page 38). This parameter sets how long the command line will wait for a response from the `rasrv` process before timing out. The default is 10 seconds, and 0 declares an indefinite time limit.

Commands

`add servicename [servicepath]`	Adds a new service (with the name `servicename`) for the RAS (`rasrv` process) to manage. When `servicepath` is omitted, the default service path of the `rasrv` process is used.
`list`	Lists the services managed by the RAS (`rasrv` process).
`remove servicename`	Removes a service from those managed by the RAS (`rasrv` process).
`status`	Lists the status of the services (and their DSA subsystems) managed by the RAS (`rasrv` process).
`stop`	Stops the RAS (`rasrv` process). If the process does not respond, the associated processes and files are cleaned up to ensure the process can start cleanly when it is started next.
`servicename status`	Reports the status of a specified service and its subsystem DSA.
`servicename dsa configure`	Adds a DSA subsystem to a service and configures the RAS to start the DSA with its database open.
`servicename dsa start [closed]`	Starts the DSA subsystem managed by `servicename` (optionally, with its database closed).
`servicename dsa stop`	Stops the DSA subsystem managed by `servicename`.

`servicename dsa unconfigure`	Removes the DSA subsystem from the service.
`servicename license install licensefile`	Installs license-key information from the specified file.
`servicename show configuration`	Displays the values of all configuration file parameters. It shows the values set in the configuration file (displayed in double quotes); and the default values for the remaining parameters (enclosed in brackets).

Billing statistics

The billing statistics utility produces a file of billing statistics for a specified period by parsing the system-activity log files. (These files are generated if the `alog` operational parameter is enabled – see page 18.)

The billing period is from 00:00:00 on a specified start date until 23:59:59 unspecified end date.

Synopsis

```
bstats [-t vfhome] -s startdate -e enddate [-d levels]
```

Options

`-t vfhome`	Uses `vfhome` as the ViewDS root directory instead of the value set by the environment variable `${VFHOME}`.
`-s startdate`	Sets the start date for billing period in the format `dd/mm/yy`.
`-e enddate`	Sets the end date for billing period in the format `dd/mm/yy`.
`-d levels`	Sets the number of levels in the user's name, which is used when aggregating the statistics for billing. If this option is not supplied then the bill assumes zero levels of names. Users with names shorter than specified will be billed separately.

Database backup

The database backup script (Solaris or Linux only) prepares the directory for an incremental or full backup, and copies backup files to a tape device.

The script closes the database, consolidates and merges the log files, creates a new set of empty log files, and reopens the database. During a full backup, it dumps the database (which continues to run) and moves the dumped `dib.*` files to the `dump/fulldump` directory.

The script should be run daily when there is no user activity. It usually runs from an entry in the administrator's `cron` table, and creates an incremental backup on weekdays and a full backup at the weekend.

An incremental backup consists of the current database files (`dbm.*`) plus the consolidated update logs that have accumulated since the last incremental or full

backup. Restoring from an incremental backup involves restoring the saved database files to disk and then replaying the update logs generated since the backup.

A full backup comprises a full dump of the database in Stream DUA *insert* format. Restoring from a full backup involves reloading using the ViewDS Fast Load (vfload) utility and replaying the update logs collected since the backup.

For more information about backup operations, see the *ViewDS Directory: Installation and Operation Guide*.

Synopsis

```
dbbackup [-t vfhome] [-c] [-f tapedevice]
```

Options

`-t vfhome`	Uses `vfhome` as the ViewDS root directory instead of the value set by the environment variable `${VFHOME}`.
`-c`	Sets to continuous operation mode. This mode avoids shutting down the directory by using the Stream DUA `save` command to make a copy the `/data` directory safely. The directory remains available and the backup reflects the state of the database when the `save` command was invoked.
`-f tapedevice`	Sets the name of the tape device for the backup files.

DSA utility

The DSA utility allows you to start and stop the DSA from the command line when the ViewDS Management Agent is unavailable.

NOTE: If you use this utility to stop a DSA that was started by the RAS (for example, through the ViewDS Management Agent) then the RAS will restart the DSA immediately.

Synopsis

```
dsa [-t vfhome] [-c] [-r] [-i | -u] [-n service-name] [stop]
```

Options

`-t vfhome`	Uses `vfhome` as the ViewDS root directory instead of the value set by the environment variable `${VFHOME}`.
`-c`	Runs the DSA with default values rather than those stored in the operational parameters file (see page 17).
`-r`	Starts the DSA without opening the database.
`-i`	Installs the DSA as a Windows service.
`-u`	Uninstalls the DSA as a Windows service.
`-n service-name`	Specifies the DSA with the given Windows `service-name`, rather than the one derived from the `keybase` parameter in the configuration file (see 38).
`stop`	Stops the DSA if it is running (or cleans up a crashed DSA).

Printing DUA

The Printing DUA extracts data from ViewDS and prepares it for other applications. It can sort data, tag data items, and insert text and other formatting information. The resulting data can then be used, for example, with a desktop publishing package to produce a printed directory listing.

The Printing DUA reads a script from a specified file that contains a sequence of enquiry operations for the DSA. The script indicates the entries and attributes to be output, the sorting parameters, and the tags, text, and formatting to be inserted. By convention, the file extension is `.ds`, and the output is directed to a similarly named file with the extension `.do`.

Synopsis

```
pdua [ -t vfhome ] [ -a | -u username –p password]
     [ -b baseobject] [ -r requestor ] [ -o address ]
     [ -e errorfile ] [ -f outputfile ] [ inputfile ]
```

Options

`-t vfhome` Uses `vfhome` as the ViewDS root directory instead of the value set by the environment variable `${VFHOME}`.

`-a` Authenticates to the DSA using anonymous credentials.

`-u username` Sets the user name with which the Printing DUA authenticates to the DSA. The user name is either the value of a `viewDSUserName` attribute or the DN of an entry in Stream DUA notation enclosed in curly brackets.

`-p password` Sets the password with which the Printing DUA authenticates to the DSA.

`-b baseobject` Sets the starting point in the DIT from where the Printing DUA generates a report. This option overrides the base-object option in the input script.

`-r requestor` Enables proxy authorisation for each request sent to the DSA, using `requestor` as the identity when access controls are evaluated.

`-e errorfile` Sets the name of the file for error messages from the Print DUA.

`-f filename` Sets the name of the file for all normal output from the Print DUA. When this option is not specified, *std out* is used.

`-o dsaAddress` Connects to `dsaAddress` instead of the address specified by the configuration-file parameter `dsaaddress` (see page 38).

SNMP proxy agent

This is a proxy SNMP agent designed to present the SNMP services of a number of DSAs as a single DSA to an SNMP manager.

Synopsis

```
proxy -p proxyAddress {-a agentAddress} [-t timelimit]
          [-r resends]
```

Options

`-p proxyAddress`	Sets the address the SNMP manager console will use to talk to the proxy. This option is mandatory.
`-a agentAddress`	Sets the address of an agent being proxied. Multiple `-a` options are permitted. If no agents are specified, the proxy looks for `${VFHOME}/setup/config` and assumes there is a DSA agent listening on the `snmpagent` address (see *Address parameters* on page 41).
`-t timelimitr`	Sets the time limit for how long the proxy will wait for a response from an agent before resending to, or timing out, the agent. The time is given in hundredths of a second. The default is 50 (half a second).
`-r resends`	Sets the number of times the proxy will try resending a request to an agent that has not responded within the time limit. The default value is 1.

smerge – sort update log files

This script (Solaris and Linux) or program (Windows) sorts the transactions in an update log according to the order in which they were committed to the database. It is required because updates are written to the update log when an operation is completed. In order to replay an update log, its contents must first be sorted according to when each update started.

If the script is run without any update logs specified at the command line, it waits for the name of the file to be entered. The output from smerge is written to *stdout*.

Synopsis

`smerge` *`filename`*`...`

Example

```
smerge unsorted-ulog > sorted-ulog
```

Chapter 3
Configuring ViewDS

This chapter includes details of the ViewDS configuration parameters. All parameters can be accessed from the command line and most can be access from the ViewDS Management Agent.

This chapter describes:

- DSA runtime settings
- Communications configuration
- ViewDS configuration file

DSA runtime settings

The Directory System Agent (DSA) runtime settings affect performance and memory requirements, and whether log files are generated. It is usually unnecessary to change the default configuration. However, some changes may help tune a system according to its host and data.

Modifying runtime settings

The DSA must be running before its runtime settings can be modified through the DSA Controller. They can, however, also be modified through the *ViewDS Fast Load* tool (see page 11) if the DSA is not running.

The DSA can be started or stopped through either:

- the Remote Administration Service (RAS) at the command line (see page 11) or the ViewDS Management Agent; or
- the DSA utility (see page 14).

Its runtime settings can be modified through either ViewDS Management Agent or the DSA Controller (see page 8).

If necessary, the DSA can be started with the DSA utility's `-c` option so that it starts with default values for the runtime settings.

Runtime settings file

The DSA's runtime settings are stored in a binary file:

`${VFHOME}/setup/cmsrv.cfg`

where `${VFHOME}` is the location of ViewDS.

dsa process

The `dsa` process is a multi-threaded process comprising the main `dsa` thread and one or more `dot` threads.

The `dsa` is normally started through the RAS (for example, using the ViewDS Management Agent) or DSA utility (see page 14). It automatically starts its subordinate processes and reports whether the DSA has started successfully.

The `dsa` threads are:

`dsa` Handles all communications with client applications and peer servers. It initiates and controls the state of the dot threads, queues and coordinates the processing of requests, and can manage the state of the database.

`dot` The `dot` (Directory Operation Thread) threads receive and process requests from the dsa. The `dot` threads will retrieve or update data from the database in order to satisfy requests from client applications. They also provide ViewDS's flexible searching capabilities.

The number of `dot` threads running can be configured; the default number of `dot` threads is three and the maximum is 5.

DSA runtime settings

The DSA runtime settings are modified using the DSA Controller's `set`, `setwrite`, or `write` commands (see page 9); or through the ViewDS Management Agent. The settings are described in alphabetical order below.

NOTE: If a setting can be accessed through the ViewDS Management Agent, the name displayed by the interface is shown in brackets.

alog

Controls whether activity records are logged. If it is set to `on`, all binds to the directory and all operations are recorded in the system-activity log (`alog.*`). This log is used to generate usage bills (see *Billing statistics* on page 11).

NOTE: This setting cannot be accessed through the ViewDS Management Agent.

async (Async mode)

Deprecated.

- In pre-7.4 systems a value of `on` (asynchronous) is recommended.
- In 7.4 and above systems this setting is not used.

bindtimeout (Bind timeout)

The time-out period for all initiated connection attempts. If a non-zero value is specified, then the DSA will abort any initiated connection attempts that have seen no activity for the number of seconds indicated by `bindtimeout`.

If this option is not set but the `disptimelimit` and `dsptimeout` options are set, then the appropriate specific limit will be used to timeout initiated connection attempts.

cache (Cache size)

Sets the size of the memory cache used by the database.

The larger the cache, the less the database will need to access the disk, and the faster the response time. The cache should therefore be made as large as possible, ideally to the point where the entire database can be held in memory.

The following is a rough calculation for the amount of available memory, and therefore the suggested cache size:

```
available memory = maximum memory - 6 - (12 x DOTs)
```

where:

- `available memory` is the amount of available memory in MB
- `maximum memory` is the maximum amount of RAM on the DSA's host in MB
- `6` is the space required in MB for the DSA process (excluding the DOT threadss)
- `12` is the space required in MB for each DOT thread
- `DOTs` is the number of DOT threads

If this calculation gives a negative number, consider adding more RAM to the host system. Otherwise, set the cache to this number.

If the cache is too small, there is a serious effect on performance. If the available memory for the memory cache is less than 10% of the expected size of the database, then the host system needs more RAM.

clog

Deprecated.

daptimeout (DAP timeout)

The time-out period for an inactive connection to a DUA. When there have been no requests from a DUA for the number of seconds defined by this setting, the connection is unbound.

If set to zero, then there is no time-out period.

If either of the following are lower than this time-out period, then they override it:

- the default time-out for the DUA (see the `defaultEntitlement` attribute in the *Technical Reference Guide: User Interfaces*)
- the time-out set in the DUA client

However, the above have no effect if they are greater than `daptimeout`.

disptimelimit (DISP time limit)

The time-out period for an idle or unresponsive DISP connection. If a non-zero value is specified, then the DSA will abort any DISP connection that has seen no activity for the number of seconds indicated by `disptimelimit`. The value should be greater than the time a total refresh would be expected to take.

The `disptimelimit` with respect to DISP behaves in a similar way to the `dsptimeout` with respect to DSP.

If set to zero, then there is no time-out period.

dots (Dot threads)

The number of `dot` (Directory Operation Thread) threads running on the DSA.

Each `dot` thread handles database queries synchronously. By having multiple `dot` threads, the DSA can process queries asynchronously – that is, multiple queries can be processed simultaneously.

There are implications to having different numbers of `dot` processes:

- 3 `dot` threads: The default setting. Having more than three DOT threads might improve throughput, but it will be at the expense of memory use.
- 2 `dot` threads: If the system is low on memory, try running with two DOT threads.
- 1 `dot` threads: This setting is appropriate for single-user operation, but it will slow throughput with multiple users. More than one DOT thread is required for replication and is strongly recommended for distributed operations in order to prevent the DSA deadlocking.

Three or four dot threads will deliver the best performance for a non-distributed DSA with a high query load.

dotsize (Max dot size)

This is the maximum size that a `dot` thread can reach before it is restarted.

The size of a `dot` thread increases according to the size of each query it receives. If a `dot` receives a query that takes it above the `dotsize`, it will handle the query and then restart. It is more efficient to have a `dotsize` that avoids this scenario.

The default is 20MB.

dsptimeout (DSP timeout)

The time-out period for an idle or unresponsive connection to another DSA. When no activity has occurred for the number of seconds defined by this setting, the connection is unbound. If set to zero, then there is no time-out period.

heapsize (Max heap size)

This specifies the maximum amount of heap memory all the DOTs combined can be allocated at any given time in MB. Operations that take the heap memory allocation over this limit will be aborted.

The default is 2048MB (except 32-bit Windows for which it is 1800MB).

key (Key)

The key used to encrypt users' passwords in dumps and logs. This setting is a 32 character hexadecimal string encoding a 16 byte AES key.

If no key is specified a default is used.

optimistic (Concurrency)

Sets optimistic concurrency mode to on or off:

- `on` – the database is in optimistic mode, which gives priority to the first update to complete (it favours updates that take less time to complete).
- `off` – the database is in locking mode, which gives priority to the first update to start (it favours updates that take more time to complete). Locking mode is recommended for a DSA in a replication agreement, because it will need to perform update transactions larger than a normal DAP/LDAP update transaction.

Default: `on` (optimistic mode).

qlog (Query logging)

Controls whether query operations are logged.

When query logging is on, all attempted query operations are written to the query log. The query log is useful when monitoring performance, tracking problems or building a file of typical queries.

The setting is normally off. If it is left on, however, the query log will grow very quickly and the available disk space will need to be carefully monitored.

recovery (Recovery)

Determines whether the database writes temporary information to disk for every update transaction. This temporary information allows the database to recover all committed transactions even if the host or database process fails during a transaction.

NOTE: For ViewDS to operate reliably, recovery must be on. Recovery can only be set to off for the current invocation of the DSA. The off setting is never saved.

safe-size (Save size)

The safe file is a database recovery log. It contains details of database transactions that are waiting to be committed. After a crash or improper shutdown, the DSA uses the safe file to recover the database before reopening it.

The safe file should be set to a size that will accommodate the largest anticipated transaction. Even though the file grows to accommodate larger transactions, it is advisable to pre-allocate disk space to ensure efficient performance.

A larger size results in faster database writes, but slower database restarts; and a smaller size results in slower writes, but faster restarts. The minimum size is 1 MB, and the default is 8 MB. (The size for the demonstration directory, Deltawing, is 2 MB.)

Changing the safe file size

The size of this file can only be changed when the database is closed. To change its size to *m* MB:

```
dsac close init m open
```

The new size is automatically recorded in the runtime settings file (see page 17) and becomes the new default for the next time the init command is used without parameters.

searchalias

Deprecated.

sessions (Max sessions)

The maximum number of simultaneous user connections to the DSA (excluding sessions by the super-administrator). The user connections may include DUAs, LDAP clients, other DSAs and the RAS.

If this setting is set to a value that is less than the current number of connections, the current connections can continue. However, no new connections are permitted until the number of connections falls below the new setting. The value `-1` means there is no limit on the number of connections.

sizefactor (SEP size factor)

This setting allows you to control the trade-off between the response time and the success of a search operation.

The setting's value is a multiplication factor that the DSA applies to whichever of the following is the smallest:

- the DSA's `sizeLimit` (see page 22); or
- the size limit in a DUA's search request.

The resulting 'calculated size limit' is the limit on the number of candidate entries the DSA will inspect during a search.

To put this in context, consider what happens when the DSA evaluates a search. It starts by applying all indexed terms in the search filter. It then determines whether the number of candidate entries is:

- less than the 'calculated size limit' – in this case, the DSA inspects each candidate entry and applies the remaining terms in the search filter.
- greater than the 'calculated size limit' – in this case, the DSA returns a partial result and the message 'size limit exceeded'.

sizelimit (Size limit)

The maximum number of entries the DSA will return in response to a search or list operation. The default value is 2000. If either of the following are lower than this size limit, they override it:

- the default size limit for the DUA (see the `defaultEntitlement` attribute in the *Technical Reference Guide: User Interfaces*)
- the size limit set in the DUA client

However, the above have no effect if they are greater than `sizelimit`.

timelimit (Time limit)

The DSA's time limit (in seconds) for a DUA user's read, compare, search and list operations. A normal value is 5 seconds. A value of -1 means there is no time limit. If either of the following are lower than this time limit, they override it:

- the default time limit for the DUA (see the `defaultEntitlement` attribute in the *Technical Reference Guide: User Interfaces*)
- the time limit set in the DUA client

However, the above have no effect if they are greater than `timelimit`.

ulog (Update logging)

Determines whether update operations are logged. The update log contains all users' update operations (add, remove, modify, move or rename an entry) and is critical for maintaining database integrity after a failure. After restoring a backup, replaying this log updates the database according to all committed transactions since the backup was made.

It is essential to log updates for database recovery from a backup (see the *ViewDS Directory: Installation and Operation Guide*). This setting should always be on.

updates (Max updates)

The number of DOT threads that process update operations simultaneously. The maximum number of updates:

- cannot exceed the number of DOT threads.
- should be less than the number of DOT threads to ensure that some DOT threads are always available for queries. Otherwise, during heavy updating, the directory may be too slow in responding to queries.
- should be set to 1 for normal operation.
- should be set to 0 to disable updates to the database.
- should be greater than 1 for a DSA in a replication agreement. (For distributed operations, this setting avoids deadlocks while processing a chained update operation. For replication, this setting avoids DAP/LDAP update operations having to wait for considerably longer than normal while replication updates are processed.)

Communications configuration

The ViewDS processes communicate with each other using either an OSI Stack or the ViewDS TCP/IP-based Lightweight Stack (LWS). Both are configured through the addressing parameters in the configuration file.

This section describes the different ViewDS communications options. It describes addressing and address formats, how to set up LDAP access, and how to enable OSI-based communications between ViewDS components and between ViewDS and other X.500 products. It describes:

- ViewDS and OSI
- Configuring for LDAP access
- Configuring for XLDAP access
- Configuring for SNMP access
- Configuring for SPML access
- Addressing

ViewDS and OSI

The DSA and Stream DUA include a built-in upper layer OSI stack. Both automatically communicate over OSI if the address they are connecting to is an OSI address.

Installation and configuration

ViewDS's OSI stack needs no special configuration other than setting up OSI addresses as follows:

- Set the `dsaaccesspoint` and `dualocal` parameters (see *Address parameters* on page 41) to be OSI presentation addresses.
- If the DSA has knowledge of other DSAs and will use OSI to communicate with them, set up (or change) the knowledge references (see *Chapter 7*) and `myAccessPoint` operational attribute to be OSI presentation addresses.

Reserved ports and RFC 1006

If you are using *RFC 1006* and have configured the listening port to a value below 1024, run the DSA as root. This is necessary because most host operating systems will not allow user processes to listen on a reserved port below 1024.

Configuring for LDAP access

The native X.500 access protocol is Directory Access Protocol (DAP). The Lightweight Directory Access Protocol (LDAP) can also be used to connect to an X.500 directory such as ViewDS.

LDAP is an internet protocol defined by the Internet Engineering Task Force (IETF). It offers similar functionality to X.500 (1988), but has a simpler treatment of data types (by treating everything as strings) and makes no use of an OSI stack (sending LDAP PDUs directly over TCP/IP).

The LDAP protocol defined in *RFCs 1777–79* is referred to as LDAP Version 2 (LDAPv2). A new version of LDAP known as version 3 (LDAPv3) is defined in *RFCs 2251–55*, and an update to these specifications is defined in *RFCs 4510–19*.

ViewDS supports LDAPv2 and LDAPv3 client access to the DSA. LDAPv2 support does not include CLDAP, Kerberos, or LDAP referrals. LDAPv3 support includes UTF-8 encoding, referrals as URLs, all protocol changes and all syntax representation changes.

Enabling LDAP access

To enable direct LDAP access:

- Define a value in the configuration file (see page 38) for `ldapaddress`, or `sldapaddress` if SSL LDAP is required, or both.
- For Solaris or Linux, if `ldapaddress` or `sldapaddress` specify a port below 1024 (such as the IANA-assigned standard port for LDAP of 389), ensure the DSA is started as root. This is necessary because these operating systems will not allow a user process to listen on a port number below 1024.

LDAP controls

LDAP Version 3 defines a mechanism for extending the functionality of LDAP operations. A number of LDAP controls have been defined in RFCs and Internet Drafts. The following controls are included in the ViewDS LDAP implementation.

LDAP control	Description
Server-side sorting of search results (as defined by *RFC 2891*)	Allows an LDAP client to request the directory server to sort the results to a search request before returning them to the LDAP client.
Paged search results (as defined by *RFC 2696*)	Allows an LDAP client to request the directory server to only return a subset, or page of the results, of a search request. The LDAP client can then repeat the request, asking for the next page of results. The size of the page (the number of entries in each page of results) is specified by the LDAP client in its request.

LDAP control	Description
Proxied authorization (as defined by Internet Draft *draft-weltman-ldapv3-proxy-09.txt*)	Allows an LDAP client with suitable permissions to make LDAP requests on behalf of different users without having to bind as each of the users. This is designed for an application which uses the directory as a repository of information where the application takes responsibility for authenticating its users and wishes the directory to apply authorization decisions on its accesses to the directory data as if it were that user.
Password policy management (as defined by Internet Draft *draft-behera-ldap-password-policy-05.txt*)	Provides enhanced security to LDAP clients and applications. For further information, see *LDAP password management* on page 129.
Virtual List View (as defined by Internet Draft *draft-ietf-ldapext-ldapv3-vlv-09.txt*)	A Virtual List View is a way to return a set of data to a third-party application. The set of data is specified by either an LDAP or XLDAP search operation. For example, an email client can be configured to make an LDAP connection to ViewDS, extract the entries identified by a Virtual List View defined at the DSA, and use them to populate its address book.

LDAP Extended Operations

LDAP Version 3 defines a mechanism for extending the functionality of the LDAP protocol by defining new extended LDAP operations.

The ViewDS LDAP implementation includes the LDAP Extension for Transport Layer Security (as defined by *RFC 2830*).

ViewDS extensions

ViewDS's implementation of LDAP Version 2 includes extensions to the LDAP specification of *RFCs 1777–1779*:

- The functionality of the X.500 (1993) ModifyDN operation is available: non-leaf entries can be renamed, and entries or whole subtrees can be moved. There is no protocol change needed to rename non-leaf entries – ViewDS simply allows it whereas strict LDAP would refuse it. To move an entry or subtree to a new superior requires the name of the new superior to be provided as though the LDAP `ModifyDNRequest` PDU were defined as:

```
ModifyDNRequest ::= [APPLICATION 12] SEQUENCE {
     entry           LDAPDN,
     newrdn          RelativeLDAPDN,
     deleteoldrdn    BOOLEAN,
     newSuperior     [0] LDAPDN OPTIONAL }
```

- Attributes with an unknown syntax (or a syntax without a text mapping) are returned as text strings in ASCII hex format if the configuration-file parameter `ldapasciihex` is `on` (it is `off` by default) and the connection is using LDAPv2. This includes attributes with `Certificate` syntax.

The behaviour of some LDAP clients does not fully conform to LDAP specifications. ViewDS will, where possible, include non-standard extensions to support these LDAP clients. The configuration-file parameter `strictldap` allows these non-standard extensions to be disabled and enforces strict compliance to the LDAP specification. This option is `off` by default.

Configuring for XLDAP access

The XML Lightweight Directory Access Protocol (XLDAP) is one of the XML Enabled Directory (XED) protocols and is defined in the IETF Internet Draft, XED: Protocols (*draft-legg-protocols-xx.txt*).

XLDAP is an internet protocol based largely on LDAP Version 3 and defined by the Internet Engineering Task Force (IETF). XLDAP offers the same functionality as LDAP Version 3, but encodes PDU's using an XML-based encoding rule, Robust XML Encoding Rules (RXER). XLDAP does not use an OSI stack; it sends XLDAP PDUs directly over TCP/IP using a SOAP or IDM-style framework.

ViewDS supports XLDAP access to the DSA either using a SOAP or IDM-style framework. Both frameworks operate over a TCP/IP connection.

Enabling XLDAP access

To enable direct XLDAP access in the DSA:

- Define a value in the configuration file (see page 38) for the parameter `soapaddress` or `xldapaddress` depending on the transport framework to be used. Alternatively, use both.
- For Solaris or Linux, if `xldapaddress` or `soapaddress` specifies a port below 1024, ensure the DSA is started as root. This is necessary because these operating systems will not allow a user process to listen on a port number below 1024.

XLDAP controls

LDAP Version 3 defines a mechanism for extending the functionality of the LDAP operations. A number of LDAP controls have been defined in RFCs and Internet Drafts. The following LDAP Controls are implemented in the ViewDS XLDAP implementation.

XLDAP control	Description
Server-side sorting of search results (as defined by *RFC 2891*)	Allows an XLDAP client to request the directory server to sort the results to a search request before returning the results to the XLDAP client.
Paged search results (as defined by *RFC 2696*)	Allows an XLDAP client to request the directory server to only return a subset or page of the results of a search request. The XLDAP client can then repeat the request, asking for the next page of results. The size of the page (the number of entries in each page of results) is specified by the XLDAP client in its request.
Proxied authorization (as defined by Internet Draft *draft-weltman-ldapv3-proxy-09.txt*)	Allows an XLDAP client with suitable permissions to make XLDAP requests on behalf of different users without having to bind as each of the users. This is designed for an application which uses the directory as a repository of information where the application takes responsibility for authenticating its users and wishes the directory to apply

XLDAP control	Description
	authorization decisions on its accesses to the directory data as if it were that user.
Password policy management (as defined by Internet Draft *draft-behera-ldap-password-policy-05.txt*)	Provides enhanced security to XLDAP clients and applications. For further information, see *LDAP password management* on page 129.
XLDAP Attribute Selection Control	The Attribute Selection control is defined in the XED specification, which is yet to be published with the IETF. In LDAP, a client can include the asterisk character value (`*`) of the attribute selection in a search request. This character indicates that all user attributes are to be returned. Many implementations also use the plus character (`+`) to indicate that they would like all operational attributes to be returned. As the 'attributes' field of an XLDAP `SearchRequest` expects Object Identifiers (as opposed to a string value in LDAP), the '`*`' and '`+`' characters cannot be used to request all users and/or operational attributes. This XLDAP control allows you to specify to the DSA that all user and/or operational attributes are to be returned.
Virtual List View (as defined by Internet Draft *draft-ietf-ldapext-ldapv3-vlv-09.txt*)	A Virtual List View is a way to return a set of data to a third-party application. The set of data is specified by either an LDAP or XLDAP search operation. For example, an email client can be configured to make an LDAP connection to ViewDS, extract the entries identified by a Virtual List View and use them to populate its address book.

XLDAP extended operations

XLDAP does not currently support any extended operations.

Configuring for SNMP access

A ViewDS DSA may be configured as an agent for the Simple Network Management Protocol (SNMP). The DSA supports Community-based SNMP Version 2 (SNMPv2c) as defined in *RFC 1901*, and supports all the managed objects from the Network Services Monitoring MIB (*RFC 2248*) and Directory Server Monitoring MIB (*RFC 2605*).

To enable the DSA as an SNMP agent, define a value in the configuration file (see page 38) for the parameter `snmpagent`. This option determines the address of the UDP port the DSA will listen on for SNMP requests from an SNMP manager or SNMP proxy agent. The value for this parameter is an address in the form `host:port`. For example:

```
snmpagent = localhost:3000
```

The SNMP manager needs to be configured to poll this address for information about the ViewDS DSA.

The maximum size for an SNMP request that the DSA agent can receive is 4096 bytes. The maximum size for an SNMP response that the DSA agent will send is set by the configuration-file parameter `snmpmaxpdusize` (see page 38). If this option is omitted, the default maximum size for responses is 484 bytes.

One of the managed objects defined by *RFC 2248* is an application index, which is used to distinguish between the various applications being monitored. Each DSA being monitored counts as one application and must have a unique application index. The application index for the DSA is set by the configuration-file parameter `snmpapplindex`. Its value must be a number greater than zero (the default is 1). Normally, the application index only needs to be set for the second and subsequent DSAs being monitored, if any.

Multiple DSAs

If you have multiple DSAs, each can be set up as an independent SNMP agent. Alternatively, ViewDS includes an SNMP proxy agent that allows multiple DSAs to be monitored through a single agent – the SNMP proxy agent (see page 16).

Configuring for SPML access

ViewDS supports the DSMLv2 profile of SPML version 2.0 (SPMLv2). SPMLv2 requests are received on the DSA's SOAP address port (see page 42). This port is the only configuration required for an existing directory tree. Even though SPMLv2 could be used to create a directory tree from scratch, it is simpler to create an initial organisation entry and make it a *subschema administrative point* using the ViewDS Management Agent.

With respect to the DSMLv2 profile, the mandatory core operations of SPMLv2 are supported – `listTargetsRequest`, `addRequest`, `modifyRequest`, `deleteRequest` and `lookupRequest`. Of the optional capabilities only the search (`searchRequest`) and suspend (`suspendRequest`, `resumeRequest` and `activeRequest`) capabilities are currently supported.

listTargetsRequest

The `listTargetsRequest` reports two targets, named Directory/DN and Directory/ID, which differ only in how provisioning service objects (PSOs) – i.e. directory entries – are identified:

- Directory/DN target – PSO identifiers are always LDAP Distinguished Names.
- Directory/ID target – PSO identifiers are client provided strings or UUIDs.

Although there are two targets, there is only one collection of entries. The advantage of the Directory/ID target is that the PSO identifiers are immutable. However, the use of LDAP DNs as PSO identifiers, as in the Directory/DN target, is more commonly used by other implementations. The Directory/DN target is slightly more efficient than the Directory/ID target.

addRequest

The `addRequest` creates a new entry. An SPMLv2 client has the option to provide a PSO identifier for the new entry with the `addRequest`. The requirements for the identifier differ, however, according to the target used to create the entry – as does the DSA's behaviour when the client does not provide it.

Directory/DN target

If the client supplies a PSO identifier for a new entry, it must be an LDAP DN.

If the client does not supply a PSO identifier, the DSA assigns naming attributes for the new entry's RDN. It does so by selecting from the set of attributes provided in the `addRequest` in accordance with the schema (name forms and structure rules) that apply at the container entry nominated in the `addRequest`.

The DSA selects the first name form that fits with the provided attributes. If more than one name form is applicable, then the SPMLv2 client can only control which is actually used by providing the new PSO's identifier in the `addRequest`.

Directory/ID target

If the client supplies a PSO identifier for a new entry, it can be any character string. The DSA will check that the string is unique across the entire collection of directory entries. If it is not unique, the request will fail. Otherwise, the DSA will subsequently use the string as the entry's PSO identifier when it is accessed through the Directory/ID target.

If the client does not supply a PSO identifier, the DSA will generate and return a UUID URN to identify the PSO, which will also be the UUID in the entryUUID attribute of the underlying directory entry. The DSA will subsequently use the UUID URN as the entry's PSO identifier when it is accessed through the Directory/ID target.

The new entry's RDN is determined in the same way as under the Directory/DN target.

Accessing the new entry

Irrespective of which target is used to create an entry, it can be subsequently accessed through either. The Directory/DN target always requires the LDAP DN of the entry. If the client provided a string as the PSO identifier, then the Directory/ID target will always use that string; otherwise, it will use the entry's UUID.

suspendRequest and resumeRequest

The `suspendRequest` and `resumeRequest` allow a user account to be suspended or reactivated later at a specified time (the effective date).

SPMLv2 does not have a standardised method to either examine or cancel pending requests. However, ViewDS applies the following rule to provide more control to an SPMLv2 client:

> When a suspend or resume request is successfully accepted, any pending suspend or resume request with an effective date that is the same as, or after, the effective date of the current request is cancelled.

A non-trivial sequence of suspend and resume events can be set up by issuing the requests in the order of ascending effective dates. ViewDS can queue, and action at the nominated time, any number of such requests. The entire sequence can be cancelled by issuing a suspend or resume request without an effective date (which defaults to the current time). A trailing part of the sequence can be cancelled by issuing a suspend or resume request with an effective date equal to the beginning of the trailing sequence.

If a request's effective date is earlier than the current time at the DSA, then the effective date is taken to be the current time.

Addressing

Protocol stacks

ViewDS processes communicate with each other using a layered protocol stack. At the top of the stack are the X.500 protocols: Directory Access Protocol (DAP), Directory System Protocol (DSP), etc.

These protocols may be carried over either an OSI Stack, IDM Stack, XIDM Stack, HTTP, an IDM style transport for XLDAP, LDAP, SLDAP or ViewDS's TCP/IP-based Lightweight Stack (LWS). For each of the stacks, there are further choices as to the transport and network used to carry the communications.

There are ten possible combinations of stack and transport/network in a ViewDS installation:

- OSI stack using *RFC 1006* (with *RFC1277* addressing)
- LWS using TCP/IP
- LWS using Unix-domain sockets
- LDAP using TCP/IP
- SLDAP using TCP/IP
- IDM using TCP/IP
- XIDM using TCP/IP
- IDM style transport using TCP/IP as a transport for XLDAP
- SOAP using TCP/IP (using HTTP) as a transport for XLDAP
- SOAP using TCP/IP (using HTTP) as a transport for SPML

NOTE: The TCP/IP combinations can use either TCP/IP version 4 or version 6.

Each of above combinations has a specific address format (except for the last two which both use the same address format).

The DSA must have a `dsaaddress` defined and this is usually a Lightweight Stack address. All other settings depend on what is in the configuration file – the default configures OSI, IDM, LDAP, XLDAP, SOAP and Lightweight Stack ports.

Presentation addresses

Addresses in an OSI application like X.500 are called *presentation addresses*. A presentation address is the address of an OSI application-layer communications entity, and consists of three optional *selectors* called the p-selector, s-selector, and t-selector (for presentation, session, and transport), and one or more *network addresses*.

```
PresentationAddress ::= SEQUENCE {
      pSelector          [0] OCTET STRING OPTIONAL,
      sSelector          [1] OCTET STRING OPTIONAL,
      tSelector          [2] OCTET STRING OPTIONAL,
      nAddresses          [3]    SET SIZE (1..MAX) OF OCTET STRING}
```

The `pSelector`, `sSelector`, and `tSelector` arguments are arbitrary strings, typically only a few characters long or absent altogether. The `nAddresses` argument is a list of network addresses, each for a different network; however, ViewDS will make use of only the first address in the list if multiple addresses are given.

For example:

```
      address {
          sSelector '0403'H,
```

```
        tSelector '0402'H,
        nAddresses { '49520086FF01'H }
    }
```

The Lightweight Stack does not make use of the selectors (they are ignored if present).

Network addresses

The `nAddresses` field in a presentation address holds one or more network addresses (NSAPs). An NSAP is a maximum of 20 bytes (40 hexadecimal digits) in length – its format is described in *X.213 Annex A and ISO 8348 Addendum 2* – and includes the following components:

- AFI – Authority and Format Identifier (two digits)
- IDI – Initial Domain Identifier (fixed length which depends on AFI)
- DSP – Domain Specific Part (variable length)

The AFI is a two-digit number in the range 36 to 59. It defines the authority that allocates the remainder of the NSAP, indicates whether leading zeros in the IDI are significant, and whether the DSP has a binary or decimal or other syntax.

The AFI and IDI together are referred to as the IDP (Initial Domain Part). The IDP is given in BCD-encoded decimal digits, which identify the addressing authority for the remainder of the NSAP.

URI representation

The format shown above is a suitable for specifying an address using the Stream DUA (for example, when configuring knowledge). However, a simplified address representation, based on the Uniform Resource Identifier (URI), can be used when specifying an address in the configuration file (see page 38), with Stream DUA or from the command line.

The URI representation can take the following forms:

- OSI-RFC1006 with one nAddress
- OSI (general)
- Deprecated string representation

OSI-RFC1006 with one nAddress

This form is for an *RFC 1006* address with a single network address. The address is constructed as follows:

`osi://host[:port][/tsel[,ssel[,psel]]]`

where:

- `host` is either a host name or a host address in dotted decimal notation.
- `port` is the IP port. The default is port 102.
- `tsel`, `ssel` and `psel` are the transport, session and presentation selectors.

The latter are either:

- Strings of hex digits (upper or lower case) followed by the letter 'H' (if only H is present, the selector is present as an empty string).
- An ASCII string of alphanumeric characters plus '-' and '_' not ending in 'H'.

OSI (general)

This is a general representation for any presentation address constructed as follows:

```
osi:{nAddresses}[/tsel[,ssel[,psel]]]
```

The selectors are strings of hex digits (upper and lower case) followed by the letter H (if only H is present, the selector is present as an empty string).

For example:

```
osi:{49520086FF01H}/0101H,,0202H
```

Deprecated string representation

The string representation described here is maintained for backward compatibility. The URI representation described above should be used where possible.

The simplified string representation for an arbitrary presentation address consists of four comma-separated items, the p-selector, s-selector, and t-selector (for presentation, session, and transport), and one or more *network addresses*. Each item is given in hexadecimal with a trailing 'H', and is omitted if absent. The network address list is itself enclosed in braces with subsequent addresses separated by commas.

For example, the following presentation address:

```
osi:{49520086FF01H}/0101H,,0202H
```

Has the following string representation:

```
{,0202H,0101H,{49520086FF01H }}
```

For Lightweight Stack (LWS) addresses, a much simpler string representation is available (see following subsection).

Address formats

Each combination of stack and transport/network protocols has an address format distinguished by its internal structure. Each combination is described below.

OSI stack using RFC1006 with IPv4

AFI	IDI	DSP prefix	DSP Address
54	00728722	03	• Four-octet IP address, with three digits per octet. • Port number (optional, defaults to 102, omit for calling addresses). • Transport set (1 for TCP; 2 for UDP). If absent, TCP is implied and if the port is present a single hex 'F' digit is added (which packs out the DSP to a full octet).

For example, the OSI–RFC 1006 address of the host 137.147.17.39, listening on port 1025, is represented as:

```
address {
      nAddresses {'540072872203137147017039010 25F'H}
}
```

And the string representation is: `osi://137.147.17.39:1025`

OSI stack using RFC1006 with IPv6

This format is based on Internet Draft *draft-pandey-osidirectory-ipv6-nsapa-format-00.txt* because the alternative encodings offered in *RFC 1888* do not allow for the port number to be included in the NSAP address.

AFI	IDI	DSP prefix	DSP Address
35	99	None	• Sixteen-octet IP address, hex encoded. • Hex encoded port number (optional, defaults to port 102, omit for calling addresses).

For example, the OSI–RFC 1006 address of the host fe80::2b0:d0ff:fed0:d730, listening on port 1025, is represented as:

```
address {
      nAddresses {'3599FE8000000000000002B0D0FFFED0D7300401'H }
}
```

And the string representation is: `osi://[fe80::2b0:d0ff:fed0:d730]:1025`

LWS using IPv4

AFI	IDI	DSP prefix	DSP Address
40	07133999	11	• Four-octet IP address, with three digits per octet. • Port number. • Final digit is hex ‘F’.

For example, the LWS–TCP/IP address `dsa.deltawing.com.au:3000` where `dsa.deltawing.com.au` is 137.147.17.39 is represented as:

```
address {
      nAddresses { '400713399911137147017039030 00F'H }
}
```

And the string representation is either of the following:

```
vlws://137.147.17.39:300
```

```
vlws://dsa.deltawing.com.au:3000
```

LWS using IPv6

AFI	IDI	DSP prefix	DSP Address
40	07133999	21	• Sixteen-octet IP address hex encoded. • Hex encoded port number.

For example, the LWS IPv6 address dsa.deltawing.com.au for port 3000, where dsa.deltawing.com.au has the IPv6 address fe80::2b0:d0ff:fed0:d730, is represented as:

```
address {
      nAddresses {
           ‘400713399921FE8000000000000002B0D0FFFED0D7300BB8’H
      }
}
```

And the string representation is:

```
vlws://[fe80::2b0:d0ff:fed0:d730]:3000
```

```
vlws6://dsa.deltawing.com.au:3000
```

LWS using Unix-domain sockets

AFI	IDI	DSP prefix	DSP Address
40	07133999	10	• The remaining digits encode the ASCII integer value of each character of the socket pathname, using three digits per character. • If there is an odd number of ASCII characters, a final digit is added with value hex ‘F’.

For example, the LWS-Unix address `/tmp/x500local` is represented as:

```
address {
      nAddresses {
         '400713399910047116109112047120053048048108111099097108'H }
}
```

And the string representation is:

```
vlws:/tmp/x500local
```

LDAP address using IPv4

AFI	IDI	DSP prefix	DSP Address
54	00728722	11	• Four-octet IP address, with three digits per octet. • Port number (optional, defaults to port 389). • Final digit is hex ‘F’.

For example, the LDAP address of the host 137.147.17.39, listening on port 1025, is:

```
address {
      nAddresses {'5400728722111371470170390 1025F'H}
}
```

And the string representation is: `ldap://137.147.17.39:1025`

LDAP address using IPv6

AFI	IDI	DSP prefix	DSP Address
40	07133999	22	• Sixteen-octet IP address, hex encoded. • Hex encoded port number (optional, defaults to port 389, omit for calling addresses).

For example, the LDAP address of the host fe80::2b0:d0ff:fed0:d730, listening on port 1025, is represented as:

```
address {
      nAddresses {
          '400713399922FE8000000000000002B0D0FFFED0D7300401'H
      }
}
```

And the string representation is: `ldap://[fe80::2b0:d0ff:fed0:d730]:1025`

LDAP address using Unix-domain Sockets

AFI	IDI	DSP prefix	DSP Address
54	00728722	12	• The remaining digits encode the ASCII integer value of each character of the socket pathname, using three digits per character. • If an odd number of ASCII characters are used, a final digit is added with value hex ‘F’.

For example, the LDAP-Unix address `/tmp/x500local` is:

```
address {
      nAddresses {
         '54007287221204711610911204712005304804810811109909710 8'H }
}
```

And the string representation is: `ldap:/tmp/x500local`

SLDAP address using IPv4

AFI	IDI	DSP prefix	DSP Address
40	07133999	12	• Four-octet IP address, with three digits per octet. • Port number. • Final digit is hex ‘F’.

For example, the SLDAP address of the host 137.147.17.39 listening on port 1025 is:

```
address {
      nAddresses {'400713399912137147017039010 25F'H}
}
```

And the string representation is: `sldap://137.147.17.39:1025`

SLDAP address using IPv6

AFI	IDI	DSP prefix	DSP Address
40	07133999	24	• Sixteen-octet IP address, hex encoded. • Hex encoded port number.

For example, the SLDAP address of the host fe80::2b0:d0ff:fed0:d730, listening on port 1025, is represented as:

```
address {
      nAddresses {
          '400713399924FE8000000000000002B0D0FFFED0D7300401'H
      }
}
```

And the string representation is: `sldap://[fe80::2b0:d0ff:fed0:d730]:1025`

IDM address using IPv4

AFI	IDI	DSP prefix	DSP Address
54	00728722	10	• Four-octet IP address, with three digits per octet. • Port number. • Final digit is hex ‘F’.

For example, the IDM address of the host 137.147.17.39 listening on port 1025 is:

```
address {
      nAddresses {'5400728722101371470170390 1025F'H}
}
```

And the string representation is: `idm://137.147.17.39:1025`

IDM address using IPv6

AFI	IDI	DSP prefix	DSP Address
40	07133999	23	• Sixteen-octet IP address, hex encoded. • Hex encoded port number.

For example, the IDM address of the host fe80::2b0:d0ff:fed0:d730, listening on port 1025, is represented as:

```
address {
      nAddresses {
          '400713399923FE800000000000002B0D0FFFED0D7300401'H
      }
}
```

And the string representation is: `idm://[fe80::2b0:d0ff:fed0:d730]:1025`

XIDM address using IPv4

The XML Internet Directly Mapped (XIDM) protocol is an IETF protocol defined in the *XED: Protocols* document (*draft-legg-xed-protocols-xx.txt*). XIDM differs from the IDM protocol only in that the protocol operations are encoded using the Robust XML Encoding Rules (RXER) instead BER. In View500, XIDM is supported for the DAP and DSP protocols.

AFI	IDI	DSP prefix	DSP Address
40	07133999	14	• Four-octet IP address, with three digits per octet. • Port number. • Final digit is hex ‘F’.

For example, the XIDM address of the host 137.147.17.39 listening on port 1025 is:

```
address {
      nAddresses {'400713399914137147017039010 25F'H}
}
```

And the string representation is: `xidm://137.147.17.39:1025`

XIDM address using IPv6

The XML Internet Directly Mapped (XIDM) protocol is an IETF protocol defined in the *XED: Protocols* document (*draft-legg-xed-protocols-xx.txt*). XIDM differs from the IDM protocol only in that the protocol operations are encoded using the Robust XML Encoding Rules (RXER) instead BER. In View500, XIDM is supported for the DAP and DSP protocols.

AFI	IDI	DSP prefix	DSP Address
40	07133999	26	• Sixteen-octet IP address, hex encoded. • Hex encoded port number.

For example, the XIDM address of the host fe80::2b0:d0ff:fed0:d730, listening on port 1025, is represented as follows.

```
address {
      nAddresses {
          '400713399926FE800000000000002B0D0FFFED0D7300401'H
      }
}
```

And the string representation is: `xidm://[fe80::2b0:d0ff:fed0:d730]:1025`

XLDAP address over IP4

The XML Lightweight Directory Access Protocol (XLDAP) is an IETF protocol defined in the *XED: Protocols* document (*draft-legg-xed-protocols-xx.txt*). XED protocol defines two transport mechanisms for the XLDAP protocol.

AFI	IDI	DSP prefix	DSP Address
40	07133999	15	• Four-octet IP address, with three digits per octet. • Port number. • Final digit is hex ‘F’.

For example, the XLDAP address of the host 137.147.17.39 listening on port 1025 is:

```
address {
      nAddresses {'400713399915137147017039010 25F'H}
}
```

And the string representation is: `xldap://137.147.17.39:1025`

XLDAP address over IPv6

AFI	IDI	DSP prefix	DSP Address
40	07133999	27	• Sixteen-octet IP address, hex encoded. • Hex encoded port number.

For example, the XLDAP address of the host fe80::2b0:d0ff:fed0:d730, listening on port 1025, is represented as:

```
address {
      nAddresses {
          '400713399927FE800000000000002B0D0FFFED0D7300401'H
      }
}
```

And the string representation is: `xldap://[fe80::2b0:d0ff:fed0:d730]:1025`

XLDAP address over SOAP using HTTP (IPv4)

AFI	IDI	DSP prefix	DSP Address
40	07133999	13	• Four-octet IP address, with three digits per octet. • Port number. • Final digit is hex 'F'.

For example, the XLDAP over SOAP address of the host 137.147.17.39 listening on port 1025 is:

```
address {
      nAddresses {'400713399913137147017039010 25F'H}
}
```

And its string representation is: `http://137.147.17.39:1025`

XLDAP address over TCP SOAP using HTTP (IPv6)

AFI	IDI	DSP prefix	DSP Address
40	07133999	25	• Sixteen-octet IP address, hex encoded. • Hex encoded port number.

For example, the XLDAP over SOAP address of the host fe80::2b0:d0ff:fed0:d730, listening on port 1025, is represented as:

```
address {
      nAddresses {
          '400713399925FE8000000000000002B0D0FFFED0D7300401'H
      }
}
```

And the string representation is: `http://[fe80::2b0:d0ff:fed0:d730]:1025`

ViewDS configuration file

The configuration file is a text file:

`${VFHOME}/setup/config`

where `${VFHOME}` is where ViewDS is installed.

The parameters in the configuration file can be modified through a text editor or the ViewDS Management Agent.

The parameters are grouped as follows:

- File system parameters
- Address parameters
- Access Presence configuration parameters
- Operational parameters

File system parameters

The configuration file includes path names that can be modified. There are usually more configurable path names than distinct path names because several are usually set to the same directory.

The ViewDS path names that can be configured are described below. A file name shown with an asterisk (for example, `alog.*`) refers to the set of file names generated if the Unix shell expands the `*` wildcard.

`admpasswd` This file stores the super-administrator password (see page 110).

Default: `${VFHOME}/general/deity`

`alogdir` This directory contains the system activity log (`alog.*`).

Default: `${VFHOME}/logs/ activity`

`capath` This file is used as the user's CA Certificate. The CA Certificate is required by the Stream DUA to construct a certificate path when performing a strong authentication bind.

No default value.

`catalog` This file allows localization of error messages.

Defaults:

- `${VFHOME}/lib/catalog` (Solaris or Linux)
- `${VFHOME}/lib/catalog.dll` (Windows)

`codetabs` This file contains the tables required to translate characters between various code pages and Unicode.

`database` This directory contains the DSA's database files (`ddm.*` and `p*.*`).

Default: `${VFHOME}/data`

`dbmpath` This directory contains the safe and scratch files.

Default: `${VFHOME}/data`

`dsacertificate` This file contains the DSA's certificate (corresponding to the private key defined for `dsaprivkey`).

Default: `${VFHOME}/setup/dsa.cer`

`dsaprivkey` This file contains the DSA's private key (corresponding to the public key in the DSA's certificate).

Default: `${VFHOME}/setup/dsa.pk8`

`dsaprivpass` This file contains the clear-text password required to decrypt the DSA's private key.

Default: `${VFHOME}/general/keyaccess`

`dsatrusted`	This directory contains certificates of identities that the DSA will grant super-administrator privileges to, if they have authenticated using strong authentication. The identities should include the RAS, which needs super-administrator privilege to shut down the DSA. Default: `${VFHOME}/setup/trusted`
`dumpdir`	This directory contains dump files (`dib.*`) generated when the database is dumped. The files are in Stream-DUA format. Default: `${VFHOME}/dump`
`errorlog`	This file contains the error log. Default: • `${VFHOME}/general/error` (Solaris or Linux) • `${VFHOME}/general/error.txt` (Windows)
`licensepath`	This file contains the ViewDS license key. Default: `${VFHOME}/setup/license.xml`
`printdir`	This directory contains print scripts used by Access Presence. Default: `${VFHOME}/print`
`qlogdir`	This directory contains the query log. Default: `${VFHOME}/logs`
`rascertificate`	This file contains the certificate for the RAS. The certificate corresponds to the private key in `rasprivkey`. Default: `${VFHOME}/setup/ras.cer`
`rasprivkey`	This file contains the private key for the RAS. The certificate corresponds to the public key in `rascertificate`. Default: `${VFHOME}/setup/ras.pk8`
`rasprivpass`	This file contains a clear-text password required to decrypt `rasprivkey`. Default: `${VFHOME}/general/ras.passwd`
`rastrusted`	This directory contains certificates of identities that can manage the RAS. The identities should include the DSA, which will attempt to bind to the RAS when it starts. Default: `${VFHOME}/setup/trusted`
`savedir`	This directory contains `ddm.*` and `p*.*` files after a directory has been saved. Default: `${VFHOME}/save`

`sumtabdir` This directory contains a log comprising a summary of the DSA's activity. The content is generated by `alogproc`.

Default: `${VFHOME}/sumtab`

`tmpdir` This is the directory where a DSA stores temporary files during replication. This directory does not need to be backed up.

Default: `${VFHOME}/tmp`

`ulogdir` This directory contains the update log file.

Default: `${VFHOME}/logs`

`webdir` This directory contains Access Presence configuration and data files.

Default: `${VFHOME}/webdir`

Address parameters

ViewDS processes communicate with each other using either an OSI Stack or the ViewDS TCP/IP-based Lightweight Stack (LWS). For both, there are further choices about the transport and network used. These choices are defined according to the address specification used for the addressing parameters (see page 23).

The address parameters as follows.

`baseport` The base port for TCP/IP-domain addresses.

The value of this parameter can be referenced by other parameters through the expression `{baseport}` with an optional integer offset. For example: `{baseport}+3`

Default: `3000`

`dsaaddress` The base address for the `dsa` process. The DSA always listens on this address and on `dsaaccesspoint` if defined.

Default: `vlws://localhost:{baseport}`

`dsacontrol` The address for communication between the DSA and RAS.

No default.

`dsaaccesspoint` The presentation address for a DSA using OSI communications. Recommended values for OSI over *RFC 1006* are:

- `osi://localhost:102` (under Unix root or Windows)
- `osi://localhost:{baseport}+3` (all others)

No default.

`dsaidmpaddress` The presentation address for the DSA when using IDM communications. Recommended value: `idm://localhost:{baseport}+8`

No default.

`dsaprivate` — The `dsa` process's private address used by the `dot` threads, and available to other local processes by explicit override. It should be a Unix domain address for Solaris and Linux.

Defaults:

- `vlws:${VFHOME}/general/x500local` (Solaris or Linux)
- `vlws://localhost:{baseport}+13` (Windows)

`dualocal` — The presentation address for Unix-based DUAs if using OSI communications.

Default for OSI over *RFC1006*: `osi://localhost`

`httpsaddress` — The DSA's XLDAP address. This address is required if you use XLDAP over TLS. No default.

`ldapaddress` — The DSA listens on this address for native LDAP connections. If the DSA SLDAP Address (`sldapaddress`) is also defined, the two should specify different port numbers. Recommended values:

- `ldap://localhost:389` (Unix root or Windows)
- `ldap://localhost:{baseport}+6` (all others)

No default.

`rasaddress` — The address for the RAS.

Default: `vlws://localhost{baseport}+18`

`soapaddress` — The HTTP address the DSA listens on for XLDAP requests over SOAP.

Default: `http://localhost:{baseport}+9`

`snmpcontrol` — The RAS communicates with the SNMP Agent on this address. No default.

`sldapaddress` — The DSA listens on this address for SSL LDAP connections. If the DSA LDAP Address (`ldapaddress`) is also defined, the two should specify different ports.

Recommended values:

- `sldap://localhost:636` (if running as Unix root or under Windows)
- `sldap://localhost:{baseport}+7` (all others)

No default.

`snmpagent` — The DSA listens on this address for SNMP requests. The value should take the form `host:port` – for example: `snmpagent = localhost:{baseport}+19`

No default.

`xldapaddress` The address the DSA listens on for XLDAP requests over TCP/IP. Recommended value: `xldap://localhost:{baseport}+12`

No default.

Usage

The addresses are used as follows:

- The `dsa` process listens on `dsaaccesspoint` (if available) and `dsaaddress`. When initiating associations to other DSAs it uses the knowledge reference to the other DSA to determine whether to use OSI or Lightweight Stack associations.
- The `sdua` process (Stream DUA), `dsac` process (DSA Controller), `pdua` process (Printing DUA) and Access Presence connect to `dsaaccesspoint` (if available), otherwise `dsaaddress`. If connecting through OSI, a client process uses `dualocal` as its OSI address.
- The `dsa` and `dot` threads communicate through `dsaprivate`.

The client processes (`dsac`, `sdua`, `pdua`) use an OSI stack if `dsaaccesspoint` is defined, otherwise they use `dsaaddress`. This is illustrated below.

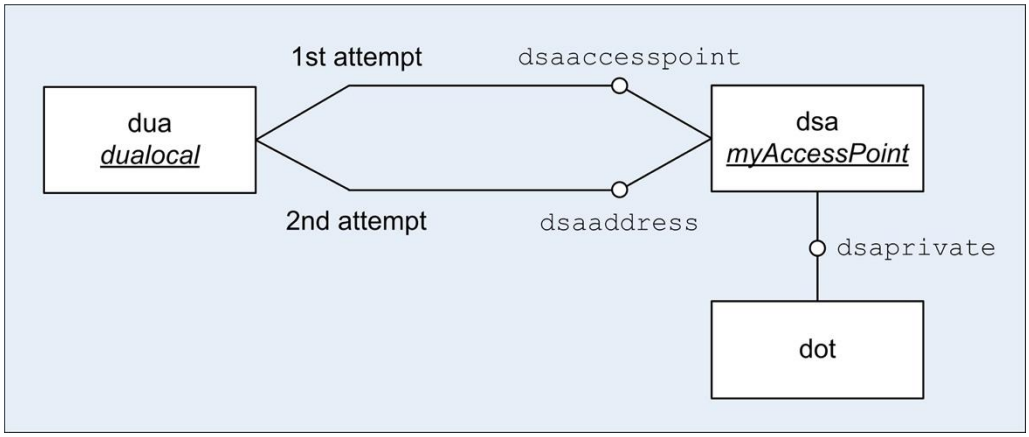


Figure 2: Addresses

The default or recommended port assignments are as follows:

Parameter	Port
dsaaddress	3000
dsaaccesspoint	3003 or 102
dsaidmpaddress	3008
soapaddress	3009
xldapaddress	3012
dsaprivate	3013 (Windows) `general/x500local` (Unix/Solaris)
snmpagent	3019
ldapaddress	3006 or 389

Access Presence configuration parameters

These configuration parameters are only relevant when Access Presence is implemented. They are described in the *Technical Reference Guide: User Interfaces*.

Operational parameters

`baseentry` The Distinguished Name (DN) of the subschema administrative point in Stream DUA notation. For example:

```
baseentry = { O "Deltawing" }
```

If Access Presence is used, this parameter must have a value so that Access Presence can retrieve schema information. Access Presence also uses this parameter as the default base entry for searches.

No default.

`bounds` Whether strict bounds checking on the length of X.520 attributes of string syntax is enabled (see *Parsing rules* for attributeSyntax on page 66).

Note that disabling bounds checking may cause protocol errors when results are chained through, or chained to, systems that enforce bounds checking. Most implementations, however, do not enforce bounds checking.

Default: `off`

`certificatelookup` Determines the way in which the DSA looks up certificates during strong authentication (see page 112). There are two possible values:

- `subjectNameIsEntryName` –the DSA expects a certificate's subject name to match the DN of the entry in which it is stored.
- `mapSubjectNameToEntryName` – the DSA searches the entire DIT for a certificate with a matching subject name. It then uses the entry containing the matching certificate, in the case of the end-entity certificate, as the authentication identity for subsequent processing of operations on the authenticated connection.

The second option has two advantages:

- a certificate naming policy can be used that is different to the naming policy of the directory.
- an entry can be moved in the DIT (therefore changing its DN) without the requirement to re-sign its certificate.

Default: `subjectNameIsEntryName`

`certrevocation` Whether certificate revocation checking is enabled during strong authentication.

Certificate path processing is carried out by requiring every certificate used during certificate verification to be present in the directory. User certificates should be stored in the `userCertificate` attribute of the appropriate entry and intermediate CA certificates should be stored in the `cACertificate` attribute of the appropriate entry.

When `certrevocation` is set to on, every certificate that is verified during the certificate path validation will have its revocation status checked if a certificate revocation list (CRL) is available. If a certificate is found to be revoked, then certificate path processing will fail. The CRL of an intermediate CA should be stored in the `certificateRevocationList` attribute within the same entry used to store that authority's `cACertificate`. The CRL of a trust anchor should be present within the `certificateRevocationList` attribute in the root entry.

Default: `on`

`deitymask`

The Unix umask to use when the DSA creates the deity file. This may be adjusted to allow all members of a Unix group to have administrative privileges. It allows an HTTP server to run under a different account and still have the Admin DUA (no longer shipped with ViewDS) behave correctly.

Default: `0400`

`displog`

Whether a consumer DSA in a replication agreement should log the binary DISP PDUs it receives from its supplier. Some of these PDUs can be quite large, so this option should be used with care. The PDUs are logged in the `tmpdir` directory.

Default: `off`

`dsigcanonicaltoleration`

Identifies the extent to which ViewDS conforms to the Oasis specification for canonicalization of XML digital signatures. If set to Strict, then ViewDS validates a digital signature according to the specification. Alternatively, if set to Tolerant, ViewDS removes comments from an invalid signature and attempts to re-verify it.

Default: `Tolerant`

`dsigrequirestrong`

Identifies whether ViewDS requires strong authentication prior to processing XACML requests. If set to On, then ViewDS will only process XACML requests that have been sent by a client whose identity has been verified using strong authentication.

Supported strong authentication mechanisms for XACML requests include:

- An XACML request sent over SSL using SSL client authentication
- A signed SAML XACML request.

Any XACML requests sent by a client whose identity has not been verified by strong authentication will be rejected with an HTTP unauthorized response.

If set to Off, then all XACML requests will be processed normally, in conjunction with any other authorization policies that may apply.

Default: `Off`

`dsigsignresponse`	Identifies whether ViewDS should sign SAML XACML responses. If set to Off, then ViewDS will only sign the responses of signed requests. If set to On, then ViewDS will sign all responses. Default: `Off`
`dsigx509data`	Identifies whether to attach the X509Certificate or the X509SubjectName to signed responses. Recipients of a signed response must be able to identify the signing certificate in order to verify the signature. ViewDS can provide clients with the signing certificate directly or provide the subject DN of the signing certificate. If set to 'X509Certificate', then the signing certificate will be attached into the signed response. If set to 'X509SubjectName', then the distinguished name of the signing certificate will be attached into the signed response. Default: `X509Certificate`

`gssService`	The `<service name>` component of the Kerberos principal name.
	
	The Kerberos principal name for a service generally takes the following form:
	
	`<service name>/<host name>@<realm name>`
	
	On a Unix host it is abbreviated to:
	
	`<service name>@<host name>`
	
	Default: `ldap`

`gssName`	The `<host name>` component of the Kerberos principle name. If this parameter is unspecified, ViewDS obtains the fully qualified domain name of the local host. No default
`gssRealm`	The `<realm name>` component of the Kerberos principle name. The `<realm name>` is usually obtained from the operating system. Under Windows, it can also be obtained from the domain name of the local host. This parameter should not normally be required. However, it should be declared when: • the DSA is running under Windows and authenticates into a Unix Kerberos environment; and • the `<realm name>` cannot be determined from the Unix domain name.

`gssUserName` The attribute that identifies the *authentication identity* in the directory.

After a SASL GSS-API *authentication identity* has been established, it must be mapped to an identity in the directory to be used for authorization decisions.

ViewDS performs this mapping by:

- Searching the directory for the attribute identified by `gssUserName`.
- If ViewDS finds a single matching entry, the DN of the entry is used as the authorization identity. Otherwise, it checks the `anonymousPrivilege` attribute (see page 127) in the root entry.
- If the `credentialType` in the `anonymousPrivilege` attribute is set to `saslGSSAPI`, the authorization identity is set to anonymous with the ViewDS access-control privilege identified in the matching `anonymousPrivilege` value. Otherwise, authentication failure is reported with the error `inappropriateAuthentication`.

Default: `viewDSUserName`

`ignorebackslash` Set to `on` for LDAP to prevent the backslash character "\" from being treated as an escape character in strings.

Default: `off`

`incrementsize` The maximum number of DAP/LDAP updates included in each incremental update sent by a supplier in a replication agreement. This setting prevents the PDUs from becoming too large to be handled in a single update transaction.

Default: `256`

`ldapasciihex` The parameter is provided for backwards compatibility with previous versions of ViewDS (where it is always `on`). It is only used for LDAPv2 connections. If `on`, binary values in LDAPv2 are returned using the ASCII hex notation "`{ASN}`...". If `off`, values without a text encoding are returned as raw ASN.1 binary values.

Default: `off`

`ldapv2syntax` The syntax expected and returned on LDAPv2 connections. It is set to either `t61`, `8859` or `utf8`.

Default: `t61` is standard (other values may be needed to inter-work with certain third-party clients).

`pselectorhack` This option is for inter-working with other X.500 products. It prevents ViewDS from distinguishing between a presentation selector in a presentation address being absent or having zero length.

Default: `on`

`rastimeout` Defines how many seconds the RAS command line will wait for a response from the `rasrv` process before timing out. You can override this setting from the RAS command line (see *Remote Administration Service* on page 11).

A value of 0 declares an indefinite time limit.

Default: 10

`rxeropt` Controls the level of optimization used when encoding data using RXER. It can be set to:

- `off` – results in well-presented RXER encoding with clear indentation and textual representation of OIDs.
- `partial` – also makes the RXER well formatted, but omits the textual OID represents.
- `full` – optimizes the RXER further by removing all indentation.

Default: `off`

`saslrealm` The SASL LDAP authentication mechanisms require a realm field. This should be an ASCII value.

Default: `viewDS`

`saslusername` The SASL LDAP authentication mechanisms permit authentication credentials to be specified using a simple user name instead of a DN. However, ViewDS must map the user name onto a DN during authorization checking.

The value of this parameter identifies the attribute type to be searched using an assertion value of the user name provided in the authentication request. Where a unique match for this value of this attribute is found, the matching entry's DN will be used for authorization checks.

The default attribute used for mapping the user name to a DN is the `viewDSuserName` attribute.

`schemachecking` Controls the schema-checking level for the ViewDS Fast Load utility (see page 11). The options are:

- `none`
- `all`
- `ignoreUserModifiableFlag`

For more information about schema checking, see 60.

If upgrading to ViewDS version 7.1, the recommended setting is `none`.

Default: `none`

`shmmax` This parameter has been deprecated.

`snmpapplindex` The SNMP application index for the DSA. Default: `1`

`snmpmaxpdusize` The maximum size of an SNMP PDU the DSA will send.

Default: `484`

`strictldap`	The DSA tolerates non-standard variations in the LDAP protocol received from LDAP clients. When this parameter is `on`, the DSA requires strict LDAP conformance by clients. Default: `off`
`truncateresult`	If set to `off`, the number of search results will match the `sizelimit`. If set to `on`, a group of search results with the same ranking will be discarded and the number of results returned will be less than the `sizelimit` (see page 22). Default: `off`
`zerolengthstring`	If `on`, this field removes the requirement that DirectoryString values have a length greater than zero. This allows zero-length DirectoryString values to be added to the directory. Default: `off`

Chapter 4
Defining schema

This chapter describes schema concepts and the operational attributes that define schema. This chapter describes how to modify the operational attributes using the *Stream Directory User Agent* (Stream DUA). However, they can also be modified through the ViewDS Management Agent.

This chapter includes:

- Concepts
- Schema checking
- Operational attributes
- Other operational attributes

Concepts

This section provides an overview of different components of schema.

Subschema area, administrative point and subentry

A *subschema area* is a subtree in a Directory Information Tree (DIT) where a specific schema applies. The entry at the top of the subtree is called the *subschema administrative point*. It holds a special operational attribute, `administrativeRole`, which has the value `subschemaAdminSpecificArea`.

NOTE: The terms *subschema* and *schema* are synonyms.

A schema is defined by operational attributes in two subentries below the administrative point:

- schema configuration subentry
- subschema subentry

The *schema configuration subentry* defines indexes and attribute-type extensions; and the *subschema subentry* defines all other aspects of a schema (including word lists – synonyms, noise words and truncated words). Neither subentry is usually displayed in the DIT. Figure 3 on the next page shows a subschema administrative point at the top of a subschema area, and the two subentries.

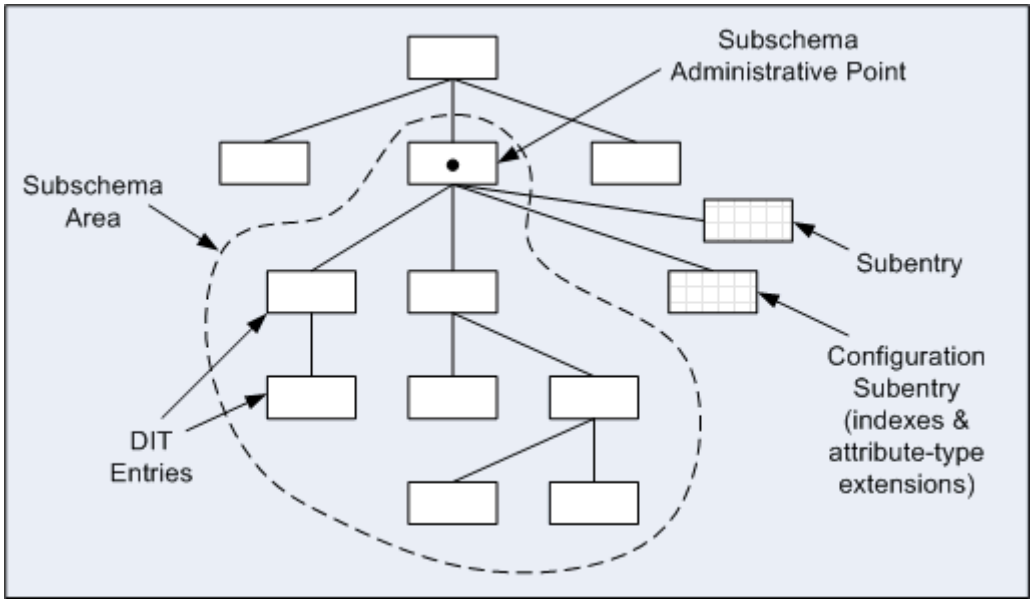


Figure 3: Subschema administrative point and area

It is worth noting that the following are standard X.500 objects: the `subschemaAdminSpecificArea` value of `administrativeRole`, and the subschema subentry. While the following are ViewDS-specific: the `schemaConfigurationArea` value of `administrativeRole`, and the schema configuration subentry.

Built-in and predefined schema

The terms *built-in schema* and *predefined schema* are both used in this guide. Built-in schema refers to the schema compiled into the ViewDS libraries; and predefined schema refers to schema definitions in text files provided with ViewDS. Predefined schema can be imported using the ViewDS Management Agent.

Schema operational attributes

A schema is set up by creating a subschema administrative point, and then defining the following schema operational attributes in the subschema subentry:

- `attributeTypes`
- `objectClasses`
- `matchingRules`
- `matchingRuleUse`
- `nameForms`
- `dITStructureRules`
- `dITContentRules`
- `definitions`

These operational attributes can be modified through the Stream DUA or ViewDS Management Agent and read by other DUAs (such as, the ViewDS Access Presence). The operational attributes are introduced below and then described in detail later in this chapter (see *Operational attributes* on page 62).

Object identifiers

An object identifier (or OID) is a sequence of integers that identifies an element of schema – such as an object class, attribute type or matching rule.

An OID describes a tree, each node in the tree represents an object and a naming authority responsible for allocating the next level of the tree. The object-identifier tree is distinct from the DIT. The object-identifier tree identifies such things like attributes and object classes; and the DIT identifies real-world entities such as countries, organizations and people.

Example object identifier

The OID that represents the attribute type `streetAddress` is `{2 5 4 9}`. It is constructed as follows:

- By international agreement, the object identifier `{2}` has been allocated to joint ISO-ITU administration.
- This administration has allocated the arc 5 to directory, which gives the OID `{2 5}`, and has given administration of this OID to the X.500 committee.
- The X.500 committee has allocated the arc `4` to attributes, which gives the OID `{2 5 4}`.
- The committee has assigned the arc `9` to the attribute `streetAddress`, which gives the OID `{2 5 4 9}`.

Object identifier prefixes

Typically, the object identifiers used in ViewDS have one of the following six prefixes, plus any other prefixes introduced by the ViewDS system administrator.

Prefix	Definition
ds	2 5
mhs	2 6
vf	1 3 32 0 1
Ads	1 2 36 79672281 1
Xed	1 3 6 1 4 1 21472
Vds	1 3 6 1 4 1 21473 5

Creating object identifiers

To be allocated an object identifier, apply to one of the international standards bodies (ITU or ISO) or to your national standards body (for example, Standards Australia). The allocated object identifier can then be used as the root for new object identifiers when you create new schema elements.

Every Australian organization has a default object identifier, which includes their Australian Business Number (ABN):

`{1 2 36 abn}`

Where *abn* is the ABN.

Alternatively, your ViewDS vendor can assign object identifiers from its own arc if you do not have (or wish to use) your own. Contact your ViewDS vendor for further details.

Object identifier organization

The following table shows the recommended organization for object identifiers. It is parallel to the organization used by X.500, which involves using the arc below your organization's arc to represent the category-of-information object (see *X.501 Annex A* for a full list).

Category	Arc
attribute type	4
object class	6
DSA operational attribute *	12
matching rule *	13
knowledge matching rule *	14
name form	15
operational attribute *	18
schema operational attribute *	21
access control attribute *	24

NOTE: The operational attributes and matching rules marked with an asterisk * are shown for information only. Operational attributes and matching rules are not user-definable.

Managing object identifier arcs

ViewDS allows you to declare object identifier arcs for user-defined matching rules, attributes, object classes and name forms. The arcs are stored in an operational attribute in a DSA subschema subentry, which can be accessed through the Stream DUA or ViewDS Management Agent.

The operational attribute is as follows:

```
viewDSSubschemaObjectIdentifiers ATTRIBUTE ::= {
      WITH SYNTAX    ObjectIdentifierArcs
      SINGLE VALUE   TRUE
      USAGE          directoryOperation
      ID             id-viewDS-soa-objectIdentifiers
}

ObjectIdentifierArcs ::= SEQUENCE {
      matchingRules  [0] OBJECT IDENTIFIER OPTIONAL,
      attributeTypes [1] OBJECT IDENTIFIER OPTIONAL,
      objectClasses  [2] OBJECT IDENTIFIER OPTIONAL,
      nameForms      [3] OBJECT IDENTIFIER OPTIONAL
}
```

For information about declaring arcs through ViewDS Management Agent, see the help topic *View or modify OID arcs*.

After arcs have been declared, the ViewDS Management Agent will present the next available identifier whenever a user creates a new schema object.

Attribute syntaxes

An *attribute syntax* is the data type used to represent values of an attribute. Every attribute type has a single attribute syntax, which is assigned to it in the attribute definition. Attribute syntaxes are also the basic building blocks from which a new attribute type is built.

ViewDS supports a fixed set of built-in attribute syntaxes, referenced by their ASN.1 type name. ViewDS supports:

- all ASN.1 built-in types
- all user and operational attribute syntaxes defined in X.500 (1993)
- four attribute syntaxes from X.400 (1994) which are required to support X.400 directory attributes
- 11 QUIPU syntaxes
- a number of ViewDS-specific attribute syntaxes

ViewDS also allows references to the built-in types of XML schema, and references to types in imported XML schema documents.

Examples of built-in attribute syntaxes are: BOOLEAN, INTEGER, DirectoryString, ORAddress, PresentationAddress, Privilege.

Matching rules

The second building block in the definition of an attribute type is the set of *matching rules* used by the Directory System Agent (DSA) to match a value of the attribute. Matching rules are invoked automatically whenever an equality, ordering or substring match is attempted on an attribute value. They can also be invoked explicitly in a search filter using a *matching rule assertion*.

In ViewDS, a matching rule is referenced by its name or by its object identifier. Every rule has a name and an object identifier in its definition.

ViewDS supports a fixed set of matching rules – those defined in X.500 (1993) and LDAP, plus a number of X.400 and ViewDS-specific matching rules. *Appendix B* on page 225 lists the set of supported matching rules along with their names, object identifiers and references to definitions.

Assertion syntax

Each matching rule has an *assertion syntax*. It defines the syntax of the information provided in the search-filter item when searching with the matching rule. The assertion syntax is a property of a matching rule similar to the syntax of an attribute.

Examples

An example of why matching rules have their own assertion syntaxes is the substrings match. The assertion syntax needs to be sufficiently complex to express the sequence of substrings required to match against strings and their order. So, the `SubstringAssertion` syntax of a substrings matching rule is as follows:

```
SubstringAssertion ::= SEQUENCE OF CHOICE {
          initial [0] UnboundedDirectoryString,
          any         [1] UnboundedDirectoryString,
          final       [2] UnboundedDirectoryString,
          control Attribute
      - at most one initial and one final component}
```

Another example is the matching rule `generalizedTimeMatch`, which has the following properties:

- object identifier `{2 5 13 27}`
- assertion syntax of `GeneralizedTime` (an ASN.1 built-in type)
- returns TRUE if the attribute value it is applied to represents the same time as the presented value

Using matching rules

To be available in a subschema area, a matching rule must be declared in the operational attribute `matchingRules` in the subschema subentry.

In general, a matching rule's assertion syntax does not allow a DUA to determine whether the matching rule can be used with a particular attribute. For this reason, the subschema subentry may include the operational attribute `matchingRuleUse` to define which attribute types use which matching rules. This is only of benefit to DUAs – the presence or absence of a `matchingRuleUse` attribute has no effect on the behaviour of the DSA.

Attributes

A directory *attribute* is a basic item of information about an entry, consisting of a *type* (or kind), a label and one or more *values* of that type. There are two kinds of directory attributes:

- **user attributes**, which hold information of interest to ViewDS users;
- **operational attributes**, which hold information required for the operation of the DSA or DUA.

The user and operational attributes in ViewDS comprise:

- standard attributes defined in the X.500 Recommendations and LDAP RFCs;
- ViewDS-specific attributes; and
- user-defined attributes.

ViewDS has built-in knowledge of all X.400 (1994) and X.500 (1993) attributes. The built-in attribute types are listed in *Appendix B*.

Using attributes

To use a particular user attribute in a schema requires the following:

- The attribute type must be declared in the operational attribute `attributeTypes` in the subschema subentry.
- If the attribute requires special treatment by the DSA (such as indexes, approximate matching, DN tracking) then an appropriate value must be declared in `entryIndexing` or `attributeTypeExtensions` in the schema configuration subentry.
- If Access Presence needs to display the attribute, then appropriate values must be declared for the operational attribute `attributePresentation` in the subschema subentry.

Matching rules

Optionally, an attribute definition can declare which matching rules should be used when performing an equality, ordering or substring match on the attribute values.

Equality matching rule

The DSA uses an equality matching rule when:

- adding a value to a multi-valued attribute to determine whether the value already exists
- deleting a value from a multi-valued attribute to identify the value to be deleted
- comparing values using the compare operation

If no equality matching rule is defined (or available by default) the DSA cannot perform the ‘add values’ or ‘remove values’ functions of the operations Modify Entry or Compare on the attribute.

Ordering matching rule

An attribute’s ordering matching rule defines its default ordering semantics. This default ordering is used when processing a greaterOrEqual or lessOrEqual item in a search filter.

If no ordering rule is specified for a particular kind of match, the DSA will be unable to perform matches of that kind.

Substrings matching rule

Similarly, an attribute’s substrings matching rule defines the default substring matching semantics, which is used when processing a substrings item in a search filter.

Object classes

An *object class* describes the kind of object an entry represents – for example, a person, an organizational unit or a distribution list.

An object class can be derived from other object classes using multiple inheritance.

The object class of each entry is stored within the entry itself in the multi-valued `objectClass` attribute. Each value is an object identifier – that of the object class or one of its superclasses in the inheritance chain. All superclasses must appear in the `objectClass` attribute, with the exception of `top`, which is the object class from which all others are ultimately derived.

NOTE: The `objectClass` attribute is classified as a user attribute in X.500 for historical reasons, even though in most respects it acts as an operational attribute.

Structural and auxiliary object classes

An object class is either *abstract*, *structural* or *auxiliary*:

- An abstract object class can only be used as a superclass of another object class. An example of an abstract object class is `top`.
- A structural object class typically defines the kind of entry – for example, organization, person, or device.
- An auxiliary object class constitutes optional, temporary characteristics – for example, message-handling user, or strong-security user.

An entry has exactly one structural object class, but may have any number of auxiliary object classes associated with it.

The value of the `objectClass` attribute at the bottom of the structural-object-class chain is called the *structural object class* of the entry. It determines the *name form* and

set of possible structure rules for the entry, and has a major role in determining the attribute types the entry can hold. The structural object class can be read from the entry as the value of the single-valued operational attribute `structuralObjectClass`. This value is generated by the DSA when the entry is created and is not user-modifiable.

Object classes are added to a schema through the operational attribute `objectClasses` in the subschema subentry. If built-in object classes are used in a schema, they must also be declared in this operational attribute.

NOTE: If Access Presence needs to display entries of a particular object class, they must be declared in the subschema subentry's `objectClassPresentation` operational attribute (see the *Technical Reference Guide: User Interfaces*).

Aliases

An alias is an entry whose `aliasedEntryName` attribute contains the Distinguished Name of another entry, and whose object class is a subclass of `alias`. The DSA normally resolves aliases automatically.

Although the `alias` structural object class makes the `aliasedEntryName` attribute mandatory, this is insufficient for naming an alias entry. To permit the various naming attributes to be present requires either subclasses of the `alias` class to be defined or an all-encompassing content rule to be defined.

Name forms

A *name form* defines which attributes can appear in the Relative Distinguished Name (RDN) of an entry.

For each structural object class, there is one or more *name form*. Each includes a list of the mandatory and optional attributes that can appear in the RDN of an entry of a particular structural object class.

The name forms for a particular structural object class must be such that any particular combination of naming attributes is permitted by only one name form. This is not a requirement of X.500, but is imposed by ViewDS to avoid ambiguity about which name form should apply.

Using name forms

A name form is identified by its object identifier.

Name forms are added to a schema through the operational attribute `nameForms` in the subschema subentry. If the built-in name forms are used in the schema, they must be declared explicitly, just as for user-defined name forms.

NOTE: If the `nameForms` and `dITStructureRules` operational attributes are both absent from the subschema subentry, the DSA will not restrict which attributes can appear in an entry's RDN.

Structure rules

A *structure rule* defines which class of entry can be superior or subordinate to another class of entry in the DIT.

Each entry has a *governing structure rule*, which the DSA selects from one of the structure rules assigned to the entry's structural object class.

The governing structure rule of an entry is determined by three factors:

- the structural object class of the entry;
- the name form of the entry, which is determined by the entry's structural object class and the attributes used to form its RDN; and
- the governing structure rule of the entry's immediate superior in the DIT.

The governing structure rule of an entry is *calculated* by the DSA when the entry is added or moved. It can be read from the entry as the value of the single-valued operational attribute `governingStructureRule`.

For each name form, the schema lists one or more permitted structure rules. Each lists the governing structure rules of permitted superiors in the DIT. These constrain the ability to add an arbitrary entry below some other entry. If no governing structure rule can be calculated for a proposed new entry, it cannot be added.

The subschema administrative point entry does not depend on the governing structure rule of its superior entry. Its structure rule must have an empty list of superior governing structure rules. If no such structure rule exists, the subschema administrative point entry is permitted to exist, but it has no governing structure rule and no subordinates can be added to it (although subentries may be).

Using structure rules

A structure rule is identified by an integer, which is arbitrary but must be unique within the subschema area. The structure rules for a particular name form must be such that any particular *superior* governing structure rule appears in at most one structure rule.

Structure rules are made known to the schema through the operational attribute `dITStructureRules` in the subschema entry. There are no built-in structure rules.

If the `nameForms` and `dITStructureRules` operational attributes are both absent from the subschema subentry, then the DSA will not restrict which class of entry can be subordinate to another class of entry in the DIT.

Content rules

Each object class (structural or auxiliary) defines a set of mandatory and optional attributes for its entries – *mandatory* attributes must have values, and *optional* attributes may or may not have values. Other attributes are not allowed in an entry.

To augment this, a *content rule* can be defined for a structural object class. A content rule can specify additional mandatory and optional attributes, and can disallow otherwise permitted optional attributes.

A content rule can also associate a set of auxiliary object classes with a structural object class. The specified auxiliary object classes can then be added to an entry of the structural object class.

Using content rules

Content rules are identified by the structural object class they augment. A separate identifier is not needed.

Content rules are made available through the operational attribute `dITContentRules` in the subschema subentry. There are no built-in content rules.

Schema checking

The Stream DUA and ViewDS Fast Load (vfload) tools allow you to load data from a file into a database. When either tool is used, the DSA checks the data against the relevant schema.

This subsection describes the following aspects of schema checking:

- Loading data files
- Setting levels of schema checking
- Checks performed by the DSA

Loading data files

When loading data from a ViewDS dump file, the DSA obtains a level of schema checking. The level can be declared in the dump file:

- individually for each operation in the file
- globally for all operations in the file that do not have an individual declaration

For an operation without an individual declaration – in a file without a global declaration – the DSA applies the level set for the tool being used to load the database. (This tool is either Stream DUA or ViewDS Fast Load.)

When loading LDIF content records, Stream DUA and ViewDS Fast Load automatically set the level of schema checking to `ignoreUserModifiableFlag`. For LDAP and XLDAP, a `schemaCheckingRequest` control sets the level of schema checking.

Setting levels of schema checking

The levels for schema checking are `ignoreUserModifiableFlag`, `all` or `none`.

The level can be set for Stream DUA and ViewDS Fast Load:

Stream DUA	The schema-checking behaviour for Stream DUA is set through the `set options` command (see page 196). The default is `all`.
ViewDS Fast Load (vfload)	The default schema-checking behaviour for ViewDS Fast Load is set by the `schemachecking` parameter in the ViewDS configuration file (see page 48). The default is `none`. The `schemachecking` configuration file option did not exist before 6.0e8 but the schema-checking behaviour did. It was, however, tied to the `manageDSAIT` service control option. When ViewDS version 7.1 is installed, the standard configuration file is installed with the `schemachecking` parameter set to `all`. Upgrading from View500 version 6 involves copy the existing configuration file to the ViewDS version 7.1 directory structure. In this case, the `schemachecking` parameter defaults to `none`. The `schemachecking` parameter can be overridden on a per-operation basis using the `schemaChecking` service control option in the common arguments.

ignoreUserModifiableFlag

Reloading from database-dump files restores the content of a directory to its state when the dump command was executed. This includes setting values of operational attributes (such as `modifyTimestamp`) that cannot normally be modified by user operations. The `ignoreUserModifiableFlag` schema-checking mode allows these operational attributes to be loaded, but all other schema violations in the dump files to generate errors.

To illustrate, consider the following scenario:

1. A schema is modified. (This may introduce inconsistencies between the existing data and the schema.)
2. The database is dumped.
3. The `schemaChecking` commands are removed from the dump file, and the level of schema checking for the ViewDS Fast Load tool is set to `ignoreUserModifiableFlag`.
4. The database is emptied and the dump file is loaded using ViewDS Fast Load. At this stage, the DSA reports an error whenever it finds an inconsistency between the data and the schema. However, the schema-checking level `ignoreUserModifiableFlag` ensures that no error is reported when the value of an operational attribute is loaded into the database.

Checks performed by the DSA

The DSA performs the following checks:

- Entry creation
- Entry modification
- Schema inconsistency

Entry creation

When an entry is created it belongs in the subschema area of its superior, unless it is created as an administrative point.

When an entry is created, the DSA checks the following:

- Entries immediately below the root must be created as autonomous administrative point entries.
- Every entry must be created with an `objectClass` attribute and contains exactly one structural object class chain and zero or more auxiliary object class chains. If an object class value is present then an object class value for its superclass(es) must also be present. The DSA determines the structural object class of the entry and creates the single-valued operational attribute `structuralObjectClass`.
- The structural object class of the entry and the attribute types used in its RDN combine to identify one of the DSA's supported name forms (the 'actual' name form of the entry). Restrictions on the definition of name forms ensure that no more than one name form can be identified. If none are identified, the entry cannot be added to the directory.
- The 'actual' name form and the governing structure rule of the entry's intended immediate superior combine to identify one of the DSA's supported structure rules. This is the entry's own governing structure rule. Restrictions on the definition of structure rules ensure that no more than one structure rule can be identified. If none

are identified, the entry cannot be added to the directory. (The DSA stores the entry's governing structure rule in the single-valued operational attribute `governingStructureRule`.)

- If there is a content rule for the entry's structural object class, the DSA checks that any auxiliary object classes declared in the `objectClass` attribute are permitted by the content rule. Otherwise, the entry cannot be added to the directory.
- The DSA performs an *entry content check*. The DSA checks that the entry's content conforms to the mandatory and optional attributes defined by its structural and auxiliary object classes (and also possibly a content rule). If it does not, the entry cannot be added to the directory.

Entry modification

When an entry is modified, renamed or moved, the DSA performs the following checks:

- If attribute values are modified, the DSA performs an *entry content check* (see the last bullet in the previous subsection). It rejects the modification if the check fails.
- If the object class attribute is modified, the DSA checks that the structural object class is unchanged (that is, that only auxiliary object classes are modified). If not, it rejects the modification.
- If the entry is renamed or moved, the DSA determines a new 'actual' name form and governing structure rule, and performs an entry content check. If no governing structure rule can be found – that is, the renamed or moved entry is not acceptable as a subordinate of its new superior – or the entry content check fails and the DSA rejects the operation.

Schema inconsistency

As renaming or moving an entry may result in its governing structure rule changing, it is possible for subordinates of an entry to become inconsistent with their schema. The DSA permits this, in accordance with X.500. The directory continues to operate normally and can be dumped and reloaded successfully. However, there are considerations:

- When reloading, the schema-checking level should be set appropriately (see *Schema checking* on page 60).
- DUAs and other DSAs that interwork with the DSA must be prepared to handle schema inconsistencies.

Schema may also become inconsistent as a result of changes to the schema publication attributes. The data in the directory may no longer conform to the modified schema. Schema inconsistencies should be rectified manually (ViewDS does not provide support for this).

Operational attributes

If the same schema object is present in multiple subschema areas (or it also appears as a built-in definition) the definitions must be consistent. The exceptions are the `name`, `description` and `obsolete` fields.

The `name` field contains a set of symbolic names for a schema definition. Each `name` must begin with a letter and may contain any number of letters (case is insignificant), digits and hyphens.

The names are used by Stream DUA when listing the contents of an entry. They are used in place of the schema definition's object identifier. If multiple names are given, Stream DUA uses the first name when displaying (outputting) entry information, but recognizes all names when loading entry information.

The DSA also uses the first name as the schema definition's descriptor in LDAP responses. That is, unless one of the names is prefixed with 'ldap:' – in which case, that name will be used in LDAP. The DSA recognizes all the names, both the prefixed ones and the ones without a prefix, in LDAP requests.

In the following descriptions of operational attributes, a 'previously defined' symbolic name refers to either:

- a name specified in a schema object that already exists in the schema; *or*
- a name being added in the same operation as the current one.

administrativeRole

This operational attribute allows you to create a subschema administrative point at an entry. To create a subschema administrative point, `administrativeRole` must have the value `subschemaAdminSpecificArea`.

Schema attributes that are not in a subschema subentry subordinate to a subschema administrative point are not used by ViewDS to enforce schema rules (though such attributes are checked for consistency with other schema definitions).

The ASN.1 definition is:

```
administrativeRole   ATTRIBUTE  ::= {
      WITH SYNTAX           OBJECT IDENTIFIER
      EQUALITY MATCHING RULEobjectIdentifierMatch
      USAGE                     directoryOperation
      ID                                {ds 18 5}  }
```

attributeTypes

This operational attribute is added to a subschema subentry and defines the user attributes in the subschema area. It is multi-valued, and each value defines one attribute type. All built-in and user-defined attribute types are defined in this way.

The ASN.1 definition is:

```
attributeTypes   ATTRIBUTE  ::= {
      WITH SYNTAX               AttributeTypeDescription
      EQUALITY MATCHING RULE    objectIdentifierFirstComponentMatch
      USAGE                     directoryOperation
      ID                                {ds 21 5} }

AttributeTypeDescription    ::= SEQUENCE {
      identifier     ATTRIBUTE.&id,
      name           SET OF DirectoryString {ub-schema} OPTIONAL,
      description    DirectoryString {ub-schema} OPTIONAL,
      obsolete       BOOLEAN DEFAULT FALSE,
      information    [0]  AttributeTypeInformation }

AttributeTypeInformation    ::= SEQUENCE {
      derivation         [0] ATTRIBUTE.&id OPTIONAL,
      equalityMatch      [1] MATCHING-RULE.&id OPTIONAL,
      orderingMatch      [2] MATCHING-RULE.&id OPTIONAL,
```

```
      substringsMatch    [3] MATCHING-RULE.&id OPTIONAL,
      attributeSyntax    [4] DirectoryString {ub-schema} OPTIONAL,
      multi-valued       [5] BOOLEAN DEFAULT TRUE,
      collective         [6] BOOLEAN DEFAULT FALSE,
      userModifiable     [7] BOOLEAN DEFAULT TRUE,
      application        AttributeUsage  DEFAULT
                                userApplications }
AttributeUsage       ::= ENUMERATED {
      userApplications      (0),
      directoryOperation (1),
      distributedOperation(2),
      dSAOperation             (3) }
```

The components are described below.

AttributeTypeDescription

`identifier` — The object identifier of the attribute. It is either a built-in symbolic name or a numeric object identifier, for example, `commonName` or `{1 3 32 0 2 0 4 42}`.

`name` — The set of symbolic names to be created for the attribute.

`description` — A natural language description of the attribute. ViewDS does not use this component, but it can be used when publishing schema for third-party DUAs.

`obsolete` — A flag that indicates that the attribute is obsolete. Existing entries may contain obsolete attributes. However, an obsolete attribute cannot be added to an existing entry or new entry.

`information` — This component comprises the subcomponents described below.

AttributeTypeInformation

`derivation` — The attribute type (if any) of which the attribute is a subtype.

It is only declared if the attribute is a subtype of some other attribute. In this case, by default, the attribute acquires the `equalityMatch`, `orderingMatch`, `substringsMatch` and `attributeSyntax` of the supertype, although these can be declared explicitly.

`equalityMatch` `orderingMatch` `substringsMatch` — The equality, ordering, and substrings matching rules for the attribute. Each is either the built-in symbolic name or the object identifier of one of the built-in matching rules (see *Appendix B*). If a `derivation` is declared, these values are only required if they differ from those of the super-type.

If an attribute is defined without an equality matching rule, the DSA uses special rules for handling it. An attempt to add or remove values of such an attribute return an `inappropriateMatching` attribute error (adding or removing

	the whole attribute is permitted). Compare and search operations always evaluate attribute value assertions containing the attribute to false. Built-in attribute types with no equality matching rule can be *redefined* to use `an` equality matching rule applicable to the attribute's syntax. In this case, an attribute's values can be compared, added or removed. A redefinition of the built-in descriptions is allowed if all the instances of the `AttributeTypeDescription` for the attribute, whatever subentries they appear in, define the same equality matching rule. (The `AttributeTypeDescriptions` are allowed to differ from the built-in description in this one way but they cannot differ from one another.)
`attributeSyntax`	The attribute syntax of the attribute. This is a string equal to either the ASN.1 type name of the syntax, normally chosen from the set of supported attribute syntaxes in *Appendix B*, or the text of an ASN.X <type> element) as described in *RFC 4912*. ViewDS only supports the ASN.X notation for a type reference (a <type> element with a `ref` XML attribute). Additionally, only references to built-in X.500 ASN.1 types and XML Schema types imported through the `definitions` attribute (see page 78) are supported. If a `derivation` is given, an `attributeSyntax` need only be given if it differs from that of the supertype (in which case it must nonetheless be a compatible ASN.1 subtype or restriction-derived XML Schema type). If the attribute syntax is the string `ANY` or a string not described in *Appendix B*, the DSA uses special rules for handling the attribute. Values of the attribute are stored and returned as is; there is no attempt to decode the values and no constraint checking is performed. Also, any attempt to match values of the attribute will evaluate to undefined. For information about declaring constraints for an attribute syntax, see *Parsing rules for attributeSyntax* on page 66.
`multi-valued`	If the attribute can be multi-valued, `TRUE`; otherwise, `FALSE`.
`collective`	If the attribute is collective, `TRUE`; otherwise, `FALSE`.
`userModifiable`	Specifies whether the attribute can be modified by a user. It must be set to `TRUE` (the default) for user-defined attributes; and is usually `FALSE` for operational attributes. The flag is enforced by the DSA's schema-checking behaviour, which can be controlled through the schema-checking options for each operation. When schema-checking is set to `ignoreUserModifiableFlag`, the DSA enforces the schema checking requirements unless `userModifiable` is `TRUE`.

`application`	Specifies whether the attribute is a user or operational attribute (and the kind of operational attribute). When creating user-defined attributes, this component should always be omitted (in which case it will default to userApplications) because it is not meaningful to create a user-defined operational attribute.

Parsing rules for attributeSyntax

The DSA has a set rules for parsing the `attributeSyntax` field. When the provided string in the `attributeSyntax` field conforms to these parsing rules, the DSA will recognise the syntax and handle it more effectively. When the string in the `attributeSyntax` field cannot be parsed by these rules, it is treated as `ANY` (described above).

There are implications of recognising the syntax of an enumerated INTEGER syntax or ENUMERATED syntax. The enumeration values can be used by the system in place of the numeric values they represent when displaying and parsing values of these syntaxes.

Additionally, there are implications of recognising the syntax of constrained string syntax. When users add or modify attributes with this syntax, a DUA (such as Access Presence) may be able to present a list of candidate values instead of a free-text input field.

For example, the `attributeSyntax` component allows:

- integer types to have named number lists, with the names appearing in the LDAP-specific encoding. For example: `INTEGER {xf (0), caas (1), other (2)}`
- enumerated types to have names appear in the LDAP-specific encoding.
- value constraints on string, integer and enumerated types. For example:
  ```
  INTEGER (0..5 | 9..MAX)
  IA5String ("captain" | "major" | "colonel")
  ```
- size constraints to be declared for string types. For example:
  ```
  PrintableString (SIZE (3..8))
  PrintableString (SIZE (1..MAX))
  OCTET STRING (SIZE (0..5 | 9..MAX))
  ```
- a DirectoryString to have an upper bound. For example:
  `DirectoryString {256}` or `DirectoryString {ub-common-name}`
  Note that `MAX` is an invalid value for the parameter of a `DirectoryString`, (that is, `DirectoryString {MAX}` is invalid syntax)

Several predefined upper bounds (for example, `ub-common-name`) are recognized as the parameter to a DirectoryString or as the upper bound in a range. For example, `(1..ub-serial-number)`.

They are, however, treated as equivalent to `MAX` if the bounds parameter is absent or set to `off` (see page 44). If bounds is set to `on` then they have their suggested value according to the UpperBounds module in the ASN.1 sources. An explicit integer value for an upper bound will always be enforced.

The ASN.1 definition of `attributeSyntax` is:

```
attributeSyntax
      Type ::=   IntegerType | EnumeratedType | ConstrainedType
      | ParameterizedType
      ActualParameter ::= number | DefinedValue
      ConstrainedType ::= ParentType Constraint
```

Type

```
IntegerType          ::=  "INTEGER" | "INTEGER" "{" NamedNumberList
"}"

NamedNumberList  ::= NamedNumber | NamedNumberList "," NamedNumber
NamedNumber          ::=  identifier "(" SignedNumber ")"

EnumeratedType   ::= "ENUMERATED" "{" Enumeration "}"
Enumeration      ::= Enumeration | EnumerationItem "," Enumeration
EnumerationItem ::=  identifier | NamedNumber

ParameterizedType ::=     "DirectoryString" "{" ActualParameter "}" |
          "NillableDirectoryString" "{" ActualParameter "}"
```

ActualParameter

```
ActualParameter ::= number | DefinedValue

DefinedValue ::=
"ub-common-name" |
"ub-name" |
"ub-match" |
"ub-knowledge-information" |
"ub-pseudonym" |
"ub-locality-name" |
"ub-state-name" |
"ub-street-address" |
"ub-organization-name" |
"ub-organizational-unit-name" |
"ub-title" |
"ub-description" |
"ub-business-category" |
"ub-postal-string" |
"ub-postal-code" |
"ub-post-office-box" |
ub-physical-office-name" |
"ub-directory-string-first-component-match" |
"ub-localeContextSyntax" |
"ub-content" |
"ub-tag" |
"ub-domainLocalID" |
"ub-search" |
"ub-answerback" |
"ub-country-code" |
"ub-destination-indicator" |
"ub-international-isdn-number" |
"ub-postal-line" |
```

```
"ub-privacy-mark-length" |
"ub-serial-number" |
"ub-surname" |
"ub-telephone-number" |
"ub-teletex-terminal-id" |
"ub-telex-number" |
"ub-user-password" |
"ub-x121-address" |
"ub-schema"
```

ConstrainedType

```
ConstrainedType ::= ParentType Constraint

ParentType ::=
      IntegerType |
      EnumeratedType |
      "OCTET" "STRING" |
      "NumericString" |
      "PrintableString" |
      "VisibleString" |
      "IA5String" |
      "TeletexString" |
      "VideotexString" |
      "GraphicString" |
      "GeneralString" |
      "ObjectDescriptor" |
      "BMPString" |
      "UniversalString" |
      "UTF8String"

Constraint       ::= "(" ElementSetSpec ")"
ElementSetSpec   ::= Elements | Elements "|" ElementSetSpec
Elements         ::= SubtypeElements | "(" ElementSetSpec ")"
SubtypeElements  ::= Value | ValueRange | SizeConstraint
Value     ::= SignedNumber | EnumeratedValue | CharacterStringValue
ValueRange       ::= LowerEndValue ".." UpperEndValue
LowerEndValue::= Value | "MIN"
UpperEndValue::= Value | "MAX" | DefinedValue
SizeConstraint   ::= SIZE Constraint
SignedNumber     ::= number | "-" number
EnumeratedValue  ::= identifier
CharacterStringValue ::= cstring
```

Examples

The following Stream DUA script reads all values of the `attributeTypes` attribute in the demonstration directory, Deltawing. Because schema attributes are available throughout a subschema area, the attributes can be read from the `Deltawing` entry.

```
read {
      organizationName "Deltawing"
}
return { attributeTypes };
```

The following script adds a user-defined attribute `supervisor` with object identifier `{1 3 32 0 2 0 4 42}` and syntax `DirectoryString`. Schema attributes can only be modified at the subentry within which they are held, so the modify operation must be directed to the subschema subentry:

```
modify {
      organizationName "Deltawing"
      / commonName "Subschema"
}
with changes {
      add values attributeTypes {
          identifier {1 3 32 0 2 0 4 42},
          name { "supervisor", "foreman" },
          information {
              derivation name,
              attributeSyntax "DirectoryString{ub-name}"
          }
      }
};
```

NOTE: There is no need to supply an `attributeSyntax` in this case as the syntax is implied by the `derivation`. It is shown for illustrative purposes only.

objectClasses

This operational attribute defines the object classes in the subschema area. It is a multi-valued attribute, each value defining one object class. Both built-in and user-defined object classes must be defined in this way. The attribute is added to the subschema subentry.

The ASN.1 definition is:

```
objectClasses    ATTRIBUTE  ::= {
      WITH SYNTAX               ObjectClassDescription
      EQUALITY MATCHING RULE    objectIdentifierFirstComponentMatch
      USAGE                     directoryOperation
      ID                        {ds 21 6} }
ObjectClassDescription      ::= SEQUENCE {
      identifier         OBJECT-CLASS.&id,
      name               SET OF DirectoryString { ub-schema } OPTIONAL,
      description        DirectoryString { ub-schema } OPTIONAL,
      obsolete           BOOLEAN DEFAULT FALSE,
      information    [0] ObjectClassInformation }
ObjectClassInformation  ::= SEQUENCE {
      subclassOf         SET OF OBJECT-CLASS.&id OPTIONAL,
      kind               ObjectClassKind DEFAULT structural,
      mandatories    [3] SET OF ATTRIBUTE.&id OPTIONAL,
      optionals          [4]  SET OF ATTRIBUTE.&id OPTIONAL }
```

The attribute's components are described below.

ObjectClassDescription

`identifier`	The object identifier of the object class. It is either a built-in symbolic name or a numeric object identifier, for example, `organization` or `{1 3 32 0 2 0 6 27}`.
`name`	The set of symbolic names to be created for the object class.
`description`	A natural language description of the object class. ViewDS does not use this component, but it can be used when publishing schema for third-party DUAs.
`obsolete`	A flag that indicates that the object class is obsolete. Existing entries can continue to use an obsolete object class. However, an obsolete object class cannot be included in a new entry's `objectClass` attribute.
`information`	This component comprises the subcomponents described below.

ObjectClassInformation

`subClassOf`	The set of superclasses for the object class. Each is either a built-in symbolic name, including top or alias, a symbolic name defined in the `name` field of a previously defined `objectClasses` value, or a numeric object identifier.
`kind`	The object class kind, which is either `abstract`, `structural`, or `auxiliary`. If omitted, the default is `structural`. (There is never a need to define an `abstract` object class.)
`mandatories`	A list of mandatory attributes for the object class. Optionally, the list can include the inherited mandatory attributes. Each attribute is specified either as a built-in symbolic name, a symbolic name defined in the `name` field of a previously defined `attributeTypes` value, or a numeric object identifier.
`optionals`	A list of the optional attributes for the object class.

Examples

The following Stream DUA script reads all values of the `objectClasses` attribute:

```
read {
      organizationName "Deltawing"
}
return { objectClasses };
```

The following script defines the built-in object class `person`:

```
modify {
      organizationName "Deltawing"
      / commonName "Subschema"
}
with changes {
      add values objectClasses {
```

```
            identifier { 2 5 6 6 },
            name { "person" },
            information {
                subclassOf { top },
                mandatories { commonName, surname },
                optionals {
                    description,
                    telephoneNumber,
                    userPassword,
                    seeAlso
                }
        }
};
```

matchingRules

This operational attribute specifies the matching rules in the subschema area. It is a multi-valued attribute, each value specifying one matching rule. Only the built-in matching rules may be declared; built-in matching rules not declared in this attribute cannot be used in the subschema area. The attribute is added to the subschema subentry.

The ASN.1 definition is:

```
matchingRules    ATTRIBUTE  ::= {
      WITH SYNTAX        MatchingRuleDescription
      EQUALITY MATCHING RULE objectIdentifierFirstComponentMatch
      USAGE                  directoryOperation
      ID                             {ds 21 4} }

MatchingRuleDescription ::= SEQUENCE {
      identifier         MATCHING-RULE.&id,
      name               SET OF DirectoryString { ub-schema } OPTIONAL,
      description    DirectoryString { ub-schema } OPTIONAL,
      obsolete           BOOLEAN DEFAULT FALSE,
      information    [0] DirectoryString { ub-schema } OPTIONAL }
                             -- describes the ASN.1 assertion syntax
```

MatchingRuleDescription

`identifier` The object identifier of the matching rule (preferably its built-in name).

`name` The set of symbolic names to be created for the matching rule.

`description` A natural language description of the matching rule. ViewDS does not use this component, but it can be used when publishing schema for third-party DUAs.

`obsolete` A flag that indicates that the matching rule is obsolete.

`information` The ASN.1 syntax of the matching rule's assertion syntax. This is a string equal to the ASN.1 type name of the assertion syntax defined for the built-in matching rule.

Examples

The following Stream DUA script reads all values of the `matchingRules` attribute:

```
read {
      organizationName “Deltawing”
}
return { matchingRules };
```

The following script defines the built-in matching rule `caseIgnoreMatch`:

```
modify {
      organizationName “Deltawing”
      / commonName "Subschema"
}
with changes {
      add values matchingRules {
          identifier { 2 5 13 2 },
          name { "caseIgnoreMatch" },
          information "DirectoryString{ub-match}"
      }
};
```

matchingRuleUse

This operational attribute identifies the attribute types that may be used with each matching rule. The attribute is added to the subschema subentry.

The ViewDS DSA does not use the `matchingRuleUse` attribute. It is for the benefit of third-party DUAs so that they can determine which matching rules can be used with particular attributes.

The DSA allows any attribute whose attribute syntax is appropriate for any particular matching rule to be used with that matching rule. The ViewDS DUAs have this knowledge built in (since the matching rules supported by the DSA are built in).

NOTE: According to X.500, this attribute is not required where the use of a matching rule with a particular attribute is implied by the attribute definition (that is, the matching rule is the equality, ordering or substrings matching rule for the attribute).

The ASN.1 definition is:

```
matchingRuleUse      ATTRIBUTE  ::= {
      WITH SYNTAX           MatchingRuleUseDescription
      EQUALITY MATCHING RULE objectIdentifierFirstComponentMatch
      USAGE                 directoryOperation
      ID                            {ds 21 8} }

MatchingRuleUseDescription  ::= SEQUENCE {
      identifier         MATCHING-RULE.&id,
      name               SET OF DirectoryString { ub-schema } OPTIONAL,
      description        DirectoryString { ub-schema } OPTIONAL,
      obsolete           BOOLEAN DEFAULT FALSE,
      information          [0]  SET OF ATTRIBUTE.&id }
```

MatchingRuleUseDescription

`identifier`	The object identifier of the matching rule (preferably its built-in name).
`name`	The set of symbolic names to be created for the matching rule. This field is not used by ViewDS and can be omitted.
`description`	A natural language description of the matching rule.
`obsolete`	A flag that indicates that the matching rule is obsolete.
`information`	The set of attribute types that can be used with the matching rule. Each attribute type is specified by either a built-in symbolic name, a symbolic name defined in the `name` field of a previously defined `attributeTypes` value, or a numeric object identifier.

Example

The following Stream DUA script specifies the matching rule `keywordMatch`, and declares two attributes to be used with it.

```
modify {
      organizationName “Deltawing”
      / commonName "Subschema"
}
with changes {
      add values matchingRuleUse {
          identifier keywordMatch,
          information { organizationName, organizationalUnitName }
      }
};
```

nameForms

This operational attribute defines name forms and makes them available throughout a subschema area. It is a multi-valued attribute, each value defining a name form, which must be added to the subschema subentry. Both built-in and user-defined name forms must be defined by this attribute.

```
nameForms     ATTRIBUTE  ::= {
      WITH SYNTAX        NameFormDescription
      EQUALITY MATCHING RULE objectIdentifierFirstComponentMatch
      USAGE                 directoryOperation
      ID                    {ds 21 7} }
NameFormDescription  ::= SEQUENCE {
      identifier         NAME-FORM.&id,
      name               SET OF DirectoryString { ub-schema } OPTIONAL,
      description    DirectoryString { ub-schema } OPTIONAL,
      obsolete           BOOLEAN DEFAULT FALSE,
      information    [0] NameFormInformation }
NameFormInformation  ::= SEQUENCE {
      subordinate    OBJECT-CLASS.&id,
      namingMandatories  SET OF ATTRIBUTE.&id,
      namingOptionals    SET OF ATTRIBUTE.&id OPTIONAL }
```

NameFormDescription

`identifier` The object identifier of the name form (preferably its built-in name).

`name` The set of symbolic names to be created for the name form.

It is recommended that the `name` includes the name of the relevant structural object class, and has the suffix `NameForm`.

`description` A natural language description of the name form. (ViewDS does not use this component, but it may be useful for a third-party DSA.)

`obsolete` A flag that indicates that the name form is obsolete.

`information` This component comprises the subcomponents described below.

NameFormInformation

`subordinate` The name of the object class to which the name form is relevant (‘subordinate’ is a misnomer). It is either a built-in symbolic name, a symbolic name defined in the `name` field of a previously defined `objectClasses` value, or a numeric object identifier.

`naming Mandatories` The set of mandatory naming attributes for this name form. Each attribute is specified by either a built-in symbolic name, a symbolic name defined in the `name` field of a previously defined `attributeTypes` value, or a numeric object identifier.

`namingOptionals` The set of optional naming attributes for this name form.

If more than one non-obsolete name form is defined for a particular object class, the `namingMandatories` and `namingOptionals` must be such that any particular combination of naming attributes is permitted by at most one of those name forms.

Examples

The following Stream DUA script reads all values of the `nameForms` attribute:

```
read {
      organizationName “Deltawing”
}
return { nameForms };
```

The following script adds a definition of a name form called `orgPersonNameForm`, used to name organizational persons.

```
modify {
      organizationName "Deltawing"
      / commonName "Subschema"
}
with changes {
```

```
        add values nameForms {
            identifier { 2 5 15 6 },
            name { "orgPersonNameForm" },
            information {
                subordinate organizationalPerson,
                namingMandatories { commonName },
                namingOptionals { organizationalUnitName }
            }
        }
};
```

dITStructureRules

This operational attribute defines structure rules and makes them available throughout a subschema area. It is a multi-valued attribute, each value defining one structure rule. The attribute is added to the subschema subentry.

The ASN.1 definition is:

```
dITStructureRules    ATTRIBUTE  ::= {
      WITH SYNTAX        DITStructureRuleDescription
      EQUALITY MATCHING RULE integerFirstComponentMatch
      USAGE                 directoryOperation
      ID                            {ds 21 1} }

DITStructureRuleDescription ::= SEQUENCE {
      COMPONENTS OF      DITStructureRule,
      name           [1] SET OF DirectoryString { ub-schema } OPTIONAL,
      description    DirectoryString { ub-schema } OPTIONAL,
      obsolete            BOOLEAN DEFAULT FALSE }

DITStructureRule ::= SEQUENCE {
      ruleIdentifier     RuleIdentifier ,
                       - must be unique within the scope of the subschema
      nameForm           NAME-FORM.&id,
      superiorStructureRules    SET OF RuleIdentifier OPTIONAL }

RuleIdentifier  ::=  INTEGER
```

DITStructureRuleDescription

`name` The set of symbolic names to be created for the structure rule. This field is not used by ViewDS.

`description` A natural language description of the structure rule. This field is not used by ViewDS.

`obsolete` A flag that indicates that the structure rule is obsolete.

DITStructureRule

`ruleIdentifier` The integer identifier of the structure rule. There are no built-in identifiers. An integer should be assigned that is not being used for another structure rule in the same subschema area.

`nameForm`	The name form to which this structure rule pertains. It is either a built-in symbolic name, a symbolic name defined in the `name` field of a previously defined `nameForms` value, or a numeric object identifier.
`superior StructureRules`	The set of superior structure rules for this structure rule. Each superior structure rule is specified by its `ruleIdentifier`, an integer. If more than one non-obsolete structure rule is defined for a particular name form, they must be such that any particular structure rule appears at most once in the `superiorStructureRules` of all those structure rules.

Examples

The following Stream DUA script reads all values of the `dITStructureRules` attribute:

```
read {
      organizationName "Deltawing"
}
return { dITStructureRules };
```

The following script adds a structure rule for organizational persons, and assigns it identifier 23. Entries with this structure rule must be named as per `orgPersonNameForm`, and must have an immediate superior entry with structure rule 11 or 12.

```
modify {
      organizationName "Deltawing"
      / commonName "Subschema"
}
with changes {
      add values dITStructureRules {
          ruleIdentifier 23,
          nameForm orgPersonNameForm,
          superiorStructureRules { 11, 12 },
          name { "orgPersonSR" }
      }
};
```

dITContentRules

This operational attribute defines content rules and makes them available throughout a subschema area. Content rules are used to specify the auxiliary object classes and additional attributes permitted in an object class. It is a multi-valued attribute, each value specifying one content rule. The attribute is added to the subschema subentry.

The ASN.1 definition is:

```
dITContentRules      ATTRIBUTE  ::= {
      WITH SYNTAX           DITContentRuleDescription
      EQUALITY MATCHING RULEobjectIdentifierFirstComponentMatch
      USAGE                 directoryOperation
      ID                            {ds 21 2} }
```

```
DITContentRuleDescription   ::= SEQUENCE {
      COMPONENTS OF      DITContentRule,
      name           [4] SET OF DirectoryString { ub-schema } OPTIONAL,
      description        DirectoryString { ub-schema }OPTIONAL,
      obsolete             BOOLEAN DEFAULT FALSE}

DITContentRule   ::= SEQUENCE {
      structuralObjectClass OBJECT-CLASS.&id,
      auxiliaries        SET OF OBJECT-CLASS.&id OPTIONAL,
      mandatory      [1] SET OF ATTRIBUTE.&id OPTIONAL,
      optional       [2] SET OF ATTRIBUTE.&id OPTIONAL,
      precluded      [3]  SET OF ATTRIBUTE.&id OPTIONAL }
```

DITContentRule

`structuralObjectClass` The object identifier of the structural object class with which this content rule is associated. It is either a built-in symbolic name, a symbolic name defined in the `name` field of a previously defined `objectClasses` value, or a numeric object identifier.

`auxiliaries` The set of allowed auxiliary object classes for entries with the associated structural object class. Each object class is specified by either a built-in symbolic name, a symbolic name defined in the `name` field of a previously defined `objectClasses` value, or a numeric object identifier.

If no content rule exists or this field is absent, no auxiliary object classes are permitted for entries with this structural object class.

`mandatory` The set of additional mandatory attributes for entries with this structural object class. Each attribute is specified either a built-in symbolic name, a symbolic name defined in the `name` field of a previously defined `attributeTypes` value, or a numeric object identifier.

`optional` The set of additional optional attributes for entries with this structural object class.

`precluded` The set of precluded optional attributes for entries of this structural object class. It disallows optional attributes otherwise allowed by the `optionals` field of the entry's object classes.

`name` The set of symbolic names to be created for the content rule. This field is not used by ViewDS and can be omitted.

`description` A natural language description of the content rule. ViewDS does not use this field.

`obsolete` A flag that indicates whether the content rule is obsolete.

Examples

The following Stream DUA script reads all values of the `dITContentRules` attribute:

```
read {
      organizationName "Deltawing"
}
return { dITContentRules };
```

The following script adds a content rule for the structural object class `organizationalPerson`, specifying `mhs-user` as a permitted auxiliary object class, `sortSubs` as an additional optional attribute, and `teletexTerminalIdentifier` as an excluded operational attribute.

```
modify {
      organizationName "Deltawing"
      / commonName "Subschema"
}
with changes {
      add values dITContentRules {
          structuralObjectClass organizationalPerson,
          auxiliaries { mhs-user },
          optional { sortSubs },
          precluded { teletexTerminalIdentifier }
      }
};
```

definitions

This operational attribute is defined in the XML Enabled Directory (XED) specification. It holds user-provided *XML schema language documents* and makes their data type definitions available for defining attribute syntaxes in a subschema area.

It is a multi-valued attribute. A particular XML schema language document may appear in more than one subschema subentry; in which case, the same `identifier` must be used in each subschema subentry.

The ASN.1 definition is:

```
definitions      ATTRIBUTE  ::= {
      WITH SYNTAX               IdentifiedSchema
      EQUALITY MATCHING RULE    schemaIdentityMatch
      USAGE                     directoryOperation
      ID                                { 1 3 6 1 4 1 21472 5 21 0 }
}

IdentifiedSchema ::= SEQUENCE {
      identifier     SchemaIdentity,
      document       SchemaDocument }

SchemaDocument ::= CHOICE {
      module     [RXER:ELEMENT-REF {
                     namespace-name
                         "http://xmled.info/ns/ASN.1",
                     local-name "module" }] AnyType,
      schema     [RXER:ELEMENT-REF {
                     namespace-name
                         "http://www.w3.org/2001/XMLSchema",
```

```
                    local-name "schema" }] AnyType,
      grammar    [RXER:ELEMENT-REF {
                          namespace-name
                              "http://relaxng.org/ns/structure/1.0",
                          local-name "grammar" }] AnyType,
          dtd             ExternalSubset }
ExternalSubset ::= UTF8String (CONSTRAINED BY {
          -- contains an external DTD subset,
          -- i.e., text conforming to
               -- the extSubset production of XML -- })
SchemaIdentity ::=  AnyURI
AnyURI ::= UTF8String (CONSTRAINED BY
                    { -- conforms to the format of a URI -- })
```

Identifier

A URI that uniquely identifies the XML schema language document.

document

The `document` component holds as a child element (within an `AnyType` value, see *RFC 4910*) the XML schema language document being imported into a subschema. The child element must be either:

- a ‘schema’ element from the namespace ‘http://www.w3.org/2001/XMLSchema’ (that is, an XML Schema)
- an external DTD subset (that is, text conforming to the extSubset production of XML)
- a ‘module’ element from the namespace ‘http://xmled.info/ns/ASN.1’ (that is, an ASN.X module)
- a ‘grammar’ element from the namespace ‘http://relaxng.org/ns/structure/1.0’ (that is, a RELAX NG schema)

The last two options (module and grammar) are unsupported in this version of ViewDS.

subtreeSpecification

This operational attribute is in every subentry and defines the set of entries controlled by the subentry. Its syntax is also used to define areas of replication (see *Replicating or distributing data* on page 147) although these are not managed through subentries. It must be added when a subentry is created.

The ASN.1 definition is:

```
subtreeSpecification ATTRIBUTE  ::= {
      WITH SYNTAX           SubtreeSpecification
      SINGLE VALUE          TRUE
      USAGE                 directoryOperation
      ID                            {ds 18 6}
}
```

```
SubtreeSpecification ::= SEQUENCE {
      base               [0] LocalName DEFAULT {},
      COMPONENTS OF          ChopSpecification,
      specificationFilter [4]  Refinement OPTIONAL }

LocalName ::= RDNSequence

ChopSpecification ::= SEQUENCE {
      specificExclusions [1] SET OF CHOICE {
      chopBefore         [0] LocalName,
      chopAfter          [1] LocalName } OPTIONAL,
      minimum            [2] BaseDistance DEFAULT 0,
      maximum            [3] BaseDistance OPTIONAL }
BaseDistance ::= INTEGER (0..MAX)

Refinement ::= CHOICE {
      item    [0] OBJECT-CLASS.&id,
      and     [1] SET OF Refinement,
      or      [2] SET OF Refinement,
      not     [3] Refinement }
```

Other operational attributes

These operational attributes provide information about a schema or its entries.

governingStructureRule

This operational attribute is calculated by the DSA when an entry is added to the directory. Its value is the governing structure rule of the new entry, which may change if the entry is renamed or moved.

```
governingStructureRule   ATTRIBUTE  ::= {
      WITH SYNTAX           INTEGER
      EQUALITY MATCHING RULE integerMatch
      SINGLE VALUE          TRUE
      NO USER MODIFICATION  TRUE
      USAGE                 directoryOperation
      ID                    {ds 21 10}
}
```

structuralObjectClass

This operational attribute is calculated by the DSA when an entry is added to the directory. Its value is the structural object class of the entry, which is the same as one of the `objectClass` attribute's values.

```
structuralObjectClass    ATTRIBUTE  ::= {
      WITH SYNTAX           OBJECT IDENTIFIER
      EQUALITY MATCHING RULE objectIdentifierMatch
      SINGLE VALUE          TRUE
      NO USER MODIFICATION  TRUE
      USAGE                 directoryOperation
      ID                    {ds 21 9}
}
```

subschemaTimestamp

This operational attribute provides the last modification time of any subschema operational attribute. It is used by Access Presence to determine whether its cached copy of the schema attributes is current. It is available from any entry in the subschema area.

```
subschemaTimestamp   ATTRIBUTE  ::= {
      WITH SYNTAX           GeneralizedTime
      EQUALITY MATCHING RULE generalizedTimeMatch
      ORDERING MATCHING RULE generalizedTimeOrderingMatch
      SINGLE VALUE          TRUE
      NO USER MODIFICATION  TRUE
      USAGE                 directoryOperation
      ID                    {ds 18 8}
}
```

subschemaSubentry

This operational attribute provides the DN of the governing subschema subentry of an entry.

```
subschemaSubentry    ATTRIBUTE ::=  {
      WITH SYNTAX           DistinguishedName
      EQUALITY MATCHING RULE distinguishedNameMatch
      SINGLE VALUE              TRUE
      NO USER MODIFICATION      TRUE
      USAGE                     directoryOperation
      ID                        id-oa-subschemaSubentry
}
```

numberOfMasterEntries

This read-only operational attribute is only available in the root entry of the DSA. It returns the number of master entries stored in the directory, which can then be published by the DSA.

The attribute has the following definition:

```
numberOfMasterEntries ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      USAGE                     dSAOperation
      ID  { iso(1) 2 36 79672281 directory(1) 12 0 }
}
```

numberOfShadowEntries

This read-only attribute is only available in the root entry of the DSA. It returns the number of replicated entries stored in the directory, which can then be published by the DSA.

It has the following definition:

```
numberOfShadowEntries ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      USAGE                     dSAOperation
      ID  { iso(1) 2 36 79672281 directory(1) 12 1 }
}
```

Time and date attributes

The following operational attributes relate to time and date – all are generated dynamically, except for `userTimeZone`. They are particularly useful when declaring role-based access controls (see page 124) that support time-based refinements.

`currentDateTime`	The DSA's current local date and time in ISO 8601 format with the difference from UTC. For example: `2007-09-19T15:49:10+10:00`
`currentTimeOfDay`	The DSA's current local time in ISO 8601 format with the difference from UTC. For example: `15:49:10+10:00`
`currentDayOfWeek`	The current day of the week in the DSA's time zone: `monday`, `tuesday`, `wednesday`, `thursday`, `friday`, `saturday` or `sunday`. For ordering purposes, `monday` has the lowest value and `sunday` has the highest value (this aligns with ISO 8601).
`userDateTime`	The user's local date and time in ISO 8601 format. For example: `2007-09-19T13:49:10`. The time zone is determined by the `userTimeZone` attribute.
`userTimeOfDay`	The user's local time of day in ISO 8601 format. For example: `13:49:10`
`userDayOfWeek`	The current day of the week in the user's time zone.
`userTimeZone`	A user-modifiable attribute that represents the time zone of the user associated with the entry. If the attribute is absent, the user is assumed to be in the same time zone as the DSA. Its format is the time zone difference from UTC – the trailing part of a `currentDateTime` or `currentTimeOfDay` value. For example: `+08:00`

The corresponding matching rules are described in *Appendix B*.

Chapter 5
Indexes, extensions and word lists

This chapter describes indexes and word lists (synonyms, noise words, truncated words) which help optimize searches on a directory. It also describes attribute-type extensions that allow an attribute's value to be hashed, tracked (if it is a Distinguished Name), or written to a separate file when the database is dumped.

The chapter has the following sections:

- Concepts
- Indexes
- Operational attributes
- Word lists

Concepts

Indexes and attribute-type extensions are configured through operational attributes stored in a *schema configuration subentry* (see Figure 3 on page 52); while word lists are stored in a *subschema subentry*.

The operational attributes for indexes, word lists and attribute-type extensions apply to all entries in the same *information plane* – an information plane comprises either *master data* or *shadow data*. Master data comprises entries that are not supplied by another Directory System Agent (DSA) through replication; shadow data comprises entries supplied by another DSA through replication (a *supplier* DSA sends entries to a *consumer* DSA).

If an information plane has multiple subschema areas, then the indexes, extensions and word lists are cumulative.

This chapter describes how to modify these operational attributes using the *Stream Directory User Agent* (Stream DUA). However, they can also be configured through the ViewDS Management Agent.

Indexes

Like all databases, ViewDS uses indexes to speed up access to its data. In ViewDS, an index is a list of all entries sorted by a particular property. The index allows the DSA to find the set of entries matching a search filter very rapidly.

Unless a base entry other than the root is specified in a search request, ViewDS requires at least one of the attributes in the request to be indexed before it will perform the search.

It is normal to index most of the user attributes in a ViewDS directory. Exceptions might be very long free-text attributes (for example, narrative text) for which the time and space required for an index cannot be justified.

Attribute-type extensions

DN tracking

DN (Distinguished Name) tracking applies to an attribute type with a syntax that is (or contains) a DN that references some other entry in the directory. When DN tracking is enabled for an attribute type, it causes the DNs in its values to be automatically updated when the referenced entry, or one of its superiors, is moved or renamed.

To illustrate, consider the `manager` attribute of the Deltawing entry in the demonstration directory, Deltawing. By default, this attribute references the DN of the entry for 'Margaret Hunter'. If the entry for 'Margaret Hunter' is moved to another location in the DIT, then DN tracking ensures that the manager attribute in the Deltawing entry still references Margaret's entry correctly.

Attribute hashing

By default, the DSA stores and returns attribute values as clear text. It might be desirable, however, to perform a one-way hash on the value of an attribute before storing or returning it (for example, for attributes holding user passwords).

Dumping to a separate file

If an attribute has the syntax of OCTET STRING or BIT STRING, its values can be written to a separate file when the database is dumped. This is useful for attributes that store, for example, documents or images. Dumping to separate files allows a document or image to be opened with an appropriate application.

To enforce security, the `userPassword` attribute cannot be dumped to a separate file.

Word lists

There are three categories of word lists that can be associated with an attribute type:

- **Synonyms** – a set of words treated as equivalent when a user requests an approximate match on one of the words in the set. For example, a set of synonyms might be 'high school', 'secondary college' and 'secondary school'.
- **Noise words** – these are keywords to be ignored when keyword-matching an attribute value. They are usually words – such as 'the', 'and', 'a' – that are so common to be rendered of little use for searching or indexing.
- **Truncated words** – these are preferred truncations for keywords to be used by the DSA when building the `hierarchyName` attribute (see *Technical Reference Guide: User Interfaces*).

Indexes

This subsection describes:

- Different kinds of indexes
- Specifying indexing and approximate matching
- Index maintenance
- Rebuilding indexes
- Virtual View List indexes

Different kinds of indexes

An attribute type can be indexed in many ways:

- *entry index* – identifies entries that have a matching value in their *entry information* (attribute values). This excludes collective attributes, which only *appear* to be present in an entry.
- *RDN index* – identifies entries that have a matching value in their Relative Distinguished Name (RDN).
- *DN index* – identifies entries that have a matching value in their *DN.*
- *collective index* – identifies entries that have a matching value in a *collective attribute* (that appears to be present in the entry).

An additional index, a *base-object index*, can be defined for an entire Directory Information Tree (DIT). It identifies all entries that are subordinate to any given entry (subtree enumeration).

With the exception of a base-object index, each index has a ***kind*** that determines the kind of matching it supports.

```
IndexKind ::= ENUMERATED {
      none                          (0),
      -- special --
      presence                      (1),
      equality                      (2),
      substrings                    (3),
      approximate                   (4),
      -- equality indexes --
      case-ignore-string           (10),
      case-exact-string            (11),
      numeric-string               (12),
      telephone-string             (13),
      list-string                  (14),
      utc-time                     (15),
      generalized-time             (16),
      integer                      (17),
      bit-string                   (18),
      octet-string                 (19),
      boolean                      (20),
      object-identifier            (21),
      mapped-string                (22),
      decimal                      (23),
      truncated-date-time          (24),
      qualified-name               (25),
      distinguished-name           (26),
```

```
        mhs-case-ignore-string          (27),
        mhs-all-case-ignore-string      (28),
        xacml-policy-id                 (29),
        xacml-policy-set-id             (30),
        xacml-embedded-policy-id        (31),
        xacml-embedded-policy-set-id    (32),
        xacml-named-expression-id       (33),
        xacml-embedded-expression-id    (34),
        xacml-rule-id                   (35),
        -- substring indexes --
        case-ignore-substring           (40),
        case-exact-substring            (41),
        numeric-substring               (42),
        telephone-substring             (43),
        list-substring                  (44),
        octet-substring                 (49),
        mhs-case-ignore-substring       (51),
        mhs-all-case-ignore-substring   (52),
        -- approximate match indexes --
        abbreviation                    (70),
        synonym                         (71),
        phonetic                        (72),
        typing                          (73),
        keyword                         (74),
        keyword-synonym                 (75),
        keyword-phonetic                (76),
        keyword-typing                  (77),
        mapped-keyword                  (79),
        keyword-phonetic-mandarin       (80)
}
```

The indexes are described below.

Special indexes

`presence` Specifies a presence index, which is used to support presence matching. It is useful when no other indexing is specified, as an equality index also supports presence matching.

Equality indexes

A string equality index supports equality and substring matches with a supplied substring.

`case-ignore-string`
`case-exact-string`

Supports case-insensitive and case-sensitive string matching of values of the syntaxes:

- DirectoryString{} ASN.1 type
- ASN.1 restricted string type (for example, `PrintableString`)
- XML Schema string type (or subtypes/restrictions thereof)

A `case-ignore-string` index is normally very effective for case-sensitive matching. Therefore, a `case-exact-string` index is usually unnecessary if there is also a `case-ignore-string` index.

`numeric-string`	Supports numeric string matching (for example, using `numericStringMatch`) of values of the `NumericString` ASN.1 type (or a subtype thereof).
`telephone-string`	Supports telephone number matching (for example, using `telephoneNumberMatch`) of values of the `TelephoneNumber` or `PrintableString` ASN.1 type (or a subtype thereof).
`list-string`	Supports case-insensitive matching of values of a SEQUENCE OF DirectoryString{} ASN.1 type (for example, `PostalAddress`).
`utc-time`	Supports equality matching of values of the UTCTime ASN.1 type (or a subtype thereof).
`generalized-time`	Supports equality matching of values of the GeneralizedTime ASN.1 type (or a subtype thereof) or the XML Schema dateTime type (or a restriction thereof).
`integer`	Supports equality matching of values of the INTEGER or ENUMERATED ASN.1 type (or subtypes thereof) or the XML Schema integer type (or a restriction thereof).
`bit-string`	Supports equality matching of values of the BIT STRING ASN.1 type (or a subtype thereof).
`octet-string`	Supports equality and leading substring matching of values of the OCTET STRING ASN.1 type (or a subtype thereof); or the XML Schema hexBinary or base64binary types (or restrictions thereof).
`boolean`	Supports equality matching of values of the BOOLEAN ASN.1 type or the XML Schema Boolean type.
`object-identifier`	Supports equality matching of values of the OBJECT IDENTIFIER ASN.1 type (or a subtype thereof).
`decimal`	Supports equality matching of values of the XML Schema decimal type (or a restriction thereof).
`truncated-date-time`	Supports equality matching of values of the XML Schema time, date, gYearMonth, gYear, gMonthDay, gDay or gMonth type (or restrictions thereof).
`qualified-name`	Supports equality matching of values of the XML Schema QName type (or a restriction thereof).
`distinguished-name`	Supports equality matching of values of the `RDNSequence` ASN.1 type (or a subtype thereof, such as `DistinguishedName`).

If the type of an attribute syntax is not one of the ASN.1 or XML Schema types referred to by the preceding paragraphs then whole values of that attribute cannot be indexed.

Substring indexes

A substring index supports arbitrary substring (including leading wildcard) matches. Substring indexes are potentially much larger than equality indexes, and you should only specify them if you expect typical search requests to include leading wildcard substring matches.

`case-ignore-substring`
`case-exact-substring`

Support case-insensitive or case-sensitive substring matching of values of the following syntaxes:

- DirectoryString{} ASN.1 type
- ASN.1 restricted string type (for example, `PrintableString`)
- XML Schema string type (or subtypes/restrictions thereof).

A `case-ignore-substring` index is normally very effective for case-sensitive matching. Therefore, a `case-exact-substring` index is usually unnecessary if there is also a `case-ignore-substring` index.

`numeric-substring`

Supports numeric string substring matching (for example, using `numericStringSubstringsMatch`) of values of the `NumericString` ASN.1 type (or a subtype thereof).

`telephone-substring`

Supports telephone number substring matching (for example, using `telephoneNumberSubstringsMatch`) of values of the `TelephoneNumber` or `PrintableString` ASN.1 type (or a subtype thereof).

`list-substring`

Supports substring matching of values of a SEQUENCE OF DirectoryString{} ASN.1 type.

`octet-substring`

Supports substring matching of values of the OCTET STRING ASN.1 type (or a subtype thereof) or the XML Schema hexBinary or base64binary types (or restrictions thereof).

X.400 indexes

`mhs-case-ignore-string`
`mhs-case-ignore-substring`

Support individual string components of attribute values of ORName and ORAddress syntax. For example:

```
{
      type mhs-dl-members,
      component "built-in-standard-
      attributes.organization-name",
      index mhs-case-ignore-string
}
```

`mhs-all-case-ignore-string`

`mhs-all-case-ignore-substring`

Support individual values of the ORName and ORAddress syntax to build an index of all the character strings of the value, irrespective of which component they belong to.

For example:

```
{
     type mhs-or-addresses,
     index mhs-all-case-ignore-string
}
```

X.400 matching rules

The X.400 matching rules are described in *Appendix B* on page 225. The relationships between the X.400 indexes and matching rules are outlined below.

X.400 matching rule	Uses X.400 index
`oRNameSingleElementMatch`	Uses the `mhs-all-case-ignore-string` index for the attribute, if it is available

`orAddressMatch`
`orAddressElementsMatch`
`oRNameExactMatch`
`orNameMatch`
`oRNameElementsMatch`
	Use whatever `mhs-case-ignore-string` component indexes are available, but will fallback to using the `mhs-all-case-ignore-string` index on the attribute, if it is available.

`oRAddressSubstringElements Match` `oRNameSubstringElements Match`	Use whatever `mhs-case-ignore-substring` indexes are available, but will fallback to using the `mhs-all-case-ignore-substring` index on the attribute, if it is available. This is the only scenario when the `mhs-all-case-ignore-substring` index is used.

When enabling component indexes for `ORName` and `ORAddress` attribute values it is important to note that many of the components of these values are semantically paired. It is therefore necessary to enable an index on each component in the pair.

For example, the following make up a pair:

```
built-in-standard-attributes.administration-domain-name.numeric
built-in-standard-attributes.administration-domain-name.printable
```

Another example pair is:

```
built-in-standard-attributes.organization-name
extension-attributes.*.extension-attribute-value.(3)
```

The simplest approach is to configure the `mhs-all-case-ignore-string` index on the attribute type and rely on the fallbacks. This will give reasonable lookup performance.

Approximate indexes

`abbreviation`
`synonym`
`phonetic`
`typing`
	These are approximate whole-value indexes that support the following approximate match types, respectively:
	• `nk-abbrev`
	• `nk-synonym`
	• `nk-phonetic`
	• `nk-typing`
	See `attributeTypeExtensions` on page 95.
	Note: `nk-prefix` is supported by an equality index kind.

`keyword-equal`
`keyword-synonym`
`keyword-phonetic`
`keyword-typing`
`keyword-phonetic-mandarin`
	These are approximate keyword indexes that support the following approximate match types, respectively:
	• `key-equal`
	• `key-synonym`
	• `key-phonetic`
	• `key-typing`
	• `key-phonetic-mandarin`
	See `attributeTypeExtensions` on page 95.

Specifying indexing and approximate matching

Indexing and approximate matching are specified through six operational attributes:

- `attributeTypeExtensions`
- `baseObjectIndexing`
- `entryIndexing`
- `rdnIndexing`
- `dnIndexing`
- `collectiveIndexing`

There is an additional operational attribute that temporarily defers index building (useful when, for example, first loading the database): `indexingDisabled`.

Several operational attributes are indexed even if indexing is not specified by the `entryIndexing` or `attributeTypeExtensions` attribute. This is necessary for ViewDS to operate properly. If you specify more indexing than ViewDS needs (for example, `object-identifier` where the DSA only needs `presence`), then this additional indexing is performed as well as ViewDS's required indexing.

Scope of indexes

The operational attributes for indexes are stored in a schema configuration subentry (see Figure 3 on page 52). The indexes they define apply to all subschema areas in the same information plane (see *Concepts* on page 83).

The operational attributes can also be added to the root entry from where they are inherited by each schema configuration subentry subsequently created. This is of use when the supplier in a replication agreement is a non-ViewDS DSA.

When the supplier is a non-ViewDS DSA, the operational attributes should be added to the root of the consumer DSA. In this scenario – because the supplier does not

provide a configuration subentry – the consumer will automatically create one for the shadow plane. This new configuration subentry then inherits the indexing operational attributes from the root.

Index maintenance

The DSA maintains all specified indexes automatically, unless indexing is explicitly disabled through the `indexingDisabled` attribute (see page 103). The DSA does the following:

- maintains the indexes when entry information is modified through directory protocol update operations (for example, in DAP, LDAP or XLDAP).
- builds, deletes or rebuilds indexes when index specifications are modified (through the ViewDS Management Agent or Stream DUA).
- updates the relevant synonym index when a synonym is added or removed.

Rebuilding indexes

The DSA maintains all specified indexes automatically, although you may need to rebuild them manually if they become corrupted.

The database's indexes can be rebuilt in a single atomic transaction. During the rebuild:

- Query operations can be performed using the old version of the indexes.
- Update operations are blocked (the DSA returns a `serviceError` with the problem `busy`).

Rebuilding all indexes

To rebuild all indexes as a single transaction:

```
modify {
      organizationName "Deltawing"
      / commonName "Schema Configuration" }
with changes {
      add attribute indexingDisabled NULL,
      remove attribute indexingDisabled };
```

Rebuilding specific indexes

To rebuild specific indexes, remove and add the appropriate index configuration attributes in a single modify operation. For example, to rebuild the synonym index for `organizationalUnitName`:

```
modify {
      organizationName "Deltawing"
      / commonName "Schema Configuration" }
with changes {
      remove values entryIndexing {
          type    organizationalUnitName
          value   synonym },
      add values entryIndexing {
          type    organizationalUnitName
          value   synonym }
};
```

Virtual View List indexes

A Virtual List View is a way to return a set of data to a third-party application. For example, an email client can be configured to make an LDAP connection to ViewDS, extract the entries identified by a Virtual List View defined at the DSA, and use them to populate its address book.

The indexes in this subsection are required to evaluate a Virtual List View search. They can be configured through the ViewDS Management Agent (see the help topic *Define a Virtual List View*).

viewDSListIndexing

An index for evaluating a Virtual List View request is configured by adding a value to the `viewDSListIndexing` schema operational attribute. This attribute should be in a schema configuration subentry with the other index configuration attributes (see page *Subschema area, administrative point and subentry* on page 51).

```
viewDSListIndexing ATTRIBUTE ::= {
      WITH SYNTAX ListIndexDescription
      EQUALITY MATCHING RULE allComponentsMatch
      USAGE directoryOperation
      ID { 2 16 840 1 113730 3 4 9 }
}
ListIndexDescription ::= SEQUENCE {
      filter [0] Filter,
      order [1] SortKeyList
}
SortKeyList ::= SEQUENCE OF SEQUENCE {
      attributeType AttributeDescription,
      orderingRule [0] MATCHING-RULE.&id OPTIONAL,
      reverseOrder [1] BOOLEAN DEFAULT FALSE
}
```

The `filter` is an XLDAP Filter and `SortKeyList` is the request format of the server-side sorting control for XLDAP.

filter

The `filter` must be equivalent to the filter in the LDAP or XLDAP search operation. However, the terms in an ‘and’ or ‘or’ `filter` item can be in any order.

sortKeyList

A search operation with a Virtual List View control must have a sorting control.

The sort keys in the search operation must be the same as (or be a prefix of):

- the sort keys in the index configuration; or
- the sort keys in the index configuration, except the effective value of `reverseOrder` in each sort key is the opposite.

If `orderingRule` is omitted, then it is treated as the implied ordering rule for the attribute type. Attribute options in the sort keys are significant, and there must be at least one sort key specified.

It is not possible to configure an index where the effective value of `reverseOrder` is `TRUE`, or where the `filter` contains a substrings filter item.

If no index matches the search filter and sort keys, then the request returns an error. There is no fallback processing in the absence of a suitable index.

Examples

The following example Stream DUA operation adds an index to support Virtual List View searches for `person` entries ordered on `surname`.

```
modify {
      organizationName "Deltawing"
        / commonName "Schema Configuration"
}
with changes {
      add attribute viewDSListIndexing
      {
          filter equalityMatch:{
              attributeDesc {
                  type objectClass
              },
              assertionValue person
          },
          order {
                  {
                      attributeType {
                      type surname
                      },
                      orderingRule caseIgnoreOrderingMatch
                  },
                  {
                      attributeType {
                      type givenName
                      },
                      orderingRule caseIgnoreOrderingMatch
                  }
          }
      }
};
```

Supplying additional sort keys ensures that results are ordered predictably when entries have the same values for the keys in a search request. So, for above the example, if a client only specifies `surname` for the sort key, entries with the same `surname` will be sorted according to their `givenName` attribute.

The following is an example Stream DUA command for a Virtual List View operation that uses the example index.

```
ldap search {
      organizationName "Deltawing"
}
for (objectClass = "person")
return { surname givenName }
```

```
controls {
      controlType "1.2.840.113556.1.4.473",
      criticality TRUE,
      controlValue containing {
          {
              attributeType "surname",
              orderingRule "caseIgnoreOrderingMatch"
          }
      }
}
{
      controlType "2.16.840.1.113730.3.4.9",
      criticality TRUE,
      controlValue containing {
          beforeCount 0,
          afterCount 5,
          target byOffset:{
              offset 50,
              contentCount 100
          }
      }
};
```

The same index would be used if the sort control were as follows:

```
{
      controlType "1.2.840.113556.1.4.473",
      criticality TRUE,
      controlValue containing {
          {
              attributeType "surname",
              orderingRule "caseIgnoreOrderingMatch",
              reverseOrder TRUE
          }
      }
}
```

Or as follows:

```
{
      controlType "1.2.840.113556.1.4.473",
      criticality TRUE,
      controlValue containing {
          {
              attributeType "surname",
              orderingRule "caseIgnoreOrderingMatch"
          },
          {
              attributeType "givenName",
              orderingRule "caseIgnoreOrderingMatch"
          }
      }
}
```

Operational attributes

The operational attributes for indexing are stored in a schema configuration subentry (see Figure 3 on page 52).

attributeTypeExtensions

This operational attribute specifies additional configuration parameters for attribute types. It is a multi-valued operational attribute, each value identifying an attribute type and additional information.

The ASN.1 type definition is:

```
attributeTypeExtensions ATTRIBUTE   ::= {
      WITH SYNTAX           AttributeTypeExtension
      NO USER MODIFICATION      TRUE
      USAGE                     dSAOperation
      ID                                vf 21 0 }

AttributeTypeExtension ::= SEQUENCE {
      identifier            [0] OBJECT IDENTIFIER,
      indexing              [1] Indexing DEFAULT in-none,
      dnTracking            [2] BOOLEAN DEFAULT FALSE,
      approxMatchType       [3] ApproximateMatchType OPTIONAL,
      valueFileSuffix       [6] BMPString OPTIONAL,
      hashValues            [7] BOOLEAN OPTIONAL,
      hashAlgorithm         [8] UTF8String OPTIONAL,
      returnHash            [9] BOOLEAN OPTIONAL,
      returnTagged          [10] BOOLEAN OPTIONAL,
      deleteValuesReferencingDeleted    [11] BOOLEAN DEFAULT TRUE,
      deleteValuesReferencingMoved      [12] BOOLEAN DEFAULT TRUE
      matchQuality      [13] MatchQualityControl OPTIONAL
}

Indexing ::= ENUMERATED {
      in-none (0), in-partial (1), in-full (2) }

ApproximateMatchType ::= BIT STRING {
      nk-prefix (1), nk-synonym (18), nk-phonetic (13),
      nk-typing (15), key-equal (5), key-prefix (6),
      key-synabb (7), key-phonetic (14), key-typing (16),
      key-suffix (12), nk-abbrev(17), key-phonetic-mandarin(80)
}

MatchQualityControl ::= SEQUENCE {
      bias INTEGER (-100..100)
}
```

identifier

The object identifier of the attribute being extended; it is either a built-in symbolic name or a numeric object identifier.

indexing

`indexing` exists for historical reasons only.

If you add a value of `attributeTypeExtensions` for an attribute and the `indexing` field is set to `in-partial` or `in-full`, the DSA will:

1. Remove the `indexing` field.
2. Add an `entryIndexing` value to create the requested form of indexing.

Further changes to the `indexing` field of `attributeTypeExtensions` will not affect `entryIndexing`.

The possible values for the `indexing` field are described below.

Value	Description
`in-none`	No indexing is specified. This is the default.
`in-partial`	The attribute is indexed for presence. This is converted to a value of `entryIndexing` with an IndexKind of `presence`.
`in-full`	The attribute is indexed for equality. This is converted to a value of `entryIndexing` with the IndexKind appropriate to the equality matching rule of the attribute, and a value of `entryIndexing` with the associated `IndexKind` for each of the approximate match types (see page 96).

dnTracking

`dnTracking` is only relevant for attributes whose syntax is `DistinguishedName` or contains one or more components of type `DistinguishedName` or `LocalName`.

If `TRUE`, the DSA automatically tracks the DN – that is, the attribute's value will change so that it continues to refer to the original entry even if that entry is renamed or moved (on the same DSA).

Set this field is set to `TRUE` for attributes of the applicable syntaxes, unless the attribute has values containing digitally signed content (such as X.509 certificates and CRLs). In this case set it to `FALSE`.

NOTE: The `aliasedEntryName` and `privilege` attributes are always tracked.

Indexing attributes that have DN tracking

An entire DN, or a component of a DN, that is tracked can be indexed.

A component can be indexed if its component reference begins with -1. This is with respect to the DN, which may itself be a component in an encompassing attribute syntax. The remainder of the component reference, if any, is unrestricted.

For example, the component reference `-1.*.value.(2.5.4.3)` is supported for the `seeAlso` attribute, but the component reference `*.*.value.(2.5.4.3)` is not. The DSA will generate an error message if an index is configured with an unsupported component reference for a DN-tracked attribute.

approxMatchType

This specifies the types of approximate matches performed for a string-type attribute. They are applied when a filter request for an approximate match is received for the attribute type.

The possible values are described below.

Approximate match type	Description
`nk-prefix`	Prefix match on whole value.
`nk-synonym`	Synonym match on whole value.
`nk-phonetic`	Phonetic match on whole value.
`nk-typing`	Typing correction match on whole value.
`nk-abbrev`	Abbreviation match on whole value.
`key-equal`	Equality match on keywords.
`key-prefix`	Prefix match on keywords.
`key-synabb`	Synonym match on keywords.
`key-phonetic`	Phonetic match on keywords.
`key-typing`	Typing correction match on keywords.
`key-suffix`	Enables suffix stripping when forming keywords.
`key-phonetic-mandarin`	Phonetically matches Mandarin using *Hanyu Pinyin*, a system for representing the pronunciation of Mandarin Chinese through a set of phonetic codes. Each phonetic code is expressed as a word in the Roman (Latin) alphabet. A number of Chinese characters can have the same phonetic code, and are treated as synonyms by this approximate match mechanism. The mechanism also recognises the phonetic code and matches it to the Chinese characters with the associated pronunciation.

Generally, an attribute on which approximate matching is enabled has several of these flags set.

If the attribute type is a string type and has `nk-abbrev` enabled, the DSA automatically generates up to two abbreviations for each value. These abbreviations then become additional synonyms to the ones explicitly added for the attribute type.

The automatically generated abbreviations are the first letters of each word in the value, with and without noise words. However, if the attribute value ends in a word in square brackets, then that word is the abbreviation. For example:

- The value `Sales and Marketing Division` generates the abbreviations `SAMD` and `SM` (assuming `and` and `Division` are noise words).
- The value `Business Planning [BusPlan]` generates the abbreviation `BusPlan`.

valueFileSuffix

`valueFileSuffix` allows you to specify that the value of an attribute should be written to a separate file when the DSA receives a dump command. It only applies to attribute types with the syntax OCTET STRING or BIT STRING.

If `valueFileSuffix` is present, the DSA dumps attributes of the attribute type using the *from path* syntax for the generated entry command with the attribute value itself written to a file.

File names have the following format:

`valNNNNN-MMM[valFileSuffix]`

Where:

- `NNNNN` is the number of the associated `dib` file (dump file).
- `MMM` is an integer assigned by the DSA to identify distinct values associated with the same `dib` file (it starts at 000 and increments as necessary).
- `valueFileSuffix` is the file extension (a typical value might be `jpg`)

To maintain security, the `userPassword` attribute cannot have a `valueFileSuffix` specified.

hashValues

`hashValues` indicates whether an attribute's values should be hashed. If set to `TRUE`, a value of `hashAlgorithm` must also be specified.

When an attribute with this extension is modified, its value is converted from a clear-text format to a hashed format. This operation cannot be reversed. When storing hashed values, it is inappropriate to have indexing or approximate matching enabled. Therefore, ViewDS disallows any attempt to enable either.

ViewDS allows hashing to be enabled for the following syntaxes:

- OCTET STRING
- IA5String
- UTF8String
- VisibleString

hashAlgorithm

This specifies the hash algorithm that values of an attribute are stored and returned in. The following are the supported algorithms with the UTF8String values that can be assigned to the `hashAlgorithm` field.

- `x-hashed-crypt` (Unix CRYPT)
- `x-hashed-md5` (MD5)
- `x-hashed-sha` (SHA)
- `x-hashed-sha1` (SHA1)
- `x-hashed-ssha` (SSHA)

Unix CRYPT is only available when ViewDS is running on a UNIX operating system. The Unix CRYPT algorithm should only be used for password values that are typically under 10 characters long.

returnHash

This specifies whether search and read operations will return the value of an attribute. If set to `TRUE`, a value of `hashAlgorithm` must be set.

This extension has three possible settings:

- `TRUE` – the values returned for the attribute are in the hash format specified by `hashAlgorithm`. If the value is not in the specified format, it is not returned.
- `FALSE` – values are never returned for the attribute.
- no value – if the attribute is `userPassword`, nothing is returned unless the `passwordEncryption` operational attribute (see page 117) has been set.

returnTagged

This indicates whether the returned hashed values are prefixed with a special tag to identify the hash algorithm used.

The following tags identify the hash algorithms:

- `{crypt}` – Unix CRYPT
- `{MD5}` – MD5
- `{SHA}` – SHA
- `{SHA}` – SHA1
- `{SSHA}` - SSHA

This extension has no effect unless `returnHash` is `TRUE`.

deleteValuesReferencingDeleted

If set to `TRUE`, the DSA deletes a DN value when the entry it refers to is deleted. This extension only applies if `dnTracking` is enabled.

deleteValuesReferencingMoved

If set to `TRUE`, the DSA deletes a DN value when the entry it refers to is moved to a new superior. This extension only applies if `dnTracking` is enabled.

matchQuality

This allows a bias to be set for an attribute that influences its match-quality score returned by `viewDSMatchQuality` (see the *ViewDS Technical Reference Guide: User Interfaces*).

The range of the bias is from -100 to 100 where a negative value will improve the match-quality score of the associated attribute, giving a ‘better’ match. A positive value will decrease the match-quality score, giving a ‘worse’’ match.

The score returned by `viewDSMatchQuality` for all attributes is normalized within the range of 0 to 100. Therefore, an attribute with a bias of -99, for example, will always return a score between 0 and 100.

Examples

The following Stream DUA script reads all values of the `attributeTypeExtensions` attribute. It is directed a schema configuration subentry, which is where the `attributeTypeExtensions` attribute is normally held.

```
read {
      organizationName “Deltawing”
      / commonName “Schema Configuration”
}
return { attributeTypeExtensions };
```

The following script adds full equality indexing, and prefix and phonetic approximate matching to the built-in attribute `surname`.

```
modify {}
with changes {
      add values attributeTypeExtensions {
          identifier surname,
          indexing in-full,
          approxMatchType {
              nk-prefix,
```

```
            nk-phonetic
        }
    }};
```

The following script adds value hashing to the `userPassword` attribute. It will allow values of `userPassword` to be returned as MD5 hashes only.

```
modify { }
with changes {
    add value attributeTypeExtensions {
        identifier userPassword,
        hashValues TRUE,
        hashAlgorithm "x-hashed-md5",
        returnHash TRUE,
        returnTagged FALSE
    }};
```

entryIndexing

This operational attribute specifies the kinds of entry index to build for an attribute.

The ASN.1 definition is:

```
entryIndexing ATTRIBUTE ::= {
    WITH SYNTAX                IndexDescription
    EQUALITY MATCHING RULE     indexDescriptionMatch
    USAGE                      dSAOperation
    ID                                 {vf 21 1} }

IndexDescription ::= SEQUENCE {
    type        [0] AttributeType,
    component       ComponentReference OPTIONAL,
    path        [3] ComponentPath OPTIONAL,
    index       [1] IndexKind }
    (WITH COMPONENTS { …, component ABSENT }  |
        WITH COMPONENTS { …, path ABSENT } )

ComponentReference ::= UTF8String

ComponentPath ::= Markup

    (CONSTRAINED BY

        { -- component path expression -- })
```

The `IndexDescription` allows you to declare an index for an entire attribute value or an index for a component of an attribute value. Both options are described below.

Index entire attribute

If neither the `component` nor `path` fields are declared (the typical case) then `index` specifies an index of `IndexKind` for the *entire value* of the attribute type.

To illustrate, the following Stream DUA script adds a `case-ignore-string` index to support efficient equality and prefix matching on the `commonName` attribute:

```
modify { }
with changes {
    add value entryIndexing {
        type commonName,
        index case-ignore-string
    }
};
```

Index a component of an attribute

If either the `component` or `path` fields are declared, then `index` specifies an index of `IndexKind` for a *component value* of the attribute type. (Only one of these fields can be present in an `IndexDescription`.)

component

If the `component` field is present:

- `ComponentReference` identifies a component value of an attribute type with a syntax defined in ASN.1; and
- `index` identifies an `indexKind` to be applied to the above component value.

The format for `ComponentReference` strings is described in *RFC 3687*.

The `index` must be an `IndexKind` applicable to the data type of the identified component part (rather than to the attribute's syntax as a whole).

The following Stream DUA script adds an `object-identifier` index to support equality matching on the `objectClasses` attribute.

```
modify { }
with changes {
      add value entryIndexing {
          type objectClasses,
          component "identifier",
          index object-identifier
      }
};
```

Note that the equality matching rule for the `objectClasses` attribute is `objectIdentifierFirstComponentMatch` and that the first component of an `objectClasses` attribute value is an `OBJECT IDENTIFIER` named `identifier`.

path

If the `path` field is present:

- `ComponentPath` identifies a component value of an attribute type with a syntax defined in XML Schema or ASN.1, or defined by a DTD; and
- `index` identifies an `indexKind` to be applied to the above component value.

The format for `ComponentPath`, which is based on XPath notation, is described in *draft-legg-xed-matching-xx.txt* (available from http://xmled.info).

The `index` must be an `IndexKind` applicable to the data type of the nominated component part (rather than to the attribute's syntax as a whole).

rdnIndexing

This operational attribute specifies the kinds of RDN index to build for an attribute type. An RDN index supports `dnAttributes` matches on values that occur in DNs, but occur more frequently in entries as non-naming attributes (for example, `locality`). A `dnAttributes` match is a match for an attribute that occurs in an entry or in its DN.

An RDN index is redundant if a DN index is specified for the same attribute type. However, it results in a smaller index and does not affect the performance of move and rename operations.

The ASN.1 definition is:

```
rdnIndexing ATTRIBUTE    ::= {
      WITH SYNTAX                IndexDescription
      EQUALITY MATCHING RULE     indexDescriptionMatch
      USAGE                      dSAOperation
      ID                         {vf 21 2} }
```

dnIndexing

This operational attribute specifies the kinds of DN index to build for an attribute type.

A DN index supports `dnAttributes` matches. Every entry is indexed on the RDN values of itself and its superiors.

This index results in much faster `dnAttributes` matches, but makes move and rename operations more expensive. Use only for large databases in which the non-leaf entry structure is relatively static (for example, White Pages).

If a DN or collective index is specified, then base-object indexing (without aliases) will occur regardless of the presence of the `baseObjectIndexing` attribute.

The ASN.1 definition is:

```
dnIndexing ATTRIBUTE ::= {
      WITH SYNTAX                IndexDescription
      EQUALITY MATCHING RULE     indexDescriptionMatch
      USAGE                      dSAOperation
      ID                         {vf 21 4}
}
```

collectiveIndexing

This operational attribute specifies the kinds of collective index to build for an attribute type.

A collective index supports collective-attribute matches (matches that find all entries that *appear* to have a particular collective attribute). Use this instead of an entry index for collective attributes (an entry index for a collective attribute will index only the subentries that actually hold the attribute). A collective index makes move and rename operations more expensive.

If a collective index is specified, then base-object indexing (without aliases) will occur regardless of the presence of the `baseObjectIndexing` attribute.

The ASN.1 definition is:

```
collectiveIndexing ATTRIBUTE::= {
      WITH SYNTAX                IndexDescription
      EQUALITY MATCHING RULE     indexDescriptionMatch
      USAGE                      dSAOperation
      ID                         {vf 21 5}
}
```

baseObjectIndexing

This operational attribute causes enumerated subtree lists to be maintained for every entry. In the absence of an appropriate DN or collective indexes, this removes the need for tree traversals when the DSA evaluates `dnAttributes` matches or collective-attribute filter items.

The ASN.1 type definition is:

```
baseObjectIndexing ATTRIBUTE::= {
      WITH SYNTAX               BaseObjectIndexing
      EQUALITY MATCHING RULE    integerMatch
      USAGE                     dSAOperation
      ID                        vf 21 3}
BaseObjectIndexing ::= ENUMERATED {
      withoutAliases (0), withAliases (1)
}
```

The attribute is multi-valued with up to two values.

Enabling `baseObjectIndexing` speeds up a search request that includes a base object – it limits the scope of a search to an area of the full DIT. For example, consider a large database in which entries are organized by locality – depending on the search criteria, all entries outside a particular locality could be excluded from further evaluation.

The value `withoutAliases` maintains subtree lists in which aliased entries are excluded, and supports searches that do not specify alias dereferencing in the search subtree. The value `withAliases` maintains subtree lists in which aliased entries are included, and supports searches that specify alias dereferencing in the search subtree.

These indexes may make move operations slightly more expensive.

indexingDisabled

This operational attribute disables indexing temporarily – this is useful when, for example, initially loading the database.

The ASN.1 type definition is:

```
indexingDisabled ATTRIBUTE  ::= {
      WITH SYNTAX               NULL
      SINGLE VALUE              TRUE
      USAGE                     dSAOperation
      ID                        {vf 21 6}
}
```

If this attribute is added to a schema configuration subentry, all indexes in the same information plane are disabled. The indexing operational attributes are not altered, and they can still be modified, but any such modifications will have no immediate effect. When the `indexingDisabled` attribute is removed, all indexes are rebuilt.

The Stream DUA operation `empty for filling` adds `indexingDisabled` at the beginning of a load using ViewDS Fast Load (see page 11). The operation `fill` removes `indexingDisabled` at the end of a fast load, which results in automatic building of the indexes. This is the fastest way to load a database.

automaticIndexing

This operational attribute is a single-valued Boolean. If set to `TRUE`, automatic indexing is enabled. If set to `FALSE` – or absent from the schema configuration subentry – automatic indexing is disabled.

The ASN.1 type definition is:

```
automaticIndexing ATTRIBUTE ::= {
```

```
    WITH SYNTAX              BOOLEAN
    EQUALITY MATCHING RULE   booleanMatch
    SINGLE VALUE             TRUE
    USAGE                    dSAOperation
    ID                       id-adacel-soa-automaticIndexing
}
```

When this operational attribute is added to the schema configuration subentry and set to `TRUE`, equality indexes are created for all attribute types that have a description in the associated subschema subentry.

When you use either ViewDS Fast Load or Stream DUA to load an LDIF content record for a root entry, two operations occur. First, the database is emptied, and then the root entry is initialized to the contents of the LDIF content record. Automatic indexing is enabled, unless the LDIF content record includes the `automaticIndexing` attribute set to `FALSE`.

When using the LDIF dump format, the DSA ensures consistent behaviour between a dump and reload. When the DSA performs an LDIF dump, it ensures that the `automaticIndexing` attribute is in the content record it creates for the root entry. If this attribute is absent from the current root entry, the dumped value is set to `FALSE`.

Word lists

To improve approximate matching, the DSA allows you to define synonyms, noise words and truncated words for an attribute (or a component of an attribute) with a string type.

Synonyms are a set of words treated as equivalent when a user requests an approximate match on one of the words in the set. For example, a set of synonyms might be ‘high school’ and ‘secondary college’.

There are two kinds of synonyms:

- Non-keyword synonyms
  A set of non-keyword synonyms comprises phrases of one or more words. Each entire phrase in the set is equivalent. So, for the above example set of synonyms, a search on ‘high school’ would return matches on both ‘high school’ and ‘secondary college’.
- Keyword synonyms
  A set of keyword synonyms comprises phrases of one or more words. Each phrase in the set is equivalent, and is also equivalent to any other phrase they appear in. To illustrate with the above example, a search on ‘stevenville high school’ would match ‘stevenville secondary college’; and a search on ‘high school’ would also match ‘stevenville secondary college’.

Synonyms only apply if the `approximateMatchType` defined for the attribute type specifies non-keyword synonyms (`nk-synonym`) or keyword synonyms (`key-synonym`).

When an attribute type has a set of synonyms declared, the synonyms apply to all values of that type stored by the DSA. By default, sets of synonyms are included in the information a DSA replicates to another DSA.

Noise words are keywords to be ignored when keyword-matching an attribute value and when forming abbreviations. They are also omitted when the DSA constructs the

value of a `hierarchyName` attribute (see *Technical Reference Guide: User Interfaces*).

The special noise word '0' (digit 0) indicates that *all* numbers in an attribute value should be treated as noise words.

Truncated words are preferred truncations for keywords to be used by the DSA when building the `hierarchyName` attribute.

Operational attributes

Synonyms, noise words and truncated words are declared in operational attributes stored in a subschema subentry, and apply to all subschema areas in the same information plane (see *Concepts* on page 83).

ViewDSSynonyms

This operational attribute declares synonyms.

```
viewDSSynonyms ATTRIBUTE ::= {
      WITH SYNTAX ComponentSynonymList
      EQUALITY MATCHING RULE directoryComponentsMatch
      USAGE directoryOperation
      ID { 1 3 6 1 4 1 21473 5 21 0 }
}

ComponentSynonymList ::= SEQUENCE {
      type    [0]    AttributeType,
      path    [1]    ComponentPath OPTIONAL,
      synonyms [2]   SEQUENCE OF synonym UTF8String
}
```

If the `path` field is present:

- `ComponentPath` identifies a component value of an attribute type with a syntax defined in XML Schema or ASN.1, or defined by a DTD; and
- `synonyms` identifies a sequence of synonyms that apply to the component value.

The format for `ComponentPath`, which is based on XPath notation, is described in *draft-legg-xed-matching-xx.txt* (available from http://xmled.info).

ViewDSCombinedSynonyms

This is a read-only operational attribute that lists an entry's synonyms. It is constructed when an entry is read and cannot be modified.

```
viewDSCombinedSynonyms ATTRIBUTE ::= {
      WITH SYNTAX ComponentSynonymList
      EQUALITY MATCHING RULE directoryComponentsMatch
      USAGE dSAOperation
      ID { 1 3 6 1 4 1 21473 5 21 1 } }
```

ViewDSNoiseWords

This operational attribute declares noise words.

```
viewDSNoiseWords ATTRIBUTE ::= {
      WITH SYNTAX ComponentWordList
      EQUALITY MATCHING RULE objectIdentifierFirstComponentMatch
      USAGE directoryOperation
      ID { 1 3 6 1 4 1 21473 5 21 2 }
```

```
}
ComponentWordList ::= SEQUENCE {
      type [0] AttributeType,
      path [1] ComponentPath OPTIONAL,
      words [2] SEQUENCE OF word UTF8String
}
```

If the `path` field is present:

- `ComponentPath` identifies a component value of an attribute type with a syntax defined in XML Schema or ASN.1, or defined by a DTD; and
- `words` declares a sequence of noise words that apply to the component value.

The format for `ComponentPath`, which is based on XPath notation, is described in *draft-legg-xed-matching-xx.txt* (available from http://xmled.info).

ViewDSCombinedNoiseWords

This is a read-only attribute that lists an entry's noise words. It is constructed when an entry is read and cannot be modified.

```
viewDSCombinedNoiseWords ATTRIBUTE ::= {
      WITH SYNTAX ComponentWordList
      EQUALITY MATCHING RULE objectIdentifierFirstComponentMatch
      USAGE dSAOperation
      ID { 1 3 6 1 4 1 21473 5 21 3 }
}
```

ViewDSTruncatedWords

This operational attribute declares truncated words.

```
viewDSTruncatedWords ATTRIBUTE ::= {
      WITH SYNTAX ComponentWordList
      EQUALITY MATCHING RULE objectIdentifierFirstComponentMatch
      USAGE directoryOperation
      ID { 1 3 6 1 4 1 21473 5 21 4 } }
ComponentWordList ::= SEQUENCE {
      type [0] AttributeType,
      path [1] ComponentPath OPTIONAL,
      words [2] SEQUENCE OF word UTF8String }
```

If the `path` field is present:

- `ComponentPath` identifies a component value of an attribute type with a syntax defined in XML Schema or ASN.1, or defined by a DTD; and
- `words` declares a sequence of truncated words that apply to the component value.

The format for `ComponentPath`, which is based on XPath notation, is described in *draft-legg-xed-matching-xx.txt* (available from http://xmled.info).

ViewDSCombinedTruncatedWords

This is a read-only attribute that lists an entry's truncated words. It is constructed when an entry is read and cannot be modified.

```
viewDSCombinedTruncatedWords ATTRIBUTE ::= {
      WITH SYNTAX ComponentWordList
      EQUALITY MATCHING RULE objectIdentifierFirstComponentMatch
      USAGE dSAOperation
      ID { 1 3 6 1 4 1 21473 5 21 5 }
}
```

Chapter 6
Managing security

This chapter provides an overview of ViewDS security. It describes how configure ViewDS to authenticate users and to control their access to a directory. The chapter also describes LDAP password management along with other miscellaneous aspects of ViewDS security.

It has the following sections:

- Authentication
- Access control
- LDAP password management
- Miscellaneous security topics

Authentication

This sections describes the following:

- Overview of authentication
- Strong authentication
- Authentication attributes
- Simple Authentication and Security Layer (SASL)

Overview of authentication

The X.500 security model makes a distinction between *authentication* and *access control*.

To access a *Directory System Agent's* (DSA's) directory, a *Directory User Agent* (DUA), another DSA or the ViewDS Management Agent must first authenticate to the DSA using a bind request. When making a bind request, the requestor presents *credentials*, which serve to *authenticate* its identity. The recipient DSA responds by signalling that it accepts or rejects the bind request. If it accepts, the DSA may also return its own credentials to the requestor.

Once authenticated, the current connection needs no further authentication in order to access directory information. Access is granted according to the requestor's *access rights*, which are determined by the *access control scheme* implemented.

Levels of authentication

X.500 provides different levels of authentication depending on the credentials in a bind request. ViewDS accepts three kinds of credentials: *none*, *simple* (using unprotected password) and *strong*.

None (no authentication)

None (no authentication) is only available if enabled by the `anonymousPrivilege` operational attribute in the DSA's root entry. This attribute can be set up as required, but should normally be present to allow Access Presence to perform the GetMyDN procedure (see page 111).

Simple authentication

The DSA verifies that the password is correct for the user name supplied in the credentials by checking either the:

- `userPassword` attribute in the user's entry (for a DUA bind); or
- `dsaCollaborators` operational attribute in the root entry (for a DSA bind).

These attributes are described under *Authentication attributes* on page 115.

The DSA returns simple credentials to the bind initiator as acceptance of the bind. The name in the credentials is the name stored in the `myAccessPoint` attribute of the root entry (see page 159). A password is also returned if the bind initiator is a DSA; and there is a `myPassword` component in the `dsaCollaborators` attribute for the other DSA.

Strong authentication

Strong authentication is available if the DSA has either a `cACertificate` attribute in its root entry, or a certificate in one of its explicit trust stores.

The values of `cACertificate` are the certificates of the Certification Authorities (CAs) that the DSA trusts (that is, its *trust anchors*). For the DSA to support two-way strong authentication it must be configured with its own private key and certificate.

For explicit trust, the DSA performs a comparison against a set of trusted certificates in its trust stores. The trust stores are:

- the DSA's trusted directory (identified by the configuration-file parameter `dsatrusted`) – this is used for strong authentication between DSA and Remote Administration Service (RAS), and between users of the ViewDS Management Agent and the DSA and RAS.
- the `certificate` field in `dsaCollaborators` – this is used for strong authentication between DSAs.

Strong authentication is discussed further on page 112.

Super-administrator

A user with *super-administrator* privileges can perform operations that bypass the access controls defined by the DSA's *access control scheme* (see page *Access control* on page 121). They can also perform additional operations (such as dump, checkpoint, verify, etc.) that are unavailable with any other privilege.

The super-administrator privilege can be obtained through either simple authentication using the *deity* password, or strong authentication using explicit trust. Both these options are outlined below.

Simple authentication using the deity password

The DSA assigns the super-administrator privilege to a user who authenticates with the following credentials:

- the Distinguished Name (DN) of the root entry (an empty RDN sequence); and

- a special password which is a random string created by DSA at start-up.

The password is held in the file specified by the configuration-file parameter `admpasswd` (usually `${VFHOME}/general/deity`). Normally, only the owner of the DSA process can read this file.

Strong authentication using explicit trust

Alternatively, a super-administrator can be an identity that authenticates using strong authentication and has a certificate in the DSA's trusted directory. This form of strong authentication is used between the ViewDS Management Agent, DSA and RAS. It is not intended for other users. (Strong authentication is discussed further on page 112.)

GetMyDN procedure

A user must supply a DN and password when binding. However, as a DN is long and unwieldy, most users would be unwilling to type theirs in when connecting to a DUA. Therefore, Access Presence supports a procedure that allows a user to obtain their DN. The procedure, *GetMyDN*, is invoked transparently when a user logs into Access Presence.

With GetMyDN, a user simply enters a user name and password, and the DUA binds to the DSA with simple credentials consisting of an empty DN without a password. The DUA then invokes a search operation for all entries below a supplied base-entry. This search filter comprises:

- an equality match for the ViewDS-specific attribute `viewDSUserName`; and
- an equality match for the standard X.500 attribute `userPassword`.

The search must not request the return of attribute information. Such a search is recognized as a GetMyDN search and ViewDS implements special security provisions for it.

If the DUA finds that exactly one entry matches the search criteria, it knows that the search result contains the user's DN. The DUA then unbinds, and rebinds using that DN and the password entered by the user. The DUA's cache the returned DN so that this procedure is needed only if the normal bind fails.

For a GetMyDN search to be allowed, access controls must be set up appropriately:

- If *ViewDS Access Control* is used (see page 121), the `anonymousPrivilege` attribute must be set up to allow anonymous binds for the initial bind operation to be accepted. (See page 127 for a description of the `anonymousPrivilege` attribute and an appropriate Stream DUA script.)
- If *Basic Access Control* or *Simplified Access Control* is used (see page 121), there should be ACI items granting `allUsers` a `filterMatch` on `viewDSUserName` and `userPassword`. Note that even if such an ACI item is present, ViewDS will not permit searches on `userPassword` for anything other than GetMyDN searches.

The success of the GetMyDN procedure depends on the combination of user name and password being unique. Access Presence checks that this uniqueness will not be compromised prior to accepting a request to change a user name or password. However, uniqueness is not guaranteed when the subtree below the base object is distributed across DSAs or when non-ViewDS DUAs are used to change user names or passwords.

Strong authentication

Trusted CAs

Strong authentication requires the DSA to maintain a list of the Certification Authorities (CAs) and their public keys, which it ultimately trusts in regard to the user and CA certificates they issue. This list of CAs is stored as values of the standard X.509 attribute `cACertificate` in the root entry of the DSA.

To add a CA to the list of trusted CAs, use Stream DUA with the `-d` option (to allow access to the root entry), and add a value of `cACertificate` to the root entry. A new CA can also be added through the ViewDS Management Agent.

Explicit trust

ViewDS also implements strong authentication based on explicit comparison with a set of trusted certificates. This is used to provide:

- two-way trust between the DSA and RAS;
- one-way trust between a ViewDS Management Agent user and DSA; and
- one-way trust between a ViewDS Management Agent user and RAS.

Verification is through explicit trust of individual certificates in the following directories:

- the DSA's trusted directory (identified by the configuration-file parameter `dsatrusted` – by default, `${VFHOME}/setup/trusted`); and
- the RAS's trusted directory (identified by then configuration-file parameter `rastrusted` – by default, `${VFHOME}/setup/trusted`).

Entities authenticated in this way are granted the super-administrator privilege (see page 110). Explicit trust should not be used to set up credentials for normal users.

Public–private key pairs

The DSA, RAS and ViewDS Management Agent require public-private key pairs.

DSA key pair

For the DSA to authenticate itself strongly to a DUA (a two-way bind), it needs its own key pair represented as a certificate and private key. The certificate is declared in the configuration-file parameter `dsacertificate` (by default, `${VFHOME}/setup/dsa.cer`).

The DSA's user certificate must be created by a CA. It must also bind the DSA's name – stored in its `myAccessPoint` attribute (see page 159) – to the public key of the DSA's public–private key pair. ViewDS does not include a CA, or functionality to generate a public–private key pair. Your ViewDS vendor can, however, help you obtain software for both purposes.

The DSA also needs a private key. The private key can be made available to the DSA in either a PKCS#8 format file (defined in the *PKCS #8 RSA Encryption Standard, Nov. 1993*) or a PKCS#12 format file (*PKCS #12 Personal Information Exchange Syntax, June 1999*).

For PKCS#8 file format, the private key can be declared as a BER encoded value of either of the following ASN.1 types:

- `PrivateKeyInfo` – the value is stored as clear-text in the file specified by the configuration-file parameter `dsaprivkey` (by default, `${VFHOME}/setup/dsa.pk8`). The file should be made read-only to the owner of the DSA process.

- `EncryptedPrivateKeyInfo` – the value is stored in the file specified by the configuration-file parameter `dsaprivkey`. The encryption password is stored in the file specified by the configuration-file parameter `dsaprivpass` (by default, `${VFHOME}/general/keyaccess`). If the `keyaccess` file is missing or cannot be accessed, the format of the `dsa.pk8` file is assumed to be clear-text.

Alternatively, the configuration-file parameters `dsacertificate` and `dsaprivkey` can be directed to a PKCS#12 file. This allows the DSA to obtain both its public- and private-key information from the same file.

A PKCS#12 file may be encrypted. In this case, the passphrase used to protect the content of the file must be provided in the file identified by the configuration-file parameter `dsaprivpass`.

It is also possible to configure the DSA to obtain the private key from a PKCS#12 file and the certificate from a separate BER encoded certificate file.

RAS key pair

For the RAS to authenticate itself strongly to a DSA (a two-way bind), it needs a user certificate. The certificate is declared in the configuration-file parameter `rascertificate` (by default, `${VFHOME}/setup/ras.cer`).

The RAS also needs a private key, which can be made available to it in either a PKCS#8 or a PKCS#12 format file.

For PKCS#8 file format, the RAS's private key can be declared as a BER encoded value of either of the following ASN.1 types:

- `PrivateKeyInfo` – the value is stored as clear-text in the file specified by the configuration-file parameter `rasprivkey` (by default, `${VFHOME}/setup/ras.pk8`). The file should be made read-only to the owner of the RAS process.
- `EncryptedPrivateKeyInfo` – the value is stored in the file specified by the configuration-file parameter `rasprivkey`. The encryption password is stored in the file specified by the configuration-file parameter `rasprivpass` (by default, `${VFHOME}/general/keyaccess`). If the `keyaccess` file is missing or cannot be accessed, the format of the `ras.pk8` file is assumed to be clear-text.

Alternatively, the `rascertificate` and `rasprivkey` configuration-file parameters can be directed to a PKCS#12 file.

If the PKCS#12 is encrypted, the passphrase must be provided in the file identified by the configuration-file parameter `rasprivpass`.

It is also possible to configure the RAS to obtain the private key from a PKCS#12 file and the certificate from a separate BER encoded certificate file.

ViewDS Management Agent key pair

Each user of the ViewDS Management Agent must have a key pair. Otherwise, the ViewDS Management Agent will not be able to connect to either the DSA or RAS.

This key pair must be in the certificate store of the user's desktop. The user's certificate must also be exported from the certificate store and placed in the trusted directory of the DSA and RAS.

Other authentication requirements

For strong authentication to succeed, the user must either:

- supply their user certificate in the bind credentials;
- have a directory entry that contains a user certificate whose subject name matches their bind name; or
- have their user certificate in the DSA's trusted directory. In this case, there is no further validation, the identity for access control is the DN in the certificate, and the authenticated session has the super-administrator privilege.

For the first two options, the user certificate must be certified by one of the DSA's trusted CAs, either directly or indirectly. If it is not directly certified, the DSA will attempt to construct a certification path from the CA that signed the certificate to one of its trusted CAs. The certificate for an intermediate CA must be stored in its DIT entry in either the `cACertificate` or `crossCertificatePair` attribute.

When the configuration-file parameter `certrevocation` (by default, `on`) is set to `off`, the DSA can use the certificates supplied in the `CACertificates` parameter of the bind argument when forming the certification path.

Locating the user's and intermediate CA's certificates

When the user has a directory entry containing a user certificate whose subject name matches their bind name, the configuration-file parameter `certificatelookup` specifies how the DSA locates the user's and intermediate CAs' certificates.

The parameter is set to either: `SubjectNameIsEntryName` or `MapSubjectNameToEntryName`. Each option is described below.

`SubjectNameIsEntryName`

When `certificatelookup` is set to `subjectNameIsEntryName`, the DSA finds a certificate in the directory using the subject name. It reads the entry with the DN matching the subject name of the certificate and looks for either:

- the end-entity certificate in the `userCertificate` attribute; or
- intermediate CA certificates in the `cACertificate` attribute or `crossCertificatePair` attribute.

This option requires certificates to be stored in a directory entry whose DN matches the subject name in the certificate.

This is not always practical as many CAs use naming policies that are not compatible with the naming policy of the directory. There is also the problem that DNs in the directory can often be changed to reflect organisational changes. However, subject names in certificates are not expected to change very often – when they do change, the certificate must be re-signed by the CA.

`MapSubjectNameToEntryName`

When `certificatelookup` is set to `mapSubjectNameToEntryName`, the DSA searches the entire DIT for an entry containing the required certificate with the matching subject name. This provides support if your CA naming policy does not match your directory naming policy.

The DSA then uses the entry containing the matching certificate, in the case of the end-entity certificate, as the authentication

identity for subsequent processing of operations on the authenticated connection.

If the required certificate is stored in more than one entry, the authentication request fails. This is because the DSA cannot determine which is the valid authentication identity to use in subsequent processing.

The `mapSubjectNameToEntryName` option provides greater flexibility in terms of where certificates can be stored, but it is less efficient. Part of this efficiency problem is addressed by the DSA automatically indexing the subject name of certificates in the `userCertificate`, `cACertificate` and `crossCertificatePair` attributes. These indexes are used during certificate lookup.

Authentication attributes

ViewDSUserName

This attribute is a simple text string that associates a user name with a user.

```
ViewDSUserName ATTRIBUTE ::= {
      SUBTYPE OF     name
      WITH SYNTAX    DirectoryString {64}
      USAGE          directoryOperation
      ID                   {vf 18 2} }
```

By convention, a user name is formed by concatenating the first letter of the user's `givenName` with their `surname`, and then truncating the result to seven characters.

Access Presence uses the GetMyDN procedure to look up the DN for the user to use in the authentication process. However, as user names are not unique, each combination of user name and password must be unique for GetMyDN to succeed.

NOTE: `userName` and `view500UserName` are alternative names for this attribute.

userPassword

This multi-valued attribute is an octet string that holds the plain-text password supplied by the user in a bind operation.

The attribute's definition is as follows:

```
userPassword ATTRIBUTE  ::= {
      WITH SYNTAX           OCTET STRING
      EQUALITY MATCHING RULE octetStringMatch
      ID                              {ds 4 35} }
```

The DSA implements special behaviour for the `userPassword` attribute:

- No user can read a clear-text value of `userPassword` (including the super-administrator using Stream DUA). However, if `passwordEncryption` (see page 117) or a value-hashing scheme is defined, then values can be read in encrypted form.
- A value is never dumped or recorded in logs in clear-text form. It is always dumped or recorded in encrypted form using the `key` set through the DSA Controller or the default key. For security reasons, the `userPassword` value will not be dumped to

a file via the `valueFileSuffix` mechanism (see the definition of the `attributeTypeExtension` on page 95 for more information).

- A value can be compared (using a compare operation) without restriction.
- A value can be used in evaluating a search filter only if the search filter contains an equality match for `userName` and an equality match for `userPassword` and no attribute information is requested.

The DSA also implements the following special behaviour when ViewDS Access Control applies:

- A value can only be added, deleted or replaced (deleted and added in the same operation) by the user whose DN is that of the entry containing the value.
- The whole attribute can only be added, deleted or replaced by a user with `admin` or `superuser` access.

Under X.500 Basic Access Control, the above behaviour is controlled by the access control applied to the attribute.

passwordModifyTimestamp

This attribute records the time of the last update to the `userPassword` attribute in the same entry.

```
passwordModifyTimestamp ATTRIBUTE  ::= {
      WITH SYNTAX           GeneralizedTime
      EQUALITY MATCHING RULE generalizedTimeMatch
      ORDERING MATCHING RULE generalizedTimeOrderingMatch
      SINGLE VALUE          TRUE
      NO USER MODIFICATION  TRUE
      USAGE                 directoryOperation
      ID                            {vf 18 15} }
```

This attribute is used by the DSA in conjunction with `passwordExpiryDays` to enforce expiry of user passwords.

passwordExpiryDays

This attribute is an integer that specifies the number of days until the last update to a `userPassword` value expires for a user. (The time of the last update is recorded by `passwordModifyTimestamp`.)

```
passwordExpiryDays ATTRIBUTE::= {
      WITH SYNTAX           INTEGER
      EQUALITY MATCHING RULE integerMatch
      SINGLE VALUE          TRUE
      USAGE                 directoryOperation
      ID                            {vf 18 16} }
```

The attribute is only permitted in the root entry. If it does not exist then passwords do not expire. The expiry time can be changed at any time, and the new value takes effect immediately.

Password expiry is checked at bind time for each user. If the user's password has expired, their access rights are reduced to the access rights granted to anonymous binds (except the user can modify their password). Normal access rights are restored when the user modifies their password – a rebind is not required.

Directory clients can read this attribute from the root entry and read `passwordModifyTimestamp` from the user's entry (subject to access controls) to determine when password expiry is imminent.

passwordEncryption

This attribute controls whether password values are returned by the DSA, and if so, how the returned values are encrypted.

```
passwordEncryption ATTRIBUTE::= {
      WITH SYNTAX           PasswordEncryption
      EQUALITY MATCHING RULE integerMatch
      SINGLE VALUE          TRUE
      USAGE                 directoryOperation
      ID                            {vf 18 17} }
PasswordEncryption ::= ENUMERATED {
      noReturn              (0),
      cryptEncryption       (1),
      shaEncryption         (2),
      sha1Encryption        (3),
      untaggedCryptHash     (4),
      untaggedSHAHash       (5),
      untaggedSHA1Hash      (6),
      md5Hash               (7),
      untaggedMD5Hash       (8),
      untaggedSSHA          (9),
      taggedSSHA            (10) }
```

The attribute is only permitted in the root entry. If it is absent or set to `noReturn`, passwords are not returned regardless of access control settings. Any other setting allows the `userPassword` attribute to be returned subject to the access controls in the directory.

The actual value returned will be a hashed value of the password, using the appropriate hashing algorithm defined by the value of the `passwordEncryption` attribute. The *untagged* versions return the hashed value of the password; other forms place a tag in the front of the value to indicate the hashing algorithm used.

dsaCollaborators

This multi-valued attribute must be present in the root entry of a DSA before it can initiate binds to, or accept binds from, other DSAs. It holds the mutual credentials needed for the DSA to initiate and accept binds, and other information relevant to the other DSAs that the DSA communicates with.

NOTE: Note that associations to other DSAs may be kept up for long periods – changes to `dsaCollaborators` are ignored in relation to existing associations.

The DSA reads `dsaCollaborators` when it starts up. Therefore, restart the DSA for a change to this attribute to take effect.

```
dsaCollaborators ATTRIBUTE ::= {
      WITH SYNTAX           DSACollaborator
      EQUALITY MATCHING RULE dsaCollaboratorMatch
      NO USER MODIFICATION  TRUE
      USAGE                 dSAOperation
      ID                            vf 12 0 }
```

```
DSACollaborator ::= SET {
      dsa-name           [0] Name,
      password           [1] OCTET STRING OPTIONAL,
      mypassword         [2] OCTET STRING OPTIONAL,
      information        [3] DSAInformation DEFAULT {},
      privilege          [4] Privilege OPTIONAL
      snmp-url           [5] OCTET STRING (SIZE (0..255)) OPTIONAL,
      certificate        [6] SEQUENCE OF Certificate OPTIONAL,
      myName             [7] DistinguishedName OPTIONAL,
      protectPasswords   [8] OCTET STRING (SIZE (16)) OPTIONAL }

DSAInformation ::= BIT STRING {
      inLocalScope (0),
      notExtensible (1),
      bACOriginatorName (2),
      bACBindName (3),
      vf40Compatible (4) -- This option has been deprecated.
      strongAuthNotSupported (5)
      dontUseProxyAuthorization (6) }
```

The fields are described below.

`dsa-name` Holds the DN of another DSA that this DSA can bind to or accept binds from.

`password` Holds the password that another DSA must use when binding to this DSA. It must be present as the DSA cannot be configured to accept anonymous DSP binds.

`mypassword` Holds the password that this DSA uses to bind to another DSA. If absent, this DSA does not use a password. If Access Proxy is implemented, the password for the identity that Access Proxy should use for simple authentication with an End-User Certificate Repository Service (EUCRS).

`information` Holds information relating to another DSA. Defined values are as follows:

- `inLocalScope` – the DSA is considered within the 'local scope' for operations that set the localScope service control.
- `notExtensible` – the DSA does not implement the critical extensions technical corrigendum to X.500 (1988) (as is the case with QUIPU DSAs).
- `bACOriginatorName` – the DSA is trusted to authenticate its DUAs so that the `originatorName` supplied in the chaining arguments can be used as the authenticated identity for access control purposes.
- `bACBindName` – ignored if `bACOriginatorName` is set. If `bACBindName` is set, the DSA's own bind name is used as the authenticated identity for access control; if not set, the user is treated as unauthenticated.
- `vf40Compatible` – this option has been deprecated

- `strongAuthNotSupported` – the peer DSA does not support strong authentication. User/password authentication should be used..
- `dontUseProxyAuthorization` – Indicates that the Proxied Authorization LDAP Control will not be used when chaining operations to the DSA identified by the `dsaCollaborators` attribute using LDAP. See LDAP Controls in Chapter 3 for more details about this LDAP Control.

`privilege` Specifies the privilege given to users invoking chained operations through another DSA. If omitted, it defaults to `read` privilege.

`snmp-url` Specifies the URL of the DSA identified by the `dsaCollaborators` attribute. This URL is used to construct SNMP MIB Objects when ViewDS is recording interactions with peer DSAs.

`certificate` A set of certificates holding the public key(s) specified by `dsaCollaborators` that the DSA should authenticate with. Any strong authentication attempt over DSP or DISP with a token that can be verified by a certificate in this field will be accepted. Otherwise, normal strong authentication processing is used.

`myName` If Access Proxy is implemented, the DN for the identity that Access Proxy should use for simple authentication with an End-User Certificate Repository Service (EUCRS).

`protectPasswords` Indicates that passwords in replication protocol messages exchanged with the DSA identified by the `dsaCollaborators` attribute will be protected with the AES-128 key that is supplied as the field value. This feature is only supported when both peers are ViewDS 7.4 (or later) servers and it is essential that each peer be configured with the same AES-128 key. If this feature is not used, passwords will be replicated in clear text.

NOTE: The passwords in `dsaCollaborators` values are encrypted in dumps, update logs and sdua output. The whole `dsaCollaborators` value is encrypted in storage.

Example

The following Stream DUA script adds a new DSA to a DSA's list of known DSAs. The password `123456` is used by NewDSA to bind to the DSA, and `ABCDEF` by the DSA when binding to NewDSA.

```
modify
      add values dsaCollaborators {
          dsa-name rdnSequence :
          { country "AU" / organizationName "NewCorp" /
          commonName "NewDSA" },
          password "123456",
          mypassword "ABCDEF" }
```

NOTE: Although the passwords are of type OCTET STRING, Stream DUA accepts values entered as normal strings.

Simple Authentication and Security Layer (SASL)

LDAP supports Simple Authentication and Security Layer (SASL) mechanisms for authentication of users accessing the directory. ViewDS supports two SASL mechanisms for authenticating user agents:

- DIGEST-MD5
- EXTERNAL
- GSS-API (Kerberos)

The security layers defined for the DIGEST-MD5 mechanism are not supported. The use of these SASL mechanisms in LDAP is defined by *RFC2829 Authentication Methods for LDAP*.

DIGEST-MD5

The SASL DIGEST-MD5 mechanism is defined in *RFC2831 Using Digest Authentication as a SASL Mechanism*.

There are two configuration options which can be used to modify the behaviour of the DSA when a user is authenticating using the DIGEST-MD5 authentication mechanism:

- *saslrealm* is used to identify a value for the realm in the SASL interaction. The default value for this configuration option in ViewDS is `ViewDS`. It can be changed to any ASCII string.
- *saslusername* is used to identify an attribute which may be used to map the user name in the SASL credentials into a DN. ViewDS requires an authentication identity and authorization identify to be DNs. However, SASL uses arbitrary string identifiers to identify the user to be authenticated. ViewDS addresses this conflict by conducting a search for the SASL user name as a value of an attribute, where the type of the attribute is determined by this configuration option. The default attribute type used in this search is the `userName` attribute.

ViewDS also supports chaining of SASL DIGEST-MD5 authentication requests to other ViewDS DSAs that it trusts. This is done using a proprietary matching rule: `saslDigestMD5Match`.

EXTERNAL

The SASL EXTERNAL mechanism is defined in *RFC2829 Authentication Methods for LDAP*.

This mechanism requests the DSA to determine the authentication identity from an underlying security layer such as SSL/TLS. ViewDS supports this SASL mechanism when the LDAP `startTLS` extended operation has been used to establish a secure TLS session on the LDAP connection. The authentication identity is determined from the user certificate, if any, that was provided during the TLS session establishment negotiations between the server and client. This certificate is validated using the same procedures used for strong authentication in DAP, and the authentication level (for Basic Access Control purposes) is set to strong if the certificate is successfully validated.

GSS-API (Kerberos)

The GSS (Generic Security Service) API (see *RFC2222*) provides a common interface for accessing different security services, including Kerberos (see *RFC1510* and *RFC1964*).

Service Principle Name

When the DSA starts up under Windows it creates a Service Principal Name (SPN), which provides the 'Kerberos principal name' required by LDAP.

The DSA stores the SPN in Active Directory below the account object for the ViewDS service. For a LocalSystem account, the account object will be the computer object for the host of the ViewDS service.

A single host can have multiple DSAs installed provided they share the same account. However, if one DSA shuts down, its SPN will unregister and Kerberos authentication will fail.

The SPNs can be managed using `setspn.exe` in Windows Support Tools.

Implementing

To enable ViewDS to authenticate Kerberos identities:

1. Perform the appropriate task below.

Platform	Task
Solaris	Grant the ViewDS DSA read-access to `/etc/krb5/krb5.keytab`
Linux	Grant the ViewDS DSA read-access to `/etc/krb5.keytab`
Windows	No task required.

2. Set the following configuration-file parameters as required (see page 46):
   - `gssService`
   - `gssName`
   - `gssrealm`
   - `gssUserName`

Access control

This section describes the following:

- Access control schemes
- Basic Access Control
- ViewDS Access Control

Access control schemes

After a user has been authenticated, access to directory information is controlled by the user's access privileges. ViewDS supports three *access control schemes*:

- *ViewDS Access Control* – a ViewDS-specific access control scheme that predates X.500 (1993) access control and is very simple to administer. It makes use of the ViewDS-specific `privilege` attribute stored in each user's entry.

- *Basic Access Control* – the X.500 (1993) standard access control scheme. This scheme makes use of the three operational attributes `prescriptiveACI`, `entryACI` and `subentryACI` which hold access control information and are stored in subentries or entries.
- *Simplified Access Control* – this is a subset of Basic Access Control also described in X.500 (1993). It differs only in that `entryACI` and access control inner areas are not used and ignored if present.
- XACML Based Access Control – this allows fine-grained access control that conforms to the XACML Version 3.0 standard. For information about operational attributes relating to this access control scheme, see the *ViewDS Access Sentinel Installation and Reference Guide*.

The access control scheme used in any particular area of the directory is specified by the `accessControlScheme` attribute described below.

accessControlScheme

This operational attribute defines the access control scheme within an access control specific area. It is defined as:

```
accessControlScheme ATTRIBUTE       ::= {
      WITH SYNTAX            OBJECT IDENTIFIER
      EQUALITY MATCHING RULE objectIdentifierMatch
      SINGLE VALUE              TRUE
      USAGE                     directoryOperation
      ID                       {ds 24 1} }
```

If this attribute is absent or there is no subentry to define an access control specific area, the access control scheme used is ViewDS Access Control (the default access control scheme). Otherwise, the attribute must hold one of two object identifier values:

- `basicAccessControlScheme` with object identifier `{ds 28 1}`
- `simplifiedAccessControlScheme` with object identifier `{ds 28 2}`

The symbolic names are understood by Stream DUA.

The `accessControlScheme` attribute can also be stored in the root entry. It then defines the access control scheme for non-entry DSA-specific entries that are subordinate to the root and do not have a copy of the real entry's access control information.

Basic Access Control

An administrative point configured for Basic Access Control has an access control subentry. It contains operational attributes that define the access controls for the entries within the access control area.

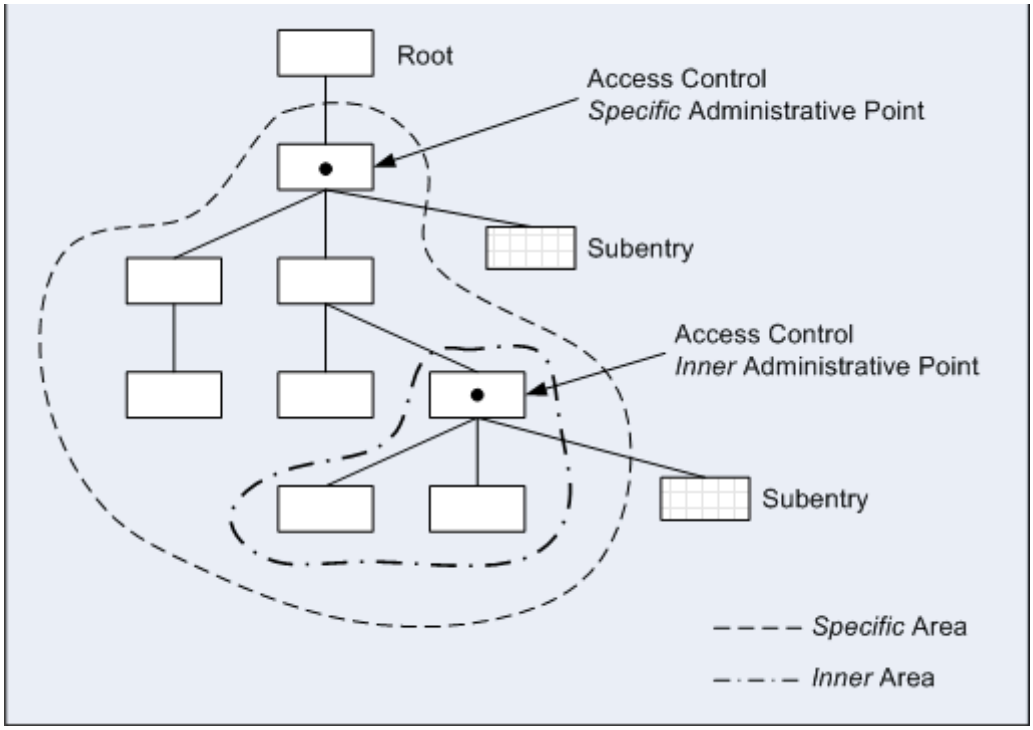


Figure 4: Administrative points and areas

With the Basic Access Control scheme, there are two kinds of access control area, specific and inner. A *specific* area might contain one or more *inner* areas. The scope of the access controls in a specific area ends at the administrative point of another specific area.

An entry is declared an inner administrative point by setting its `administrativeRole` attribute to `accessControlInnerArea`. Its subentry declares the access controls for the entries within the inner area.

Access to a subentry's entire subtree is denied for all users except the super-administrator. To grant access to other users in the subtree, the `prescriptiveACI` operational attribute should be added to the subentry, or the `subEntry` operational attribute added to the administrative point. Additionally, or alternatively, the `entryACI` operational attribute can be added to individual entries. These operational attributes are outlined below. (Their semantics are complex – for more information see *X.501 (1993) clause 16*.)

Operational attributes

The operational attributes for Basic Access Control are described below.

prescriptiveACI

This operational attribute is held in an access control subentry. It defines the access controls that apply within the subtree controlled by the subentry.

```
prescriptiveACI ATTRIBUTE       ::= {
      WITH SYNTAX           ACIItem
      EQUALITY MATCHING RULE directoryStringFirstComponentMatch
      USAGE                 directoryOperation
      ID                  {ds 24 4} }
```

The syntax of `ACIItem` is complex – for details see *X.501 (1993) clause 16.4*.

entryACI

This operational attribute is held in an entry and defines the access controls that apply to the entry or to its attribute types and values.

```
entryACI ATTRIBUTE      ::= {
      WITH SYNTAX           ACIItem
      EQUALITY MATCHING RULE directoryStringFirstComponentMatch
      USAGE                 directoryOperation
      ID                            {ds 24 5} }
```

subentryACI

This operational attribute is held in an administrative point entry. It defines the access controls that apply to immediately subordinate subentries.

```
subentryACI ATTRIBUTE       ::= {
      WITH SYNTAX           ACIItem
      EQUALITY MATCHING RULE directoryStringFirstComponentMatch
      USAGE                 directoryOperation
      ID                            {ds 24 6} }
```

userGroups

X.500 Basic Access Control allows privileges associated with an `ACIItem` to be assigned to a group of users listed in an entry of object class `groupOfNames` or `groupOfUniqueNames`. The `userGroups` operational attribute extends this functionality by allowing other object classes to fulfil this role.

The `userGroups` operational attribute should only be used in an access control administrative point. Each value of the `userGroups` operational attribute identifies an object class and one of its attributes to define a group of users. The attribute types identified should have the syntax `DistinguishedName`.

The `userGroups` operational attribute is defined as follows:

```
userGroups ATTRIBUTE ::= {
    WITH SYNTAX             UserGroup
    EQUALITY MATCHING RULE  objectIdentifierFirstComponentMatch
    USAGE                   dSAOperation
    ID                      id-adacel-oa-userGroups
}

UserGroup ::= SEQUENCE {
    objectClass         OBJECT-CLASS.&id,
    attributeType       ATTRIBUTE.&id
}
```

Role-based access control

ViewDS extends the functionality of the X.500 Basic Access Controls to provide role-based access control. The `userFilter` extends the `userClasses` component of `ACIItem` and its ASN.1 type is a DAP filter. When the filter is applied to a user's entry and evaluates to true, the user is added to `userClasses`.

A role-based access control can be time dependent – for example, a user might be granted access according to the current time and day the of the week.

The definition of the `userClasses` component is as follows.

```
UserClasses ::= SEQUENCE {
      allUsers   [0] NULL OPTIONAL,
      thisEntry  [1] NULL OPTIONAL,
      name       [2] SET SIZE (1..MAX) OF NameAndOptionalUID OPTIONAL,
      userGroup  [3] SET SIZE (1..MAX) OF NameAndOptionalUID OPTIONAL,
                 -- dn component must be the name of an
                -- entry of GroupOfUniqueNames
      subtree [4] SET SIZE (1..MAX) OF SubtreeSpecification OPTIONAL,
      superiorOfThisEntry [PRIVATE 0] NULL OPTIONAL,
      userFilter [PRIVATE 1] Filter OPTIONAL
}
```

The `Filter` type is the same as the filter in the argument to a search request:

```
Filter ::= CHOICE {
      item    [0] FilterItem,
      and     [1] SET OF Filter,
      or      [2] SET OF Filter,
      not     [3] Filter
}
FilterItem ::= CHOICE {
      equality   [0] AttributeValueAssertion,
      substrings [1] SEQUENCE {
          type ATTRIBUTE.&id({SupportedAttributes}),
          strings SEQUENCE OF CHOICE {
              initial [0] ATTRIBUTE.&Type
                          ({SupportedAttributes}{@substrings.type}),
              any     [1] ATTRIBUTE.&Type
                          ({SupportedAttributes}{@substrings.type}),
              final   [2] ATTRIBUTE.&Type
                          ({SupportedAttributes}{@substrings.type}),
              control Attribute
          },
          assertedContexts [PRIVATE 0] AssertedContexts OPTIONAL
          },
          greaterOrEqual     [2] AttributeValueAssertion,
          lessOrEqual    [3] AttributeValueAssertion,
          present            [4] AttributeType,
          approximateMatch   [5] AttributeValueAssertion,
          extensibleMatch    [6] MatchingRuleAssertion,
          contextPresent     [7] AttributeTypeAssertion
}
MatchingRuleAssertion ::= SEQUENCE {
      matchingRule   [1] SET SIZE (1..MAX) OF MATCHING-RULE.&id,
      type           [2] AttributeType OPTIONAL,
      matchValue     [3] MATCHING-RULE.&AssertionType
      ( CONSTRAINED BY {
          -- matchValue must be a value of type specified
          -- by the &AssertionType field of one of the
          -- MATCHING-RULE information objects identified
          -- by matchingRule –
```

```
      } ),
      dnAttributes   [4] BOOLEAN DEFAULT FALSE,
      assertedContexts [PRIVATE 0] AssertedContexts OPTIONAL
}
```

NOTE: The *Time and date attributes* described on page 82 are useful when declaring time-based refinements.

ViewDS Access Control

The ViewDS Access Control scheme uses two attributes: `privilege` and `anonymousPrivilege`. Each is described below.

privilege

This attribute controls the user's access privileges. It consists of an enumerated integer and an optional DN. If the attribute is absent from a user's entry, the user is given `read` access.

```
privilege ATTRIBUTE     ::= {
      WITH SYNTAX           Privilege
      EQUALITY MATCHING RULE integerFirstComponentMatch
      SINGLE VALUE             TRUE
      USAGE                    directoryOperation
      ID                      {vf 24 0} }

Privilege ::= SEQUENCE {
      accessLevel    AccessLevel,
      subtree           DistinguishedName OPTIONAL,
      assignUpdaters    BOOLEAN DEFAULT FALSE }

AccessLevel ::= ENUMERATED {
     none(0), read(1), update(2), admin(3), superuser(4),
     authenticationOnly(5), requestor(6) }
```

The fields are described below.

`accessLevel` The user's access level:

- `none` – the user has no access to the directory other than the ability to invoke a GetMyDN search.
- `read` – the user can read all attributes (except `userPassword`) from any entry in the directory, and can modify their own `userPassword`.
- `update` – the user has read access plus permission to modify any attributes (except `userPassword` and `privilege`) in any entry within a designated subtree.
- `admin` – the user has update access plus permission to modify the `userPassword` attribute in entries within the designated subtree.
- `superuser` – the user has admin access to the whole directory and can modify the `privilege` attribute.
- `authenticationOnly` – the user has no access to the directory other than to submit bind requests.
- `requestor` – the user has permission to submit update requests as part of the approval process (see the *Technical Reference Guide: User Interfaces*).

`subtree` The DN of the subtree that a user with an `accessLevel` of `update` or `admin` access can access. The user can add, delete, modify or rename entries within that subtree, and move entries within and into (but not out of) that subtree. If absent, the designated subtree is the entire DIT. If invalid, the user's access level reverts to read access.

`assignUpdaters` This flag applies to users with an `accessLevel` of `admin`. It permits them to assign `update` privilege to users within the scope of their `subtree` field, provided the `subtree` scope of the assigned `update` privilege is restricted to the `subtree` of the `admin` user.

Even though the DSA permits read or modify access to an attribute, Access Presence may restrict access. The attributes that Access Presence can read or modify are governed by the `attributePresentation` and `objectClassPresentation` attributes as well as the DSA's enforcement of `privilege`.

anonymousPrivilege

This multi-valued attribute must be present in the root entry before a DSA will accept bind requests with anonymous, or inadequate credentials, from another DSA or a DUA (including Access Presence using the GetMyDN procedure).

```
anonymousPrivilege ATTRIBUTE ::= {
      WITH SYNTAX           AnonymousPrivilege
      EQUALITY MATCHING RULE anonymousPrivilegeMatch
      NO USER MODIFICATION  TRUE
      USAGE                 dSAOperation
      ID                    {vf 12 1}
}

AnonymousPrivilege ::= SET {
      protocol       [0] BIT STRING {
                           dapProtocol(0), dapAdmProtocol(1),
                           dspProtocol(2), dopProtocol(3),
                           dispProtocol(4), ebRSProtocol(5),
                           xacmlProtocol(6), smplProtocol(7) },
      credentialType  [1] BIT STRING {
                           noCredentials(0), noNameNoPasswd(1),
                           noNameZeroPass(2), nameNoPasswd(3),
                           proxyDefault(4), saslGSSAPI(5) },
      privilege       [2] Privilege
}
```

The fields are described below.

`protocol` Specifies the protocols that allow anonymous connections:

- `dapProtocol` (Directory Access Protocol)
- `dapAdmProtocol` (DAP Admin Protocol `dapAdministrationAC`)
- `dspProtocol` (Directory System Protocol)
- `dopProtocol` (Directory Operational Binding Protocol – not currently supported)

- `dispProtocol` (Directory Information Shadowing Protocol)
- `ebRSProtocol` (ebXML Registry Service Protocol)
- `xacmlProtocol` (eXtensible Access Control Language Authorization Request Protocol)
- `spmlProtocol` (Service Provisioning Markup Language Protocol)

`credentialType` Specifies the anonymous credentials that are supported for a given protocol:

- `noCredentials` – the credentials in the Directory Bind argument are absent.
- `noNameNoPasswd` – simple credentials are present, but name is an empty RDN sequence (the name of the root) and password is absent.
- `noNameZeroPass` – simple credentials are present, but name is an empty RDN sequence and password is an empty string.
- `nameNoPasswd` – name is present and not an empty RDN sequence, and password is absent.
- `proxyDefault` – sets anonymous credentials as the default credential type for anonymous proxied users. If not set, no privileges will be given to the proxied user.
- `saslGSSAPI` – credentials are provided by a SASL authentication mechanism if the identity authenticates through the GSS-API SASL mechanism, but ViewDS cannot map them to an entry in the directory.

`privilege` Specifies the privileges given to users who bind or chain to the DSA using the indicated protocol and credentials.

An `anonymousPrivilege` attribute is essential for support of the GetMyDN procedure used by Access Presence.

The following Stream DUA script enables the GetMyDN procedure (note that the Stream DUA must run with the -d option to modify the root entry):

```
modify
      add values anonymousPrivilege {
          protocol { dapProtocol, dapAdmProtocol },
          credentialType { noCredentials },
          privilege { accessLevel none }
};
```

Values of `anonymousPrivilege` with `dspProtocol` defined are used for interworking with other DSAs that do not provide bind credentials. The following Stream DUA script allows other anonymous DSAs to read entries:

```
modify
      add values anonymousPrivilege {
          protocol { dspProtocol },
          credentialType { noCredentials, noNameNoPasswd,
              noNameZeroPass, nameNoPasswd },
          privilege { accessLevel read }
};
```

LDAP password management

Access to the directory information held in the DSA requires a DUA, or another DSA, to provide their credentials, which are then used to authenticate the user. The security behind the LDAP simple-authentication model, which is the most widely used authentication mechanism, rests on how difficult the user's password is to crack.

Applying an *LDAP Password Policy* allows the directory administrator to define a set of rules that will enhance the security of the user's password, and ultimately that of the directory information in the DSA.

The directory allows the LDAP Password Policy to vary with the position of an entry in the Directory Information Tree (DIT). The region of the DIT subject to a particular password policy is known as a *password policy administrative area*.

Password Policy rules are created by configuring various operational attributes and object classes defined by the Internet Drafts, *draft-behera-ldap-password-policy-05* and *draft-behera-ldap-password-policy-10* (for `pwdMaxIdle` and `pwdLastSuccess` only). The configuration of these operational attributes is discussed later in this section.

The ViewDS DSA supports this password management feature for LDAP operations. In addition, some extensions have been made to DAP to allow password policy requests and responses for bind, compare and modify operations so that password policy to be enforced though Access Presence.

Password usage policy

The LDAP Password Policy can be enforced for many different types of password usage. The various types of password policies which may be enforced are described in this section.

Password guessing limit

The purpose of this password policy is to minimize the threat of password guessing attacks. Enforcing a password guessing limit policy causes consecutive failed authentication attempts to be tracked and acted upon when the limit is reached.

This policy consists of five parts:

- A configurable limit of consecutive failed authentication attempts.
- A counter to track the number of consecutive failed authentication attempts.
- A timeframe in which the consecutive failures must occur within before action is taken.
- The action to be taken when the limit is reached.
- The amount of time that the account is locked (only applicable if account locking is enabled).

Password expiration

A key factor in ensuring that a password is not compromised is that it is not well known. If a password is frequently changed, the chances of the user's account being broken into are minimized.

The password policy can be configured to cause passwords to expire after a given amount of time, thus forcing users to change their passwords periodically.

To make the user aware that they must change their password, one or both of the following methods may be used:

- The user is sent a warning sometime before their password is due to expire. If the user fails to heed this warning before the expiration time, the account will be locked.
- The user may bind to the directory a preset number of times after their password has expired. If the specified limit of grace logins is exceeded, without the password being changed, the account will be locked.

Password history

The objective of the password expiration model, which states that a password will be more secure if it is changed frequently, can be rendered useless if a user constantly reuses old passwords.

By specifying a number of passwords to be remembered, a password value will be included into the history every time the value changes. Users will then be prevented from modifying their password to any of these previously used values.

Password minimum age

The password history mechanism may be circumvented if the user changes their password extremely frequently with the intention of flushing a specific previous password from the history. To deter users from doing this, a minimum age for passwords may be specified. This will enforce a time period, which must elapse, before a user is allowed to change their password.

Password syntax

In order to prevent users from choosing passwords that may be easy to guess, a password syntax policy can be employed. Currently the ViewDS DSA supports three mechanisms for this – minimum length enforcement, minimum distance enforcement and a password quality check.

Minimum length enforcement will ensure that passwords are at least a specific length, making brute force attacks a lengthy process for an intruder.

Minimum distance enforcement will ensure a minimum degree of change between an old password and a new one. This feature is a ViewDS-specific extension to the LDAP password policy standard.

In addition, we have implemented another ViewDS-specific extension to the LDAP password policy standard to allow a set of pre-defined password quality checks to be applied. One set of checks is provided, based on the guidelines in the *Australian Government Information Security Manual 2015*.

User defined passwords

In some cases, it may be desirable to disallow users from changing or creating their own password values. This is something that may be set as part of the policy, although care should be taken to ensure that this is not used in conjunction with password expiry, since users will be locked out until their password is changed by an administrator.

Password change after reset

This policy can be used to force the user to change their password after an administrator has created or changed it. This is especially useful when the administrator sets passwords to a well-known value, such a person's date of birth.

Safe modification

A potential security risk will arise if a client connects to a directory, and leaves the connection open for an extended period of time. This potentially allows an intruder to make modifications to the user's password whilst the computer is unattended.

This policy requires the user to provide their old password before a password can be changed.

Inactive account lockout

This policy can be used to lock an account automatically after it has been inactive for a specified period of time. Useful for ensuring that the accounts of staff who have left an organisation cannot be exploited.

Password Policy LDAP Control

The LDAP Password Policy uses an LDAP Control to convey password policy information from the DSA to the DUA.

An LDAP Control only applies to operations carried out using the LDAPv3 protocol. In order for useful information to be reported correctly, the DUA being used must be using LDAP Version 3 and include the `passwordPolicyRequest` control in their LDAP requests.

passwordPolicyRequest

This LDAP Control must be included into LDAP requests by DUAs wishing to receive `passwordPolicyResponse` information from the ViewDS DSA.

The `controlType` for this control is 1.3.6.1.4.1.42.2.27.8.5.1. The criticality must always be `FALSE`. This control does not have a `controlValue`.

passwordPolicyResponse

This LDAP Control is included in responses to DUAs which have acknowledged that they are willing to receive it by including the `passwordPolicyRequest` control in their requests.

This LDAP Control is the only mechanism available to convey password policy decisions, made by the DSA, back to the DUA.

The `controlType` for this control is 1.3.6.1.4.1.42.2.27.8.5.1. The criticality must be set to `FALSE`. The controlValue for this control is an OCTET STRING whose value is the BER encoding of the following structure:

```
PasswordPolicyResponseValue ::= SEQUENCE {
      warning    [0] CHOICE {
          timeBeforeExpiration  [0] INTEGER    (0 .. maxInt),
          graceLoginsRemaining  [1] INTEGER    (0 .. maxInt)
      } OPTIONAL,
      error      [1] ENUMERATED {
          passwordExpired       (0),
          accountLocked         (1),
```

```
            changeAfterReset      (2),
            passwordModNotAllowed (3),
            mustSupplyOldPassword (4),
            invalidPasswordSyntax (5),
            passwordTooShort      (6),
            passwordTooYoung      (7),
            passwordInHistory     (8)
        } OPTIONAL
}
```

Password policy administrative area

The region where a password policy applies is called a *password policy administrative area*. It encompasses a complete subtree of a DIT, excluding subordinate subtrees set up as independent password policy administrative areas.

The entry at the top of the subtree is the *password policy administrative point*. This entry holds a special operational attribute, `administrativeRole`, with the value `passwordPolicyArea`. The object identifier for this administrative role value is defined as:

```
passwordPolicyArea OBJECT IDENTIFIER ::=
      { 1 2 36 79672281 1 23 0 }
```

The actual password policy specification is stored in a subentry (a special subordinate entry) of this entry. The specifications are stored as operational attributes of the subentry. The operational attributes define the set of rules that make up the password policy. They can be modified or created using the Stream DUA by a user with sufficient access control privileges.

The password policy subentry requires a `subtreeSpecification` value to define which part of the subtree it applies to. The `subtreeSpecification` value should specify the entire subtree under the administrative point (i.e. all optional components should be absent). Only one password policy subentry can be defined in a *password policy administrative area*.

Password policy object class

Creating a password policy administrative point, and placing password policy operational attributes in the password policy subentry, sets up a password policy administrative area. The password policy subentry must contain the `pwdPolicy` object class.

pwdPolicy

As the definition below shows, this object class can be used to create a password policy with one or many of the password policy operational attributes. However, in order for a password policy entry to be valid it must contain the `pwdAttribute` operational attribute.

```
pwdPolicy OBJECT-CLASS ::= {
      SUBCLASS OF    { top }
      KIND           auxiliary
      MUST CONTAIN   { pwdAttribute }
      MAY CONTAIN    { pwdMinAge | pwdMaxAge | pwdInHistory |
                       pwdCheckSyntax | pwdMinLength |
                       pwdExpireWarning | pwdGraceLoginLimit |
```

```
                        pwdLockout | pwdLockoutDuration | pwdMaxFailure |
                        pwdFailureCountInterval | pwdMustChange |
                        pwdAllowUserChange | pwdSafeModify | pwdMaxIdle }
            ID          id-sun-sc-pwdPolicy
      }
```

Password policy operational attributes

The following operational attributes can be included within the password policy subentry to create the final password policy. At the very least, `pwdAttribute` must be included within the password policy subentry.

pwdAttribute

This operational attribute contains all of the attributes that the password policy will be applied to. This operational attribute must be present in all password policy subentries.

```
pwdAttribute ATTRIBUTE ::= {
      WITH SYNTAX               OBJECT IDENTIFIER
      EQUALITY MATCHING RULE    objectIdentifierMatch
      ID                        id-sun-at-pwdAttribute}
```

pwdMinAge

This operational attribute holds the number of seconds that must elapse from the time a password is created until it can be modified. If this attribute is not present a value of 0 is assumed.

```
pwdMinAge ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdMinAge}
```

pwdMaxAge

This operational attribute holds the number of seconds after which the password will expire. If this attribute is not present or has a value of 0, the password will never expire.

```
pwdMaxAge ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdMaxAge}
```

pwdInHistory

This operational attribute specifies the number of passwords that will be stored in history. If this attribute is not present, or has a value of 0, passwords will not be stored in history.

```
pwdInHistory ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdInHistory}
```

pwdCheckSyntax

This operational attribute indicates the level of syntax checking performed on password values that are being added or modified. This attribute can hold one of three values:

- `0` – indicates that syntax checking will not be performed. This is equivalent to the attribute not being present.
- `1` – indicates that the DSA will check the syntax of the password value if it can process the password's original value correctly (for example, it is a clear-text or encrypted value). Passwords that the DSA cannot process will be accepted.
- `2` – indicates that syntax checking must be completed satisfactorily for the password to be accepted. If the password's syntax cannot be accurately processed by the DSA (for example, hashed value) then the password will not be accepted.

It is defined as follows:

```
pwdCheckSyntax ATTRIBUTE ::= {
      WITH SYNTAX                INTEGER
      EQUALITY MATCHING RULE     integerMatch
      SINGLE VALUE               TRUE
      ID                         id-sun-at-pwdCheckSyntax}
```

Note. If syntax checking is enabled, then password syntax checks can be implemented using the `pwdMinLength` (below) and ViewDS-specific operational attributes `viewDSPasswordQuality` and `viewDSPasswordDistance`. See *viewDSPasswordQuality* and *Sequential* passwords cannot be used viewDSPasswordDistance on page 140 for further details.

pwdMinLength

This operational attribute holds the minimum length (in characters) of a password. This part of the policy is only used when `pwdCheckSyntax` is present and not equal to zero. If this attribute is not present or has a value of 0, the length of the password will not be checked.

```
pwdMinLength ATTRIBUTE ::= {
      WITH SYNTAX                INTEGER
      EQUALITY MATCHING RULE     integerMatch
      SINGLE VALUE               TRUE
      ID                         id-sun-at-pwdMinLength}
```

pwdExpireWarning

This operational attribute holds the maximum number of seconds before a password is due to expire that an expiration warning will be sent to the user. If this attribute is not present or has a value of 0, no warnings will be sent.

```
pwdExpireWarning ATTRIBUTE ::= {
      WITH SYNTAX                INTEGER
      EQUALITY MATCHING RULE     integerMatch
      SINGLE VALUE               TRUE
      ID                         id-sun-at-pwdExpireWarning}
```

pwdGraceLoginLimit

This operational attribute holds the number of times an expired password can be used to authenticate. If this attribute is not present, or has a value of 0, the user will not be able to login after the password has expired.

```
pwdGraceLoginLimit ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdGraceLoginLimit}
```

pwdLockout

This operational attribute indicates whether a password will be able to be used after a number of consecutive bind failures. The maximum number of consecutive failed bind attempts is specified in `pwdMaxFailure`. If this attribute is not present, or has a value of `FALSE`, the entry will not be locked after a number of consecutive failed bind attempts.

```
pwdLockout ATTRIBUTE ::= {
      WITH SYNTAX               BOOLEAN
      EQUALITY MATCHING RULE    booleanMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdLockout}
```

pwdLockoutDuration

This operational attribute holds the number of seconds that a password cannot be used due to too many consecutive failed bind attempts (i.e. `pwdLockout` attribute value is `TRUE`). If this attribute is not present, or has a value of 0, the password cannot be used to authenticate until `pwdLockout` is reset by the administrator.

```
pwdLockoutDuration ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdLockoutDuration
}
```

pwdMaxFailure

This operational attribute holds the number of consecutive failed authentication attempts after which a password may not be used to authenticate. If this attribute is not present, or has a value of 0, this policy is not checked and the value of `pwdLockout` will be ignored.

```
pwdMaxFailure ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdMaxFailure
}
```

pwdFailureCountInterval

This operational attribute holds the number of seconds after which password failures are purged from the failure counter. If this attribute is not present, or has a value of 0, the password failure counter will only be reset by a successful bind.

```
pwdFailureCountInterval ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdFailureCountInterval
}
```

pwdMustChange

This operational attribute indicates whether a password must be changed after the password has been reset by an administrator. If this attribute is not present a value of `FALSE` will be assumed.

```
pwdMustChange ATTRIBUTE ::= {
      WITH SYNTAX               BOOLEAN
      EQUALITY MATCHING RULE    booleanMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdMustChange
}
```

pwdAllowUserChange

This operational attribute indicates whether a user is allowed to change their own password. This operational attribute only has an effect when the user is authorized by access controls to change their password and we would like the password policy to disallow this privilege. If this attribute is not present, a value of `TRUE` will be assumed.

```
pwdAllowUserChange ATTRIBUTE ::= {
      WITH SYNTAX               BOOLEAN
      EQUALITY MATCHING RULE    booleanMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdAllowUserChange
}
```

pwdSafeModify

This operational attribute specifies whether a user must provide their current password before it can be modified. If this attribute is not present a value of `FALSE` will be assumed.

```
pwdSafeModify ATTRIBUTE ::= {
      WITH SYNTAX               BOOLEAN
      EQUALITY MATCHING RULE    booleanMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdSafeModify
}
```

pwdMaxIdle

This operational attribute specifies the number of seconds an account may remain unused before it becomes locked. If this attribute is not set or is 0, no check is performed.

```
pwdMaxIdle ATTRIBUTE ::= {
      WITH SYNTAX               INTEGER
      EQUALITY MATCHING RULE    integerMatch
      SINGLE VALUE              TRUE
      ID                        id-sun-at-pwdMaxIdle
}
```

Password policy state information

To maintain adequate information regarding the state of each password, information is stored within the user entries in operational attributes. To maintain state for multiple attributes within a single entry, the password policy utilizes attribute options.

Password policy state attribute option

Most of the operational attributes used to maintain state information must have an option to specify which attribute, specified in `pwdAttribute`, it applies to. The password policy option is described as the following:

`pwd-<passwordAttribute>`

where `passwordAttribute` is a string following the OID Syntax.

For example, the option for the `userPassword` attribute with the `pwdChangedTime` operational attribute would be:

`pwdChangedTime;pwd-userPassword: 19790322120000Z`

This attribute option follows subtyping semantics. If a client requests a password policy state attribute to be returned in a search operation, and does not specify an option, all subtypes of that policy state attribute will be returned.

Password policy state operational attributes

The operational attributes used by the password policy to maintain state for each entry are described here. All of these operational attributes use attribute options except for the `pwdPolicySubentry`.

pwdChangedTime

This operational attribute specifies the last time that the entry's password was changed. If this attribute does not exist, the password will never expire.

```
pwdChangedTime ATTRIBUTE ::= {
      WITH SYNTAX               GeneralizedTime
      EQUALITY MATCHING RULE    generalizedTimeMatch
      ORDERING MATCHING RULE    generalizedTimeOrderingMatch
      USAGE                     directoryOperation
      ID                        id-sun-at-pwdChangedTime
}
```

pwdAccountLockedTime

This operational attribute holds the time that the user's account became locked. If this value is `0000010100Z` (the lowest syntactically correct `GeneralizedTime`), the account has been locked permanently and only the administrator can unlock the password.

```
pwdAccountLockedTime ATTRIBUTE ::= {
      WITH SYNTAX               GeneralizedTime
```

```
      EQUALITY MATCHING RULE    generalizedTimeMatch
      ORDERING MATCHING RULE    generalizedTimeOrderingMatch
      USAGE                     directoryOperation
      ID                        id-sun-at-pwdAccountLockedTime
}
```

pwdExpirationWarned

This operational attribute contains the time that the user was first sent a password expiration warning. The password will be due to expire at the time indicated by the sum of this time and the number of seconds specified by the `pwdExpireWarning` attribute.

```
pwdExpirationWarned ATTRIBUTE ::= {
      WITH SYNTAX               GeneralizedTime
      EQUALITY MATCHING RULE    generalizedTimeMatch
      ORDERING MATCHING RULE    generalizedTimeOrderingMatch
      USAGE                     directoryOperation
      ID                        id-sun-at-pwdExpirationWarned
}
```

pwdFailureTime

This operational attribute holds the timestamps of the consecutive authentication failures.

```
pwdFailureTime ATTRIBUTE ::= {
      WITH SYNTAX               GeneralizedTime
      EQUALITY MATCHING RULE    generalizedTimeMatch
      ORDERING MATCHING RULE    generalizedTimeOrderingMatch
      USAGE                     directoryOperation
      ID                        id-sun-at-pwdFailureTime
}
```

pwdHistory

This operational attribute holds the history of previously used passwords. Values of this attribute are transmitted in string format as given by the following ABNF:

```
pwdHistory= time '#' syntaxOID '#' length '#' data

time            = <generalizedTimeString as specified in 6.14 of
RFC2252>

syntaxOID       = numericoid         ; The string representation of the
                                   ; dotted-decimal OID that defines the
                                   ; syntax used to store the password.

length          = numericstring      ; the number of octets in data

data            = <octets representing the password in the format
                      specified by syntaxOID>.
```

Its definition is:

```
pwdHistory ATTRIBUTE ::= {
      WITH SYNTAX               OCTET STRING
      EQUALITY MATCHING RULE    octetStringMatch
      USAGE                     directoryOperation
      ID                        id-sun-at-pwdHistory
}
```

pwdGraceUseTime

This operational attribute holds the timestamps of grace logins used after a password has expired.

```
pwdGraceUseTime ATTRIBUTE ::= {
      WITH SYNTAX               GeneralizedTime
      EQUALITY MATCHING RULE    generalizedTimeMatch
      USAGE                     directoryOperation
      ID                        id-sun-at-pwdGraceUseTime
}
```

pwdReset

This operational attribute indicates whether an administrator has changed the password. It is used to enforce the change after reset policy indicated by `pwdMustChange` attribute.

```
pwdReset ATTRIBUTE ::= {
      WITH SYNTAX               BOOLEAN
      EQUALITY MATCHING RULE    booleanMatch
      USAGE                     directoryOperation
      ID                        id-sun-at-pwdReset
}
```

pwdPolicySubentry

This operational attribute holds the DN of the password policy subentry in effect for this object. This attribute is maintained automatically by the DSA and cannot be modified by a DUA.

```
pwdPolicySubentry ATTRIBUTE ::= {
      WITH SYNTAX               DistinguishedName
      EQUALITY MATCHING RULE    distinguishedNameMatch
      NO USER MODIFICATION      TRUE
      USAGE                     directoryOperation
      ID                        id-sun-at-pwdPolicySubentry
}
```

pwdLastSuccess

This operational attribute holds the timestamp of the last successful authentication. It is used to enforce the maximum account idle time policy indicated by the `pwdMaxIdle` attribute.

```
pwdLastSuccess ATTRIBUTE ::= {
      WITH SYNTAX               GeneralizedTime
      EQUALITY MATCHING RULE    generalizedTimeMatch
      ORDERING MATCHING RULE    generalizedTimeOrderingMatch
      USAGE                     directoryOperation
      ID                        id-sun-at-pwdLastSuccess
}
```

Note. The `pwdLastSuccess` attribute implemented here differs from that described in *draft-behera-ldap-password-policy-10* in that it is a user modifiable attribute.

ViewDS-specific password policy operational attributes

viewDSPasswordQuality

This operational attribute has a syntax of object identifier and is multi-valued. It is used to identify the set of password-quality checks that should be carried out if the pwdCheckSyntax behaviour is enabled (see *pwdCheckSyntax* on page 134). Currently only one set of password-quality checks is defined:

```
pwdQuality-agism OBJECT IDENTIFIER ::=
{ 1 3 6 1 4 1 21473 5 7 2 1 }
```

This set is based on guidelines in the *Australian Government Information Security Manual 2015*, and imposes the following:

- Passwords must have a minimum of 10 characters and must contain at least one character from three of the following four character sets:
  - Lowercase Unicode characters (such as a – z)
  - Uppercase Unicode characters (such as A – Z)
  - Numeric characters (0-9)
  - Special characters (such as ! @ # $ %, etc.)
- Sequential passwords cannot be used

viewDSPasswordDistance

This operational attribute has an integer syntax and is single-valued. It is used to determine the degree of change between an old password value and a new password value based on the sum of the distances between code points of the respective characters, plus the difference in length of the passwords. It prevents the use of sequential passwords.

When the `pwdQuality-agism` mechanism is in use (see above), it defaults to 1, otherwise it defaults to 0.

Adding a password policy

The standard ViewDS, as supplied on the installation media, does not come pre-configured with a password policy. Adding, removing or modifying a password policy can be done at any time using the Stream DUA.

Creating a password policy administrative area

The first step in creating a password policy is to identify which administrative point should contain the password policy as a subentry.

If this entry does not exist it will need to be created, as illustrated by the following Stream DUA example:

```
insert {
    organizationName "Deltawing"
    / organizationalUnit "Password Policy"
}
with {
    objectClass organizationalUnit,
    administrativeRole passwordPolicyArea
};
```

If the entry does exist, but is not currently an administrative point, the following Stream DUA command will make this entry suitable to house a password policy subentry:

```
modify {
    organizationName "Deltawing"
    / organizationalUnit "Deltawing InfoSystems"
}
with changes {
    add attribute administrativeRole passwordPolicyArea
};
```

If the entry exists and is currently an administrative point for other administrative roles, the following Stream DUA command will make it suitable to house a password policy subentry:

```
modify {
    organizationName "Deltawing"
}
with changes {
    add values administrativeRole passwordPolicyArea
};
```

Creating the password policy subentry

Before a password policy subentry is created you must know exactly what policy you would like to enforce its scope within the DIT.

The following Stream DUA example creates a password policy under the Deltawing administrative point. This password policy will lock accounts for one day if the user incorrectly guesses their `userPassword` value five times.

```
insert {
    organizationName "Deltawing"
    / commonName "LDAP Password Policy" }
with {
    objectClass subentry pwdPolicy,
    subtreeSpecification { },
    pwdAttribute userPassword,
    pwdMaxFailure 5,
    pwdLockout TRUE,
    pwdLockoutDuration 86400
};
```

Once this has been done, all entries in the subtree will have their `userPassword` attribute subjected to the password policy.

Miscellaneous security topics

This subsection describes the following security topics:

- Secure Sockets Layer (SSL)
- XML digital signature
- Dump and log-file security
- Proxy permissions
- Value hashing

Secure Sockets Layer (SSL)

ViewDS can protect LDAP access with Secure Sockets Layer (SSL) security. SSL provides server authentication to the client (for example, DSA to DUA) and data confidentiality (encryption) for subsequent communication between the client and the server (including client authentication to the server). Authentication is via X.509 certificates and uses the same DSA certificate described earlier for strong authentication.

Enabling SSL/TLS requires the server key pair to be set up (see *Strong authentication* on page 112).

XML digital signature

ViewDS can protect SAML-based XACML access through XML digital signature security. XML digital signatures provide server authentication to the client (for example, PDP to PEP) and data integrity (signing) for communication between the PEP and the PDP (including PEP authentication).

Authentication is via X.509 certificates and uses the same DSA certificate described previously for strong authentication.

Enabling XML digital signatures requires the server key pair to be set up (see *Strong authentication* on page 112).

ViewDS support for XML digital signatures has been implemented according to the *XML Signature Syntax and Processing* Second Edition (2008) specification using the methods described in Section 5: *SAML and XML Signature Syntax and Processing* of the *SAML Core Specification* 2.0 (2005) with the following variation:

- The KeyInfo element is always included in a signed response and contains either the X509Certificate or the X509SubjectName.

Creating signed responses

ViewDS always signs responses to *signed* requests. The `dsigsignresponse` configuration parameter allows you to specify whether ViewDS signs responses to *unsigned* requests.

The signed response includes either the X509Certificate or the X509SubjectName in the KeyInfo element for client verification. The `dsigx509data` configuration option allows you to specify which of these should be provided.

Verifying signed requests

Processing a signed XACML request involves two steps: the validation of the digital signature and the authentication of the signer.

The first step verifies message integrity and ensures that the content of the XACML request has not been changed since it was signed.

The second step allows ViewDS to authenticate the signer and ensure that it is trusted. The validation of client authentication credentials is completed using the same process that is described for strong authentication (see *Other authentication requirements* on page 114).

Any XACML request that is signed and successfully verified is considered to have an authentication level of Strong for authorization purposes.

Dump and log-file security

The DSA encrypts values of `userPassword` when writing them to a dump file (the output from the Stream DUA dump command) or a log file. This protects the `userPassword` values, even from directory administrators, including the super-administrator.

Values are encrypted using a special 32 character hexadecimal string encoding a 16 byte AES key which can be set using the DSA Controller. If no key string has been set, the DSA uses a default key.

Loading encrypted data

An encrypted value in a dump or log file has the string prefix `ENCRYPTED:`. To load these dump files, or replay logs, the Stream DUA needs the key that was used to generate the encrypted values.

Unless explicitly overridden, Stream DUA attempts to read the key from the special password file defined by the configuration-file parameter `admpasswd` (usually `${VFHOME}/general/deity`). Note that this requires Stream DUA to be invoked by the same user who started the DSA (i.e. the DSA account user). Otherwise – or when transferring dump or log files between systems in which different keys are used – the key must be specified explicitly.

The Stream DUA takes a command line option, which sets the key to be used when uploading:

```
sdua [-K string]
```

There is an optional qualifier on the dump command to specify the key when dumping:

```
dump [name] [with key string];
```

If *string* is an empty string, the default key is used.

Changing the key

To improve security, it is advisable to set a key rather than rely on the built-in default key (which is by definition the same for every ViewDS installation). Set a different key using the DSA Controller as follows:

```
dsac setwrite key=[32 character hexadecimal string]
```

Once a new key has been set, it is necessary to supply the old key to the Stream DUA when loading data that was dumped or logged using the old key. This is normally only required when making a transition from one key to another.

When dumping data to supply to another user for modifying and uploading, dump and with a different key and supply it to the user.

NOTE: The value of the key that has been set using the DSA Controller is NOT displayed by the `dsac display` command.

Proxy permissions

The DSA can assign a proxy capability to individual users. The proxy capability allows a user to indicate, for each operation, the DN of another user whose access rights should be used when performing the operation. This allows a single entity to act on behalf of a number of different users without having to constantly bind to the DSA with each user's credentials.

The capability can be granted by adding a value of the operational attribute `proxyAgent` to a user's entry. This attribute defines:

- that the user is permitted to act on behalf of other users; and
- the authentication level to use when making access control decisions using another user's identity.

To use the proxy capability, a DUA must bind to the directory using the credentials of the user granted the proxy capability. It can then specify an alternative user's DN in the `requestor` field in the `CommonArguments` of a DAP operation or using the proxy authorisation control for an LDAP operation.

The DUA can act using its own credentials by sending operations without the `requestor` field. It can act using anonymous credentials by sending operations with the `requestor` field present and set to an empty sequence of RDNs.

The only ViewDS DUAs that can use the proxy capability are Access Presence and the Stream DUA.

proxyAgent

This attribute is a single valued operation attribute.

```
proxyAgent ATTRIBUTE ::= {
      WITH SYNTAX        AuthenticationLevel
      SINGLE VALUE       TRUE
      USAGE              directoryOperation
      ID                 { ads 24 0 }    }
```

Value hashing

ViewDS rigorously protects the `userPassword` attribute and never reveals its clear-text value. By default, the `userPassword` value is protected using a symmetric encryption scheme. Two-way encryption allows the `userPassword` value to be decrypted into clear-text for internal processing.

If you consider two-way encryption of the `userPassword` inadequate, or if you would like to protect other attributes, enable value hashing.

After enabling value hashing for an attribute, any new values of the attribute are stored as hashed values rather than clear text. Converting existing values to hashed values involves dumping and reloading the directory.

In conjunction with storing hashed values, the value-hashing policy may be configured to control the format of the values returned by ViewDS for the protected attribute.

Before implementing a value-hashing policy, read and understand the following precautionary notes. For information on setting up a value-hashing policy, see `attributeTypeExtensions` on page 95.

Original value reversion

It is very important to note that once a value has been hashed, it is impossible to obtain a clear-text value of the stored password. From a security perspective this is an ideal situation. However, it also means that the original passwords cannot be replicated or loaded onto another non-ViewDS directory (described below).

Directory replication

If a ViewDS directory replicates data containing a hash-protected attribute to a non-ViewDS directory, it will not operate as intended.

When an attribute is hash protected, the actual values are stored as values with context or in the case of LDAP, attribute options are used. These options or contexts allow ViewDS to identify whether the value is hashed and with which algorithm. The following identifiers are used by ViewDS as context values or LDAP attribute options to specify the hash algorithm used:

- `x-hashed-crypt` – for the Unix Crypt algorithm
- `x-hashed-md5` – for the MD5 algorithm
- `x-hashed-sha` – for the SHA algorithm
- `x-hashed-sha1` – for the SHA1 algorithm

These identifiers begin with the `x-` prefix to indicate that they are experimental and ViewDS specific. Since other directories do not understand these identifiers, it is unclear how they would treat the attributes' values. However, it is certain that they would not treat them as hashed values. This would result in failure of bind and query operations that contain a clear-text value of the attribute.

LDAP Password Policy – syntax checking

The subsection for the `pwdCheckSyntax` attribute (see page 134) describes how ViewDS behaves if it cannot process a password's syntax correctly.

When value hashing is turned on, the clear-text value of a password is no longer available for ViewDS to analyse. Because the length of the password cannot be determined from the hashed value, ViewDS cannot check the syntax.

It is recommended that the `pwdCheckSyntax` value should not be set to `2` when hashing is on, since this will result in every password value being rejected.

LDAP Password Policy – history

The LDAP Password Policy states that a deleted or removed password should go into the *history list*. Since the policy makes no allowances for identifying whether values in history are hashed, they can be whatever was stored in the directory at the time.

When the history list is checked, all values are treated as the value relating to the directory's current hashing policy. Therefore, if you change the hashing policy, previously used passwords can be reused.

Example

Consider a scenario where MD5 hashing is turned on and the user has a password of `testpass`. If the user changes their password to `abc`, then the history list will contain the encoded hash, `F5rUXGziy5fPECniEgRugQ==`.

If MD5 hashing is then turned off and the user changes their password to `testpass`, ViewDS will not be able to identify the value in history as relating to the value provided.

Chapter 7 Replicating or distributing data

This chapter provides an overview of X.500 distributed operations, and describes how to configure the ViewDS *Directory System Agent* (DSA) with knowledge of other DSAs to allow distributed operations. That is, to allow a DSA to contact other DSAs to satisfy a user operation. It also describes how to configure the DSA for replication, so that a DSA can supply a copy of its data to another DSA or hold a copy of data mastered in another DSA (in accordance with X.525).

This chapter has the following sections:

- Distributed operations overview
- DSE types
- Access points
- Knowledge attributes
- Reference example
- Setting up a naming context
- Setting up the root entry
- Cross references
- Knowledge example
- Remote aliases
- Replication
- Setting up a shadowing agreement
- Replication attributes
- Converting shadow into master
- Replication example
- LDAP change log

NOTE: A directory can be replicated or distributed using the ViewDS Management Agent or Stream Directory User Agent (Stream DUA). A basic configuration can be implemented using the ViewDS Management Agent, and then embellished using the Stream DUA.

Distributed operations overview

Distributed operations and the need for knowledge configuration arise when the Directory Information Tree (DIT) is split across multiple DSAs.

All directory entries in all systems which share directory information with one another via X.500 must belong to the same DIT. If the whole DIT is held in a single DSA, then there are no entries to be held in other DSAs, and no special setup for distributed operations is required. Such a model may be appropriate for a centralized corporate directory which has no connections to other corporate or public directories.

If a DIT is split between DSAs, a given DSA will in practice hold a small number of branches of the whole DIT. Each branch is called a *naming context*.

Naming contexts

A connected subtree of real directory information in a DSA is called a naming context. A naming context can consist of as little as a single entry, or can comprise the whole DIT. In general, a DSA will hold a small number of naming contexts.

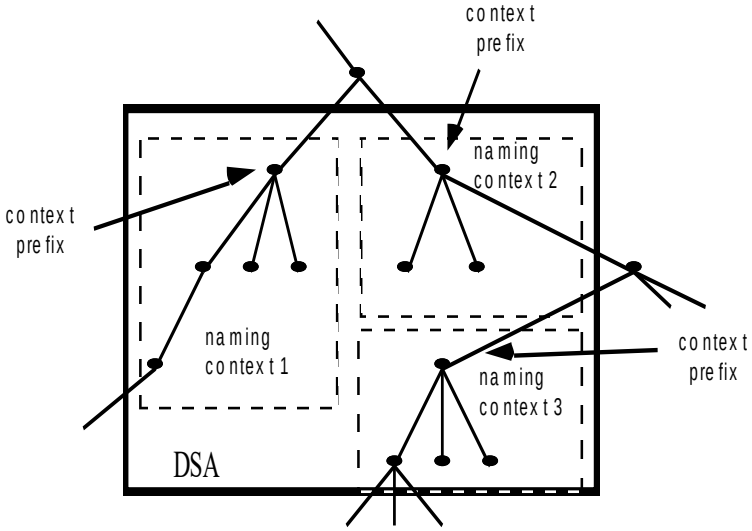


Figure 5: Naming contexts in a DSA

A naming context may start with an entry immediately subordinate to the DIT root (the DIT root can never be a real entry), or an entry deeper down in the DIT. The entry at the top of the naming context is called the *context prefix*. A context prefix is often a subschema administrative point (see page 51).

A naming context may extend to the leaves of the subtree starting at the context prefix, or it may be ‘pruned’, with branches held in other DSAs. Each of those remotely held branches is then a naming context in the other DSA.

Kinds of knowledge

Knowledge is information that allows a DSA to locate an entry held in another DSA. It consists of the access point of the DSA and may contain the name of the entry explicitly, or may simply contain a superior name.

For each naming context that a DSA holds, it must hold two kinds of knowledge:

- *Immediate superior knowledge*, which is knowledge of the DSA that holds the naming context that is immediately superior to this context prefix.
- *Subordinate knowledge* of the DSAs that hold naming contexts that are immediately subordinate to entries in this naming context.

It can be seen that the two kinds of knowledge make up a bi-directional link: if DSA2 has immediate superior knowledge that refers to DSA1, then DSA1 has subordinate knowledge that refers to DSA2. Note that immediate superior knowledge is stored using the `specificKnowledge` attribute, not the `superiorKnowledge` attribute.

In addition to its immediate superior knowledge, a DSA must hold knowledge of one DSA to turn to when it can't resolve any components of an entry's name. Such knowledge is called a *superior reference*, and is stored using the `superiorKnowledge` attribute.

If the naming context begins with an immediate subordinate of the DIT root, there is no DSA that holds the immediately superior entry (since no entry holds the real DIT root), and superior knowledge is absent. The DSA is called a *first-level DSA*.

If the naming context extends to the leaves in every case then there are no subordinate naming contexts and subordinate knowledge is absent.

There are two kinds of subordinate knowledge, depending on whether the superior DSA knows the name(s) of the immediately subordinate context prefixes in the subordinate DSA:

- A *specific subordinate reference* identifies a particular subordinate DSA with a named subordinate which is a context prefix in that DSA.
- A *non-specific subordinate reference* identifies a particular subordinate DSA with unnamed subordinates in that DSA.

First-level and subordinate DSAs

If a DSA receives a query involving an entry which attaches to the DIT at a point higher than any entry it holds, it passes the query to another DSA called the *superior DSA*. The DSA itself is, in the context of the query, the *subordinate DSA*. The superior DSA is chosen because it holds entries closer to the DIT root and is more likely to be able to process the query.

This process cannot continue indefinitely, and since the DIT root is not a real entry, the DSAs that hold immediate subordinates of the root must be able to handle the query. They do this by maintaining a (potentially long) list of peer DSAs, and are prepared to resolve any query by referring or chaining the query to those peers.

In terms of the knowledge model, first-level DSAs all hold the knowledge that would be held if the DIT root entry were a real entry in that DSA with specific subordinate references to all other first-level DSAs.

A DSA is classed as a first-level DSA if it holds any entry which is an immediate subordinate of the root. The entry will of course be a context prefix.

A special case is an isolated first-level DSA. This is a first-level DSA with no peers. It communicates with its subordinate DSAs, but is able to partially name-resolve every query. Such a DSA may be appropriate for a company that wants to limit directory operations to its own DSAs which are interconnected in a hierarchical fashion.

Configuring a DSA for distributed operations

Configuring a DSA for distributed operations involves two steps:

- Setting up each of the naming contexts in the DSA, which includes setting up superior and subordinate knowledge for the context prefix.
- Setting up the root entry.

These steps are described in subsequent sections. Note that the first step generally involves making knowledge configuration changes at more than one DSA. This is because every naming context involves at least two DSAs: the one holding the context prefix, and the one holding the superior of the context prefix.

DSE types

Directory information in a particular DSA is held in DSA-specific entries (DSEs). Each DSE holds the name of an entry and may (but need not) hold other information associated with the corresponding directory entry.

An entry in the DIT may be held in different DSAs in a variety of forms. It may be a real entry in one DSA, a shadowed entry in another, a glue entry (providing a name only) in a third DSA, and a knowledge reference in a fourth DSA. The role played by the entry in a specific DSA is called its DSE type, and is held in an operational attribute called `dseType`.

dseType

The `dseType` attribute records the type of each entry held in a DSA. Every entry in a DSA has this attribute. However, it is normally suppressed when the ViewDS DSA dumps directory information because the DSA can generally infer the appropriate value when adding an entry.

In general, ViewDS sets the `dseType` of an entry according to attributes it contains rather than any supplied `dseType` value. The following rules apply:

- If the entry is created without an `objectClass` attribute and is added using the DAP Admin Protocol, it is automatically assigned a `dseType` of `glue`. If it has an `objectClass` attribute it will be a glue entry if the `glue` bit is set in the supplied `dseType` value.
- If `specificKnowledge` is present then the `xr`, `subr`, `immSupr` and `sa` bits are taken from the supplied `dseType` value.
- If `supplierKnowledge` is present the `shadow` bit is set. Otherwise, it is set based on the supplied `dseType` value. If no `dseType` value was supplied then the `shadow` bit is set to match its parent entry.
- If the supplied `dseType` and the parent entry's `dseType` both have the `root` bit set, then the actual `dseType` will also have the `root` bit set.

In particular, the `dseType` must be given explicitly for entries which hold the `specificKnowledge` attribute in order to specify the kind of knowledge.

The attribute is defined as:

```
dseType   ATTRIBUTE  ::= {
      WITH SYNTAX           DSEType
      EQUALITY MATCHING RULEbitStringMatch
      SINGLE VALUE          TRUE
      NO USER MODIFICATION  TRUE
      USAGE                 dSAOperation
      ID                    {ds 12 0} }
DSEType   ::= BIT STRING {
      root           (0),    -- root DSE -
      glue           (1),    -- represents knowledge of a name only -
      cp             (2),    -- context prefix -
      entry          (3),    -- object entry -
      alias          (4),    -- alias entry -
      subr           (5),    -- subordinate reference -
      nssr           (6),    -- non-specific subordinate reference -
      supr           (7),    -- superior reference -
      xr             (8),    -- cross reference -
      admPoint       (9),    -- administrative point -
      subentry       (10),   -- subentry -
      shadow         (11),   -- shadow copy -
      immSupr        (13),   -- immediate superior reference -
      rhob           (14),   -- rhob information -
      sa             (15),   -- subordinate reference to alias entry -
      dsSubentry     (16),   -- DSA Specific subentry --
      familyMember   (17)}   -- family member --
```

sepType

The `sepType` attribute is a ViewDS-specific operational attribute held in entries whose `dseType` is `cp`, `alias`, `subr` or `nssr`. It is equivalent to `dseType` except it represents the bits of `dseType` as a multi-valued attribute with an enumerated integer syntax and facilitates indexing. The `sepType` attribute is normally never seen in ViewDS, being suppressed when an entry is dumped, and calculated by the DSA when an entry is loaded. It is defined as:

```
sepType ATTRIBUTE ::= {
      WITH SYNTAX           SEPType
      EQUALITY MATCHING RULEsepTypeMatch
      NO USER MODIFICATION  TRUE
      USAGE                 dSAOperation
      ID                    {vf 12 2} }
}
SEPType ::= ENUMERATED {
      cp             (0),    -- context prefix --
      alias          (1),    -- alias entry --
      subr           (2),    -- subordinate reference --
      nssr           (3),    -- non-specific subordinate reference --
      subentry       (4),    -- subentry --
      attIncomplete  (5),    -- attribute completeness --
      subIncomplete  (6),    -- subordinate completeness --
}
```

Access points

A DSA access point is represented using an ASN.1 syntax called `AccessPoint`. A number of standard operational attributes have this syntax, or a closely related one. The definition of `AccessPoint` is:

```
AccessPoint ::=  SET {
      ae-title           [0] Name,
      address            [1] PresentationAddress,
      protocolInformation [2]  SET OF ProtocolInformation OPTIONAL }

Name ::= CHOICE {
      rdnSequence        RDNSequence
}

ProtocolInformation  ::= SEQUENCE {
      nAddress           OCTET STRING,
      profiles             SET OF OBJECT IDENTIFIER }
```

`ae-title` holds the Relative Distinguished Name (RDN) of the DSA. It can be chosen arbitrarily. However, some third-party products recommend that the DSA name should belong to a naming context superior to any held in the DSA.

`address` holds the DSA's presentation address (see page 30).

`protocolInformation` holds optional information about the network profiles supported at `address`.

Example

The access point of a DSA with name `{ organizationName "Deltawing" / commonName "Deltawing DSA" }` and presentation address `{,0403H,0403H,{49520086FF00H}}` would be represented using Stream DUA notation as:

```
{
      ae-title rdnSequence :
          { organizationName "Deltawing" / commonName "Deltawing DSA"
},
      address {
          sSelector '0403'H,
          tSelector '0403'H,
          nAddresses { '49520086FF00'H }
      }
}
```

Knowledge attributes

Knowledge is information needed to find an entry in another DSA. It consists of the access point of the other DSA, and (optionally) the name of the entry in the other DSA.

There are several kinds of knowledge, represented by different attribute types:

- `supplierKnowledge` and `consumerKnowledge`, which are built automatically by the DSA from `consumerStatus` and `supplierStatus`; and
- `secondaryShadows` (which is not currently generated by ViewDS).

superiorKnowledge

This attribute holds the access point of nominated superior DSA's that can find entries which this DSA does not hold and whose superiors this DSA does not hold. It is only ever found in the DIT root entry. The definition of the attribute is:

```
superiorKnowledge    ATTRIBUTE  ::= {
      WITH SYNTAX               AccessPoint
      EQUALITY MATCHING RULE    accessPointMatch
      NO USER MODIFICATION      TRUE
      USAGE                     dSAOperation
      ID                       ds 12 2 }
```

`AccessPoint` is the access point of the superior DSA.

Each non-first-level DSA is required by X.500 to hold this attribute in their root DSE. The superior reference of a DSA may be chosen by one of the following methods:

- The administrator of the DSA may negotiate with the administrators of other DSAs to establish a superior reference path to a first level DSA.
- The superior reference may refer to any DSA which holds an entry whose DN has fewer RDN terms than any entry held by the DSA for which the superior reference is being chosen.

specificKnowledge

This single-valued attribute holds the access point of a single nominated DSA that holds the real entry information for the glue entry that contains this attribute. The real entry must be a context-prefix entry in that DSA. It is used to hold immediate superior references, specific subordinate references, and cross references. The definition of the attribute is:

```
specificKnowledge    ATTRIBUTE  ::= {
      WITH SYNTAX           MasterAndShadowAccessPoints
      EQUALITY MATCHING RULEmasterAndShadowAccessPointsMatch
      SINGLE VALUE          TRUE
      NO USER MODIFICATION  TRUE
      USAGE                 distributedOperation
      ID                            ds 12 3 }

MasterAndShadowAccessPoints    ::=  SET OF MasterOrShadowAccessPoint

MasterOrShadowAccessPoint   ::= SET {
      COMPONENTS OF         AccessPoint,
      category          [3] ENUMERATED {
                            master  (0),
                            shadow  (1) } DEFAULT master,
      chainingRequired  [5] BOOLEAN DEFAULT FALSE
}
```

`AccessPoint` is the access point of the other DSA.

`category` indicates whether the DSA holds master or shadow information.

nonSpecificKnowledge

This multi-valued attribute holds the access points of other DSAs that hold subordinates of the entry within which this attribute appears. It appears in the root entry of first-level DSAs, and in real entries at the bottom of a naming context that contain a non-specific subordinate reference.

```
nonSpecificKnowledge     ATTRIBUTE  ::= {
      WITH SYNTAX            MasterAndShadowAccessPoints
      EQUALITY MATCHING RULE masterAndShadowAccessPointsMatch
      NO USER MODIFICATION   TRUE
      USAGE                  distributedOperation
      ID                             ds 12 4 }
```

With this background, it is possible to describe how to set up distributed operations so that DSAs that hold only part of the DIT can communicate with other DSAs.

Reference example

This section provides a reference example used throughout this chapter to illustrate distributed operations and replication.

Assume Deltawing wishes to join its directory with that of a number of companies in the USA. The directories of those companies are all under the `C "US"` node of the DIT which can be accessed via a (non-ViewDS) DSA called DSA3.

Also assume that Deltawing wishes to split its directory between two DSAs: a main DSA and a secondary one holding the Deltawing InfoSystems organizational unit.

This leads to the following configuration:

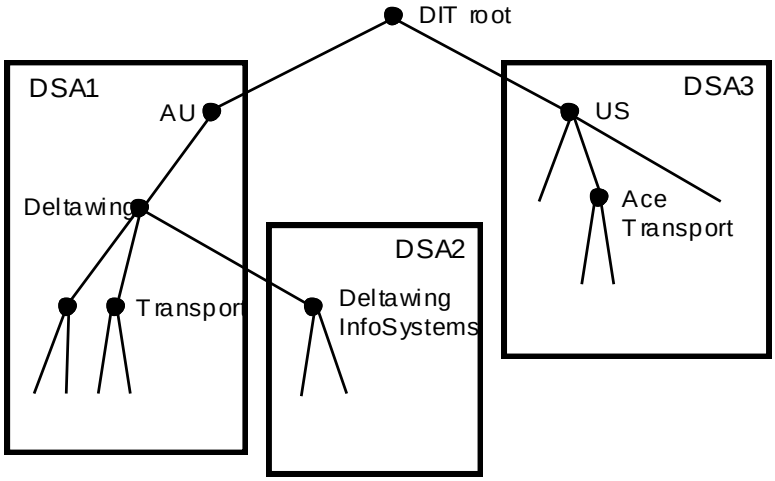


Figure 6: Example DIT and DSA configuration

The context prefixes are:

DSA1: `C "AU"`

DSA2: `C "AU" / organizationName "Deltawing" / organizationalUnit "Deltawing InfoSystems"`

DSA3: `C "US"`

Assume the three DSAs have the following access points. The strings *dsa1-access-point*, *dsa2-access-point*, and *dsa3-access-point* will be used in the examples, and should be replaced with the text shown below.

dsa1-access-point : replace with:

```
{
      ae-title rdnSequence :
          {C "AU" / organizationName "Deltawing" / commonName
          "Deltawing DSA"},
      address {
          tSelector '0401'H,
          nAddresses { '49520086FF00'H }
      }
}
```

dsa2-access-point : replace with:

```
{
      ae-title rdnSequence :
          { C "AU" / organizationName "Deltawing" / organizationalUnit
"Deltawing InfoSystems"
              / commonName "InfoSystems DSA" },
      address {
          tSelector '0401'H,
          nAddresses { '49520087FF00'H }
      }
}
```

dsa3-access-point : replace with:

```
{
      ae-title rdnSequence :
      {C "US" / organizationName "XYZCorp" / commonName "XYZCorp
      DSA"},
      address {
          sSelector '0A00'H,
          tSelector '0B00'H,
          nAddresses { '49520088FF00'H }
      }
}
```

Setting up a naming context

If the entry at the top of the naming context (the context prefix) is an immediate subordinate of the DIT root (as in DSAs 1 and 3), you need take no special action. For any other naming context (as in DSA 2), you need to:

- Create glue entries for entries between the DIT root and the context prefix.
- Determine which DSA holds the entry that is the real superior of the context prefix (the *superior DSA*), and determine the context prefix of the naming context of that superior entry (the *superior context prefix*).
- In this DSA, add an immediate superior reference to the superior DSA in the glue entry corresponding to the superior context prefix.
- In the superior DSA, add a specific or non-specific subordinate reference to the context prefix in this DSA.

Glue entries

For each of the entries between the root entry and the context prefix, it is necessary to create a glue entry that holds an RDN and joins its superior to its subordinate(s).

The purpose of the glue entry is simply to hold an RDN. The glue entry immediately superior to the context prefix entry also holds the immediate superior reference, and any glue entry can hold a cross reference.

Example

To add the entry `C "AU" / organizationName "Deltawing" / organizationalUnit "Deltawing InfoSystems"` to DSA 2 as a context prefix, add the following glue entries (knowledge and schema attributes are not shown):

```
entry
      C "AU"
with
      C "AU";
entry
      C "AU" / organizationName "Deltawing"
with
      organizationName "Deltawing", ;
```

Immediate superior reference

An immediate superior reference is required whenever the DSA holds an entry (which will be a context prefix) whose superior is in another DSA. This will be the case for every context prefix unless it is an immediate subordinate of the root.

The immediate superior reference is stored as an attribute of type `specificKnowledge` in the glue entry that corresponds to the superior context prefix in the superior DSA. The glue entry should be given an explicit `dseType` attribute which includes the value `immSupr`. The value of `specificKnowledge` is the access point of the DSA which holds the real superior entry.

Example

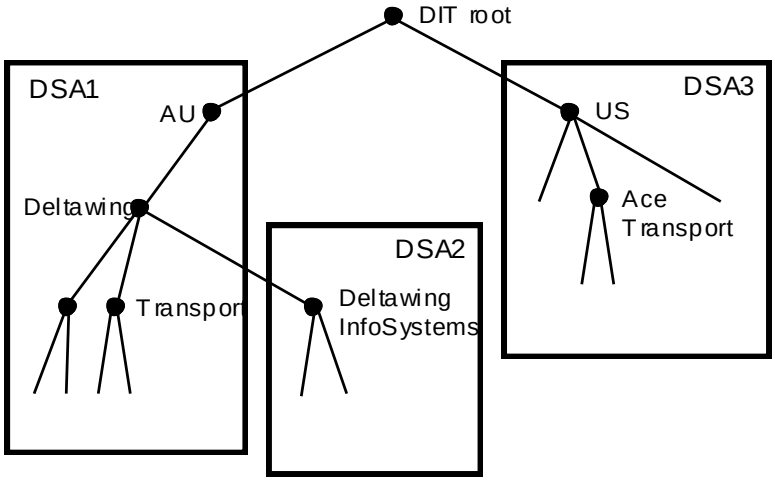


Figure 7: Example DIT and DSA configuration (repeated)

In the reference example (repeated above) DSA 1 holds the entry `Deltawing` and all its subordinates except for the subtree `Deltawing InfoSystems`; and DSA 2 holds the entry `Deltawing InfoSystems` and all its subordinates.

The entry `Deltawing InfoSystems` in DSA 2 is a context prefix, and DSA 2 must hold an immediate superior reference to DSA1 in the glue entry which is the context prefix for the immediate superior of `Deltawing InfoSystems`. The immediate superior is `Deltawing`, but since DSA1 also holds the superior of `Deltawing`, the entry `AU` is the context prefix.

A Stream DUA script for DSA 2 that adds an immediate superior reference to DSA1 is as follows:

```
modify {
      C "AU"
}
with changes {
      remove attribute dseType,
      add  attribute dseType {immSupr},
      add attribute specificKnowledge  { dsa1-access-point }
}
options manageDSAIT
;
```

NOTE: `dsa1-access-point` itself begins with an opening brace, so two opening braces appear when the access point is spelled out. The extra braces arise because `specificKnowledge` has syntax `SET OF MasterOrShadowAccessPoint`.

The subordinate reference

Every context prefix that is not an immediate subordinate of the DIT root has a corresponding real entry in another DSA. This other DSA (the superior DSA) has a subordinate that is the context prefix.

The real entry must have one of the following associated with it:

- a specific subordinate reference to this context prefix and DSA; or
- a non-specific subordinate reference to this DSA.

To illustrate, in the above example, DSA 1 must hold either a specific subordinate reference to `Deltawing InfoSystems` in DSA 2, or a non-specific subordinate reference to DSA 2.

Specific subordinate reference

A specific subordinate reference is created by adding an entry with the following characteristics:

- its name is that of the context prefix in the remote DSA;
- its `dseType` is `subr`; and
- it contains a value of the `specificKnowledge` attribute (defined above).

Example

The following Stream DUA script for DSA 1 adds a specific subordinate reference to `Deltawing InfoSystems` in DSA 2:

```
entry
      C "AU" / organizationName "Deltawing" / organizationalUnit
"Deltawing InfoSystems"
with
      organizationalUnitName "Deltawing InfoSystems",
      dseType {subr},
      specificKnowledge { dsa2-access-point }
;
```

Non-specific subordinate reference

A non-specific subordinate reference is created by adding an attribute of type `nonSpecificKnowledge` (defined above) to the superior entry.

Example

The following Stream DUA script for DSA1 adds a non-specific subordinate reference to DSA 2:

```
modify
      country "AU" / organizationName "Deltawing"
add values
      nonSpecificKnowledge { dsa2-access-point }
;
```

Setting up the root entry

A DSA that is to participate in distributed operations needs certain information in and concerning the root entry. Specifically, it needs:

- A `myAccessPoint` attribute in the root entry to establish its own OSI address.
- *Peer or superior knowledge*. If it is a first-level DSA it needs a set of specific subordinate references to all immediate subordinates of the DIT root entry that are held in other DSAs. Otherwise it needs a `superiorKnowledge` attribute in the root entry to a single nominated superior DSA.

These requirements are mandated by X.500. A ViewDS DSA needs three additional attributes in its root entry:

- An `attributeTypeExtensions` attribute to enable indexing of directory information and other features, as described on page 95.
- A `dsaCollaborators` attribute to establish the list of DSAs with which this DSA can initiate or accept binds. It provides mutual credentials and other information, and is described on page 117.
- An `anonymousPrivilege` attribute to enable anonymous binds (for example, by Access Presence to invoke the GetMyDN procedure).

Every DSA mentioned in a knowledge attribute requires an entry in `dsaCollaborators` if this DSA is to be able to communicate with it. If such an entry is missing, this DSA will return a referral to the other DSA.

myAccessPoint

This single-valued attribute names the DSA and sets up its OSI address. It must be present in the root entry of the DSA if the DSA is to participate in distributed operations. The definition for the attribute is:

```
myAccessPoint    ATTRIBUTE  ::= {
      WITH SYNTAX           AccessPoint
      EQUALITY MATCHING RULEaccessPointMatch
      SINGLE VALUE          TRUE
      NO USER MODIFICATION  TRUE
      USAGE                 dSAOperation
      ID                            ds 12 1 }
```

The attribute syntax `AccessPoint` is described earlier.

Example

The following Stream DUA script assigns the name and presentation address of DSA1, using the access point *`dsa1-access-point`* defined previously:

```
modify
add values
      myAccessPoint dsa1-access-point
;
```

Peer or superior knowledge

First level DSAs

If the DSA holds any naming context that begins immediately below the DIT root, it is a first-level DSA. As such, it must hold a set of specific subordinate references to all immediate subordinates of the DIT root entry held in other DSAs.

These specific subordinate references are created in the normal way. That is, they are created through glue entries subordinate to the root named with the context-prefix name in the other DSA, and contain a `specificKnowledge` attribute with the access point of that DSA.

If the DSA holds every immediate subordinate of the DIT root (e.g. if the DIT is constrained to have only a single such subordinate, say the local country entry), then the DSA is an isolated first-level DSA, and no such specific subordinate references are needed.

NOTE: An alternative permitted by ViewDS is for the root entry to hold a `nonSpecificKnowledge` attribute giving only the access points of other first level DSAs. Such a configuration may, however, not conform with X.500.

ViewDS supports the ability to distribute the functions of a first-level DSA among multiple DSAs. An entry that is the subordinate of the root can be set up to have the `root` bit set in its `dseType`. Hence, other DSAs set up in the same way can share knowledge of that entry in the same way first-level DSAs share knowledge of the root entry. This means that no single DSA needs to hold the master entry for a country, for example, and any number of DSAs can cooperate to share knowledge of the entry.

Example

The following Stream DUA script for DSA 1 will add DSA 3 as a first-level DSA holding the context prefix `C "US"`.

```
entry
      country "US"
with
      dseType {immSupr},
      specificKnowledge { dsa3-access-point } ;
```

Subordinate DSAs

If a DSA is *not* a first-level DSA, it is required to hold a `superiorKnowledge` attribute in the root entry which contains the access point of a single nominated superior DSA that can find entries which this DSA does not hold and whose superiors this DSA does not hold.

The `superiorKnowledge` attribute is defined earlier in this chapter. It is used only in the root entry.

Example

The following Stream DUA script for DSA 2 will add DSA 1 as its superior DSA:

```
modify
add values
      superiorKnowledge dsa1-access-point ;
```

Cross references

A cross reference is a direct reference to a context prefix in another DSA. If present in a DSA, it allows the DSA to more efficiently resolve queries that would otherwise be chained to the superior DSA.

It is created by adding an attribute of type `specificKnowledge` to an entry representing the context prefix, and setting the `dseType` of the entry to `{xr}`. This entry may be a superior (a glue entry) of a locally held naming context, or it may require that creation of a suitable glue entry as a subordinate of an existing glue entry.

Knowledge example

This section summarizes the requirements for configuring knowledge in the three DSAs of the reference example shown in Figure 6 on page 154.

DSA1

DSA1 is a first-level DSA. It must hold:

- Root entry holding `myAccessPoint` with access point of DSA1, `dsaCollaborators` with names and mutual credentials for DSA2 and DSA3, and `anonymousPrivilege`.
- Glue entry `C "US"` with `dseType {subr}` and `specificKnowledge` containing the access point of DSA3.
- Real entry `C "AU"` with schema information for AU.
- Real entry `C "AU" / O "Deltawing"` with schema information for Deltawing (assuming this is the subschema administrative entry)

- Glue entry `C "AU" / O "Deltawing" / OU "Deltawing InfoSystems"` with `dseType {subr}` and `specificKnowledge` containing the access point of DSA2.
- Real entries for all other subordinates of Deltawing and their subtrees.

DSA2

DSA2 is a subordinate DSA. It must hold:

- Root entry holding `myAccessPoint` with access point of DSA2, `superiorKnowledge` containing the access point of DSA1, `dsaCollaborators` with name and mutual credentials for DSA1, and `anonymousPrivilege`.
- Glue entry `C "AU"` with `dseType {immSupr}` and `specificKnowledge` containing the access point of DSA1.
- Glue entry `C "AU" / O "Deltawing"` with schema information for Deltawing (assuming that Deltawing InfoSystems uses the Deltawing subschema and does not set up its own).
- Real entry `C "AU" / O "Deltawing" / OU "Deltawing InfoSystems"`.
- Real entries for all subordinates of Deltawing InfoSystems and their subtrees.

DSA3

DSA3 is a first-level DSA. It must hold:

- Root entry holding `myAccessPoint` with access point of DSA3.
- Glue entry `C "AU"` with `dseType {subr}` and `specificKnowledge` containing the access point of DSA1.
- Real entry `C "US"` with schema information for US.
- Entries for subordinates of the US entry.

Remote aliases

A remote aliases is an alias to an entry in another DSA. When a remote alias is encountered in a directory operation, the directory normally chains the operation to the DSA holding the remote entry. When a remote alias is encountered in the subtree of the base object of a search request, the DSA will normally chain the search request in the same manner, provided the search argument has the `searchAliases` flag set. (Do this with Stream DUA using the `set search aliases` command or by using the search command with `and aliases`.)

This is correct X.500 behaviour, but it causes two serious problems. Each remote alias results in a separate chained search, so a single search request can potentially generate hundreds or thousands of chained searches. And even if there are very few remote aliases, the need to perform a chained search requires the DSA to check every alias within the subtree on every search request to determine whether it is a remote alias; even if most aliases are local aliases, this checking can greatly degrade search performance.

For these reasons, ViewDS will *not* normally check remote aliases within a search subtree, even if the search request specifies that aliases should be checked. Local aliases are always handled correctly. ViewDS will only check those remote aliases for

which the administrator has added the ViewDS-specific operational attribute `remoteAlias`.

It is recommended that the number of remote aliases to which this attribute is added is kept small (at most 10 say).

remoteAlias

`remoteAlias` is a ViewDS-specific operational attribute that can optionally appear within alias entries. If the alias appears in the subtree of a search request that specifies `searchAliases`, then the DSA will chain the search through that alias only if the `remoteAlias` operational attribute is present. Its syntax is currently an empty sequence. It is defined as:

```
remoteAlias ATTRIBUTE ::= {
    WITH SYNTAX             RemoteAlias
    SINGLE VALUE            TRUE
    USAGE                   distributedOperation
    ID                      {vf 18 14} }
}

RemoteAlias ::= SEQUENCE {
    -- reserved for automatically generated cached info
}
```

Replication

ViewDS DSAs support replication of directory data in accordance with the DSA Information Model of X.501, the procedures for distributed operation in X.518, and the replication protocol of X.525.

Familiarity with the concepts of X.525 is assumed in the rest of this chapter.

Restrictions

The following practices are recommended for replication:

- Ensure that the network connection between supplier and consumer DSAs is sufficiently reliable and fast to allow large protocol data units (PDUs) to be transferred successfully. X.525 `updateShadow` PDUs can be very large – typically around 1 MB per 1000 entries updated. In the case of a full refresh, a single PDU holds the data for every entry in the unit of replication; in the case of an incremental refresh, the PDU holds only the data that was changed since the last refresh.
- Ensure that the DSA operational parameter `dots` is set to greater than 1, so that replication activity can proceed without causing user requests to be suspended until replication activity completes.
- For a DSA which is acting as both supplier and consumer, ensure that the DSA operational parameter `optimistic` is set to `off`, and the DSA operational parameter `updates` is set to 2, so that large replication transactions are not constantly aborted due to influence from user updates.
- For a DSA which is acting solely as a consumer, ensure that the DSA operational parameter `updates` is set to 1, so that normal update operations do not interfere with shadow update operations.

- Avoid replication agreements that make use of replication features that ViewDS does not support. In ViewDS, subtree specifications are not supported by shadow consumers (“subordinate completeness” is not implemented) and entries copied with attribute selection applied are treated as complete.
- Do not set up more than about 10 shadowing agreements in a single DSA. Having a large number of shadowing agreements may have a detrimental effect on performance, even when no shadow updates are being processed.

Setting up a shadowing agreement

To set up a shadowing agreement between two ViewDS DSAs:

- Ensure that distributed operations between the DSAs is working – that the DSAs are known to one another via knowledge references and the `dsaCollaborators` attribute, and that a communications path exists between them. *Always establish distributed operations between the DSAs before attempting to set up replication*.
- If the shadow consumer DSA is to be a first-level DSA, ensure its knowledge is set up correctly; do not rely on using replicated data from some other first-level DSA to fill out missing first level knowledge. Consider making the shadow consumer DSA a subordinate DSA in such a case.
- Use Stream DUA to add a value of the `supplierStatus` operational attribute to the root entry of the shadow supplier DSA, and a value of the `consumerStatus` operational attribute to the root entry of the shadow consumer DSA. These operational attributes are described below.

To set up a shadowing agreement between a ViewDS DSA and a non-ViewDS DSA, proceed as above for the ViewDS DSA, and follow the procedure prescribed for the non-ViewDS DSA in its documentation.

Activating an agreement

A shadowing agreement is active as soon as it is added to a DSA. If the agreement specifies an update mode which is `scheduled` and gives a `beginTime`, updates will not begin to flow between the DSAs until `beginTime` is reached; otherwise they begin to flow immediately. In that case it is recommended that the `consumerStatus` attribute be added to the shadow consumer first and the `supplierStatus` attribute be added to the shadow supplier second. This avoids the shadow supplier attempting an initial update operation before the shadow consumer is ready.

Schema changes

Schema changes that include the definition of new attribute types not previously known to the consumer DSA pose special difficulties in replication. The new attribute type definitions must be installed in the shadow consumer DSA before it can know the ASN.1 syntax needed to decode values of those types. Hence, if the new attribute type definitions and entry changes that include the addition of values of the new attribute types are propagated in the same full or incremental shadow update PDU, the consumer DSA will fail to decode the new attributes.

ViewDS therefore requires all shadow updates that involve schema changes which define new attribute types to be propagated in a separate and earlier PDU to the one that adds such attributes to entries. To comply with this, it is recommended that the shadow supplier administrator make the schema changes, wait until the updates have

been propagated successfully to all shadow consumers (by checking the `supplierStatus lastUpdate` time described below), and only then allow values of the new attribute types to be added.

In normal operation, this is assisted by using the recommended supplier-initiated on-change update mode. However, if a full refresh is necessary, or if recent changes are batched for any reason, then the situation can arise where new attribute definitions are included with changes that depend on them. Such updates will fail.

In this case, the shadow consumer DSA's administrator must manually install the new attribute definitions ahead of the shadow update operations being sent. The new attribute definitions can be added anywhere in the shadow consumer DIT, and deleted afterwards.

Replication attributes

There are two ViewDS-specific operational attributes relating to replication which hold the definition of a shadowing agreement and its operational status.

supplierStatus

A value of this multi-valued attribute must be present in the root entry for each shadowing agreement for which this DSA is the shadow supplier. It is ViewDS-specific and the ViewDS representation of a shadowing agreement.

The attribute is added by the administrator to set up a shadowing agreement. Several components are used by the DSA to record changes to the state of the agreement.

The attribute definition is:

```
supplierStatus ATTRIBUTE ::= {
      WITH SYNTAX           SupplierStatus
      EQUALITY MATCHING RULE shadowStatusMatch
      NO USER MODIFICATION  TRUE
      USAGE                 dSAOperation
      ID                    { vf 12 6 }
}
SupplierStatus ::= SEQUENCE {
      consumer                  [0] AccessPoint,
      identifier                [1] INTEGER,
      agreement                 [2] ShadowingAgreementInfo,
      update                    [3] ShadowState,
      othertime                 [4] UpdateWindow OPTIONAL,
      lastUpdate                [5] GeneralizedTime OPTIONAL,
      nextUpdate                [6] GeneralizedTime OPTIONAL,
      onChangeRetry             [7] INTEGER DEFAULT 60 -- seconds --,
      logId                     [8] INTEGER OPTIONAL,
      fullUpdateRequired        [9] BOOLEAN DEFAULT FALSE,
      sessionId                 [10] INTEGER OPTIONAL,
      replicationExclusions    [11]
                SEQUENCE OF OBJECT IDENTIFIER OPTIONAL,
      replicateShadowPlane      [12] PlaneRef OPTIONAL,
      forceIncrementalReplace   [13] BOOLEAN DEFAULT FALSE,
      version                   [14] INTEGER DEFAULT 0,
      lastItemUpdate            [15]
                 GeneralizedTime OPTIONAL -- obsolete --,
      nextUpdateLastItem    [16] ReplicationSequenceNumber OPTIONAL,
```

```
      shadowPlaneRefreshed   [17] BOOLEAN DEFAULT FALSE,
      replicationLogStatus   [18] ReplicationLogStatus OPTIONAL,
      triggerTotalRefresh    [19] TotalRefreshTrigger OPTIONAL
}
```

consumer

The access point of the DSA which is the shadow consumer in the agreement.

identifier

An integer that identifies the shadowing agreement. It corresponds to the operational binding ID in the Directory Operational Binding protocol. It can be chosen arbitrarily, subject to the requirement that the combination of the `consumer` Distinguished Name and the `identifier` are unique.

agreement

This is shadowing agreement information, and it is defined as follows:

```
ShadowingAgreementInfo ::= SEQUENCE {
      shadowSubject      UnitOfReplication,
      updateMode         UpdateMode
                         DEFAULT supplierInitiated : onChange : TRUE,
      master             AccessPoint OPTIONAL,
      secondaryShadows   [2] BOOLEAN DEFAULT FALSE
}
```

shadowSubject

The `shadowSubject` defines the unit of replication, which is a subtree and the information within it. The unit of replication is defined as follows:

```
UnitOfReplication ::= SEQUENCE {
      area                   AreaSpecification,
      attributes             AttributeSelection,
      knowledge              Knowledge OPTIONAL,
      subordinates           BOOLEAN DEFAULT FALSE,
      contextSelection       ContextSelection OPTIONAL,
      supplyContexts         [0] CHOICE {
          allContexts            NULL,
          selectedContexts       SET OF CONTEXT.&id } OPTIONAL
}
AreaSpecification ::= SEQUENCE {
      contextPrefix          DistinguishedName,
      replicationArea        SubtreeSpecification
}
Knowledge ::= SEQUENCE {
      knowledgeType          KnowledgeType ENUMERATED {
                                 master    (0),
                                 shadow  (1),
                                 both      (2) },
      extendedKnowledge      BOOLEAN DEFAULT FALSE
}
AttributeSelection ::= SET OF ClassAttributeSelection
ClassAttributeSelection ::= SEQUENCE {
```

```
      class              OBJECT IDENTIFIER OPTIONAL,
      classAttributes    ClassAttributes DEFAULT allAttributes : NULL
}
ClassAttributes ::=  CHOICE {
      allAttributes          NULL,
      include            [0] AttributeTypes,
      exclude            [1] AttributeTypes
}
AttributeTypes ::=   SET OF AttributeType
```

Where:

`area` Defines the subtree to be replicated.

`contextPrefix` The name of the entry at the top of the naming context to which the subtree belongs.

`replicationArea` Specifies a subtree within that naming context. Note that a shadow consumer only correctly supports an empty `SubtreeSpecification`. If the `base` or `specificationFilter` components are present the shadowed information is used incorrectly.

`attributes` Defines the set of attributes to be shadowed. It is a set of `ClassAttributeSelection`, each member of the set specifying the attributes to be shadowed (in `classAttributes`) for entries whose object class or superclass is specified in `class`. If `class` is absent, the `ClassAttributeSelection` applies to all entries. `ClassAttributes` specifies either `allAttributes` (all user attributes and collective attributes), `include` (an explicit list of attributes to include), or `exclude` (all user attributes except those explicitly listed). The specification of an attribute supertype implicitly includes any subtypes.

NOTE: Access Presence requires the ViewDS-specific operational attributes `userName` and `privilege` to be present in an entry for binding and to occur and updates to be allowed (see the *Technical Reference Guide: User Interfaces*). If a ViewDS user binds to a DSA that holds a replicated copy of the user's entry and these attributes are missing from the replicated information, the bind will fail. The attributes will be replicated only if explicitly listed in `attributes`. This is recommended.

`knowledge` Defines whether or not to include subordinate knowledge of either master or shadowed naming contexts: `knowledgeType` is one of `master` (master naming context knowledge only), `shadow` (shadow naming context knowledge only), or both (both types of knowledge). `extendedKnowledge` if true specifies that all subordinate knowledge references are to be included even if they fall outside `replicationArea`; they must, however, still be subordinate to `contextPrefix`.

updateMode

Defines the update strategy for the shadowing agreement. It is defined as shown below.

```
UpdateMode ::=   CHOICE {
              supplierInitiated  [0] SupplierUpdateMode,
             consumerInitiated   [1] ConsumerUpdateMode
          }
SupplierUpdateMode ::=  CHOICE {
      onChange              BOOLEAN,
         scheduled           SchedulingParameters
}
ConsumerUpdateMode ::=  SchedulingParameters
SchedulingParameters ::= SEQUENCE {
      periodic              PeriodicStrategy OPTIONAL,
              -- must be present if othertimes is set to FALSE -
         othertimes         BOOLEAN DEFAULT FALSE
}
PeriodicStrategy ::= SEQUENCE {
      beginTime             Time OPTIONAL,
      windowSize            INTEGER,
         updateInterval      INTEGER
}
Time  ::= GeneralizedTime
      -- as per clause 34.3 b) and c) of Recommendation X.208/ISO 8824
```

Where:

`supplier Initiated`	Specifies that the supplier initiates updates. If it is `onChange`, the supplier initiates an update whenever a change occurs; if it is `scheduled`, the supplier initiates an update in accordance with the specified scheduling parameters.
`consumer Initiated`	Specifies that the consumer initiates updates in accordance with the specified scheduling parameters.
`Scheduling Parameters`	Specifies a periodic strategy; if `othertimes` is `true`, it specifies that updates may also occur at other times. `PeriodicStrategy` specifies a `beginTime`, the start time of the first window, a `windowSize`, the length of the update window in seconds, and an `updateInterval`, the interval between the start of one update window and the start of the next in seconds. If `beginTime` is not specified, the update strategy starts at the time the shadowing agreement is activated.

master

Specifies the master DSA for the shadowing agreement. It may be omitted if the master is the DSA itself – that is, it only needs to be present when the shadowing agreement is a secondary shadowing agreement.

secondaryShadows

Defines whether the consumer DSA can create secondary shadow agreements for the replicated data.

update

The status of the shadow agreement. A value of ShadowState as defined below. This parameter is maintained by the DSA. When a shadow agreement is first created (for example, as an administrative action), it is automatically set to fullUpdateRequired.

`ShadowState` is defined as follows:

```
ShadowState ::= ENUMERATED {
      idle               (0),
      fullUpdateRequired (1),
      updatePending      (2),
      fullUpdatePending  (3),
      inactive           (4)   }
```

othertime

`othertime` is:

```
UpdateWindow ::= SEQUENCE {
      start              Time,
      stop               Time }
```

lastUpdate

The time at which the last update occurred. This value is maintained by the DSA.

nextUpdate

The time at which the next update is due. This value is maintained by the DSA.

onChangeRetry

When an on-change update cannot be initiated, this is the period (in seconds) that the DSA waits before it retrying the update. This might be necessary, for example, when the consumer DSA cannot be contacted.

logID

Used by the DSA to record the identifier of the log file it uses to store incremental update information. This field should be omitted when creating a new shadowing agreement.

fullUpdateRequired

Used by the DSA to record that a full update is required due to an internal inconsistency. Omit this field when creating a new shadowing agreement.

sessionId

Used by the DSA to record internal information relating to the update session. This field should be omitted when creating a new shadowing agreement.

replicationExclusions

An optional field comprising of a list of object identifiers of object classes or attribute types which should not be included in the replication PDU. This was included to allow subentries and operational attributes to be excluded from the replication PDU despite the X.500 standard indicating they should be automatically included. This was added to assist in interworking with DSAs which do correctly handle subentries and some operational attributes in the replication PDUs.

replicateShadowPlane

An optional field used to set up a secondary shadow replication agreement. This field should be set on a secondary supplier DSA to the plane reference information of the shadow plane that this DSA receives from another supplying DSA. This plane reference is composed of the name of the supplying DSA and the agreement identifier of the shadowing agreement for this replicated information.

```
PlaneRef ::= SEQUENCE {
     dsaName                 Name,
     agreementID             AgreementID }
```

forceIncrementalReplace

An optional Boolean field that can be used to force the replication of incremental changes to use the replace option in the `incrementalRefresh` PDU instead of using the other options permitted by the X.500 standards. This option is intended to overcome interworking problems with DSAs which only support the replace behaviour in the `incrementalRefresh` PDU.

version

This content has not yet been developed.

lastItemUpdate

This content has not yet been developed.

nextUpdateLastItem

This content has not yet been developed.

shadowPlaneRefreshed

Used to flag that a total refresh was completed for the shadow plane this agreement is supplying to a secondary consumer. This could happen while a secondary total refresh is already in progress, so the fullUpdateRequired flag is not safe to use.

replicationLogStatus

replicationLogStatus is a read-only property identifying the sequence numbers currently in use in the replication log. Attempts to modify this field will be ignored.

triggerTotalRefresh

An optional configurable threshold that will cause a supplier DSA to automatically initiate a total refresh for an agreement if the number of outstanding incremental updates for that agreement exceeds the threshold. This field has a type of `TotalRefreshTrigger` which is defined as follows:

```
TriggerTimeOfDay ::= TIME (SETTINGS "Basic=Time Time=HMS
Midnight=Start")
TotalRefreshTrigger ::= SEQUENCE {
      Threshold                [0] ReplicationSequenceNumber,
      Window                   [1] SEQUENCE {
                Begin      TriggerTimeOfDay,
                End        TriggerTimeOfDay
      } OPTIONAL
}
```

Where:

`threshold` Defines the number of outstanding incremental updates that will cause an automatic total refresh to be triggered.

`window` If present, indicates that an automatic total refresh will only be triggered during the defined time period. If the begin time is later than the end time, the defined time period is assumed to start on one day and finish on the next day (bridging midnight). The "begin" and "end" fields represent times of day in the format "HH:MM:SS" with an optional time zone ("Z" or "+HH:MM" or "-HH:MM"). If the time zone is not provided, the times are assumed to be in the local time of the server.

consumerStatus

A value of this multi-valued attribute must be present in the root entry for each shadowing agreement for which this DSA is the shadow consumer. It is ViewDS-specific and the ViewDS representation of a shadowing agreement. A value of the attribute is added to set up a shadowing agreement; a number of components in the value are for use by the DSA to record changes to the state of the agreement. The definition of the attribute is:

```
consumerStatus ATTRIBUTE ::= {
      WITH SYNTAX           ConsumerStatus
      EQUALITY MATCHING RULE shadowStatusMatch
      NO USER MODIFICATION  TRUE
      USAGE                 dSAOperation
      ID                    { vf 12 7 }
}

ConsumerStatus ::= SEQUENCE {
      supplier           [0] AccessPoint,
      identifier         [1] INTEGER,
      agreement          [2] ShadowingAgreementInfo,
      coordinate         [3] ShadowState,
      othertime          [4] UpdateWindow OPTIONAL,
      lastUpdate         [5] GeneralizedTime OPTIONAL,
      fullUpdateRequired [9] BOOLEAN DEFAULT FALSE,
      sessionId          [10] INTEGER OPTIONAL,
      supplierKnowledge  [11] BOOLEAN DEFAULT TRUE,
      nonSupplyingMaster [12] AccessPoint OPTIONAL,
      supplierIsMaster   [13] BOOLEAN OPTIONAL
          -- supplierIsMaster is effectively DEFAULT TRUE unless
          -- nonSupplyingMaster is present in which case it is ignored
          -- (and treated as FALSE).
}
```

Aside from the following, the components of this syntax are the same as those of `SupplierStatus` (see page 164).

`supplierKnowledge` is a flag to control whether a `supplierKnowledge` attribute value should be automatically created in the context prefix of the replicated area. By default, this flag is `TRUE`. This flag may be required where it is not desirable for the replica to chain requests to the supplier.

`nonSupplyingMaster` is an optional field which allows an administrator of a secondary shadow DSA to provide information of the master of the replicated area (as opposed to the secondary supplier) which may be used in constructing the supplierKnowledge attribute value.

`supplierIsMaster` is an optional field used to indicate that the supplier is not the master of the replicated information. This field is assumed to have the value `TRUE` unless the `nonSupplyingMaster` is specified, in which case this field is assumed to be `FALSE`. The purpose of this field is to indicate that the supplier is not the master when the actual master DSAs access point is not specified in the `nonSupplyingMaster` field (for example, because it is not known or should not be referenced directly by this DSA).

Converting shadow into master

ViewDS can convert a shadow copy of a DIT, obtained through replication, into a master DSA. This capability should be used with care as it can lead to loss of data and cannot be reversed without discarding all the replicated information and completing a total refresh from the original supplier of the replicated information. However, this ability can be useful to provide a form of fail-over in a disaster recovery plan.

In the event that this ability is going to be used, the consumer DSA which may be converted to a master in the future should not master any data itself and should be configured to replicate as much as the supplier as possible. Any filtering of attributes or object classes will result in this information being unavailable if the consumer takes over the role as master in the replication setup.

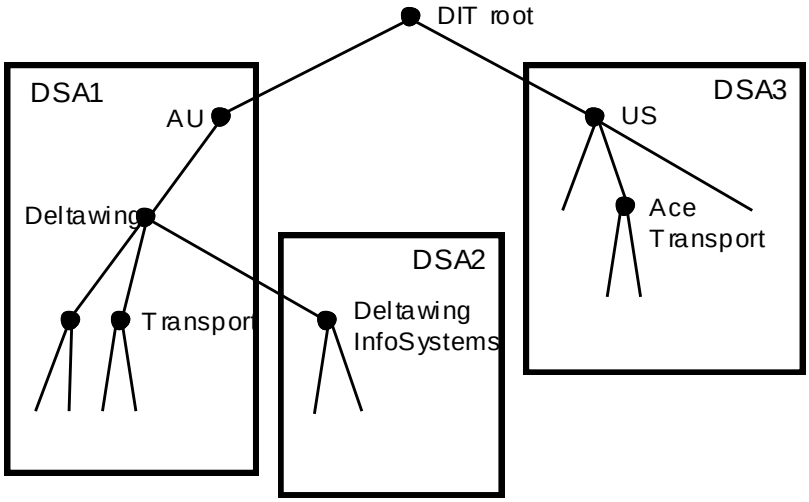
The actual process of converting a shadow DSA into a master DSA requires the following steps. These steps can be carried out on an active system, requiring no interruption to the services offered by the shadow DSA.

1. Remove existing master plane information, such as immediate superior reference and superior reference, from the shadow DSA.
2. Remove the `planeReference` attribute from the root entry of the shadow plane. This attribute is used to indicate the plane is a shadow not a master. To remove this attribute requires use of the `manageDSAITPlaneRef` in the service controls with the DSA name of the supplier and the `agreementID.identifier` field used to indicate which plane to modify. The request must also include the `schemaChecking` service control with a value of `ignoreUserModifiableFlag`. Removing the `planeReference` value will automatically remove: the `consumerStatus` value used to construct this shadow plane; the `supplierKnowledge` value in the context prefix; and clean up any references to this plane in `supplierStatus` (used for secondary shadowing).

More information and assistance for using this ability may be obtained from your ViewDS vendor.

Replication example

This section gives an example of how to set up two shadowing agreements between two of the DSAs in the reference example (reproduced below). The roles of the two DSAs are reversed in the second shadowing agreement.



Assume DSA1 is to be a *shadow consumer* for an organization `C "US" / O "Ace Transport"` mastered in DSA3. Therefore, DSA3 is shadow supplier for this subtree. Updates are to be in accordance with the default – that is, supplier-initiated and on change. The agreement has identifier 1.

Change to DSA1

Make this change to DSA1:

```
# Agreement 1 (consumer to DSA3 re Ace Transport)
modify
      add consumerStatus {
          supplier dsa3-access-point,
          identifier 1,
          agreement {
              shadowSubject {
                  area {
                      contextPrefix {
                          / countryName "US"
                          / organization "Ace Transport"
                      },
                      replicationArea { }
                  },
                  attributes {
                      {} -- all user attributes, plus: --,
                      { classAttributes include : {
                          userName, privilege, attributePresentation,
                          objectClassPresentation, searchOptions,
                          duaBanners, defaultEntitlement,
                          userEntitlement, userConfig
                      }}
                  }
              }
          },
          coordinate fullUpdateRequired
      };
```

Change to DSA3

Make this change to DSA3:

```
# Agreement 1 (supplier to DSA1 re Ace Transport)
modify
      add supplierStatus {
          consumer dsa1-access-point,
          identifier 1,
          agreement {
              shadowSubject {
                  area {
                      contextPrefix {
                          / countryName "US"
                          / organization "Ace Transport"
                      },
                      replicationArea { }
                  },
                  attributes {
                      {} -- all user attributes, plus: --,
                      { classAttributes include : {
                          userName, privilege, attributePresentation,
                          objectClassPresentation, searchOptions,
                          duaBanners, defaultEntitlement,
                          userEntitlement, userConfig
                      }}
                  }
              }
          },
          update fullUpdateRequired,
          onChangeRetry 30
      }
;
```

At the same time, assume DSA1 is to be *shadow supplier* to DSA3 for the organizational unit `C "AU" / O "Deltawing" / OU "Transport"`. Updates are to be supplier-initiated and scheduled to begin on 1 Jan 1997 at 2.00 a.m. with a window size of 1 hour and an update interval of 24 hours. The agreement has identifier 3.

Change to DSA1

Make this change to DSA1:

```
# Agreement 3 (supplier to DSA3 re Transport)
modify
      add supplierStatus {
          consumer dsa3-access-point,
          identifier 3,
          agreement {
              shadowSubject {
                  area {
                      contextPrefix {
                          / countryName "AU"
```

```
                        / organizationName "Deltawing"
                        / organizationalUnitName "Transport"
                    },
                    replicationArea { }
                },
                attributes {
                    {} -- all user attributes, plus: --,
                    { classAttributes include : {
                        userName, privilege, attributePresentation,
                        objectClassPresentation, searchOptions,
                        duaBanners, defaultEntitlement,
                        userEntitlement, userConfig
                    }}
                }
            },
            updateMode supplierInitiated:scheduled:{
                periodic {
                    beginTime "19970101020000+1000",
                    windowSize 3600,
                    updateInterval 86400
                }
            }
        },
        update fullUpdateRequired
    } ;
```

Change to DSA3

Make this change to DSA3:

```
# Agreement 3 (consumer to DSA3 re Transport)
modify
      add consumerStatus {
          supplier dsa3-access-point,
          identifier 3,
          agreement {
              shadowSubject {
                  area {
                      contextPrefix {
                          / countryName "AU"
                          / organizationName "Deltawing"
                          / organizationalUnitName "Transport"
                      },
                      replicationArea { }
                  },
                  attributes {
                      {} -- all user attributes, plus: --,
                      { classAttributes include : {
                          userName, privilege, attributePresentation,
                          objectClassPresentation, searchOptions,
                          duaBanners, defaultEntitlement,
```

```
                    userEntitlement, userConfig
                }}
            }
        },
        updateMode supplierInitiated:scheduled:{
            periodic {
                beginTime "19970101020000+1000",
                windowSize 3600,
                updateInterval 86400
            }
        }
    },
    coordinate fullUpdateRequired
};
```

LDAP change log

The LDAP change log is a mechanism to support synchronisation with third-party applications. The log is a directory entry that contains a record of all changes made to the DIT. It does, however, add a significant overhead to every update operation and should only be enabled if required by a third-party application.

The DSA adds records to the log if there is a change-log attribute in the root entry. It creates records as subordinates of the `viewDSChangeLogContainer` entry.

The LDAP change log should also have a clean-up configuration declared to remove obsolete entries. Otherwise, it will continue to grow and eventually exhaust the storage capacity of the ViewDS host.

The entries in the LDAP change log are replicated through DISP if they satisfy the area of replication in a replication agreement. Master change-log entries are only created on a DSA in response to updates to master entries on the same DSA. Updates to shadow entries by DISP do not cause change-log entries to be created on a shadow consumer (although shadow change-log entries may appear).

Updates to change-log entries are allowed (subject to access controls) but they do not cause further change-log entries to be created.

You can manage the LDAP change log through either the ViewDS Management Agent (see its help system for instructions) or through Stream DUA (see below).

Enabling the LDAP change log using Stream DUA

Enabling the LDAP change log involves:

- Creating an LDAP change log container
- Creating an LDAP change log attribute

Creating an LDAP change log container

This container entry must be added manually using Stream DUA:

```
viewDSChangeLogContainer OBJECT-CLASS ::= {
   SUBCLASS OF    { top }
   MUST CONTAIN   { commonName }
   MAY CONTAIN    { viewDSNextChangeNumber | viewDSChangeLogExpiry }
   ID             { 1 3 6 1 4 1 21473 5 6 0 }
```

```
}
viewDSNextChangeNumber ATTRIBUTE ::= {
   WITH SYNTAX               INTEGER
   EQUALITY MATCHING RULE    integerMatch
   ORDERING MATCHING RULE    integerOrderingMatch
   SINGLE VALUE              TRUE
   NO USER MODIFICATION      TRUE
   USAGE                     directoryOperation
   ID                        { 1 3 6 1 4 1 21473 5 18 16 }
}
viewDSChangeLogExpiry ATTRIBUTE ::= {
   WITH SYNTAX               ChangeLogExpiry
   EQUALITY MATCHING RULE    allComponentsMatch
   SINGLE VALUE              TRUE
   USAGE                     directoryOperation
   ID                        { 1 3 6 1 4 1 21473 5 18 17 }
}
ChangeLogExpiry ::= SEQUENCE {
   timeToLive [0] INTEGER (1..MAX),
   -- Minimum number of seconds a change-log entry is retained
   -- before being deleted
   cleanupSchedule [1] PeriodicStrategy
   -- Change-log entries that exceed their timeToLive are only
   -- deleted during the time period defined by the PeriodicStrategy
}
```

The name and location of the container entry is not restricted, but by convention it is a first-level entry with the `commonName` of `changelog`.

The `viewDSNextChangeNumber` is automatically amended by the DSA and cannot be modified manually. The `viewDSChangeLogExpiry` is optional, but highly recommended because it specifies when obsolete change-log entries will be removed. Without it, the change-log entries will accumulate indefinitely.

The `timeToLive` should be high enough to allow third-party applications sufficient time to fetch entries before they expire. It should also allow for the possibility of an application failing overnight and not being restarted until the next day. (There is no harm in retaining change-log entries for days, or even weeks.)

The `PeriodicStrategy` ASN.1 type is defined by DISP. The removal of expired entries is an expensive operation; hence, the time period specified in `cleanupSchedule` should occur when the DSA is expected to have a light load.

Creating an LDAP change log attribute

The LDAP change log is active when the DN of the `viewDSChangeLogContainer` entry is stored in the `changelog` attribute in the root entry.

```
changelog ATTRIBUTE ::= {
   WITH SYNTAX DistinguishedName
   EQUALITY MATCHING RULE distinguishedNameMatch
   SINGLE VALUE TRUE
   ID { 2 16 840 1 113730 3 1 35 }
}
```

Reinitialising the LDAP change log using Stream DUA

To reinitialize the LDAP change log, remove the `changelog` attribute and the entire subtree below the `viewDSChangeLogContainer` entry; then add a new `viewDSChangeLogContainer` entry and `changelog` attribute.

Access Proxy

ViewDS *Access Proxy* provides a Certificate Lookup Service that complies with TSCP specifications (see http://www.tscp.org). It allows email clients to obtain digital certificates in the same organization or in external collaborating organizations.

For an overview and installation instructions, see the *ViewDS Installation Guide: Access Proxy*.

A DSA configured for Access Proxy includes a knowledge reference to each End-User Certificate Repository Service (EUCRS) plus the following operational attribute.

CertificateRepositoryService

This operational attribute stores the email domains served by a particular EUCRS.

```
CertificateRepositoryService ::= SEQUENCE {
  emailDomains       [0] SEQUENCE SIZE(1..MAX) OF UTF8String
}

certificateRepositoryService ATTRIBUTE ::= {
  WITH SYNTAX        CertificateRepositoryService
  EQUALITY MATCHING RULE allComponentsMatch
  SINGLE VALUE           TRUE
  USAGE                  distributedOperation
  ID                     id-viewDS-doa-certificateRepositoryService
}
```

The `emailDomains` component contains one or more full DNS name. These are the possible host names of the email addresses for users whose entries are in the EUCRS indicated by the DSA's `specificKnowledge`.

Appendix A
Stream DUA

This appendix describes Stream *Directory User Agent* (DUA) commands and notation.

Stream DUA commands

add

Deprecated.

Adds synonyms, noise words, or truncated words:

```
add = "add" type ( add-synonym | add-noise-word |
     add-truncated ) [ options ] ";"
```

This command is still supported to assist when upgrading to ViewDS version 7.1. The synonyms, noise words and truncated words are now defined in the following operational attributes:

- `viewDSSynonyms`
- `viewDSNoiseWords`
- `viewDSTruncatedWords`

When upgrading to version 7.1, the add command results in values being added to the above operational attributes in the root entry.

The operational attributes must then be manually transferred into the subschema subentry(s) when the upgrade is complete. While the operational attributes in the root entry would affect the DSA, and would be used in indexing, they would not be replicated or transferred in naming-context information for distributed operations to other DSAs. Moving them to subschema subentry(s) ensures consistent approximate-matching behaviour across all DSAs.

add-synonym

Adds a new synonym pair to `viewDSSynonyms` in the root entry. The position of a value as the left or right component of a pair is not significant.

```
add-synonym     = "synonym" value "=" value
```

add-noise-word

Adds `value` as a noise word for the given `type`. Noise words are keywords to be ignored when indexing or keyword-matching an attribute value, and are omitted from the automatic abbreviations which the DSA forms for some attributes.

```
add-noise-word = "noise" "word" value
```

add-truncated

Adds `value` as a truncated word for the given `type`. Truncated words are preferred truncations for keywords to be used by the DSA when building the `hierarchyName` attribute (see *Technical Reference Guide: User Interfaces*) or building automatic abbreviations of the attribute value.

```
add-truncated  = "truncated" "word" value
```

add notes

To ensure that the correct values can be indexed, noise words and truncated words *must* be specified before the database is loaded with data. Adding a new noise word or truncated word to a loaded database will corrupt the indexes.

The normal order for loading a database from `dib.*` files loads the files in the correct order: `dib..root`, then `dib..words` (the noise-words file), then schema and entry information.

Adding noise words or truncated words for user-defined attributes presents a problem. The syntax for the user-defined attributes is normally defined in the schema file and is therefore unknown to the DSA when it is processing the `dib..words` file. In this situation, add `attributeTypes` definitions directly to the root entry for user-defined attributes for which noise words or truncated words will be added.

NOTE: The files `dib..synonyms` and `dib..words` are deprecated. The information here about the add command only applies when upgrading to ViewDS version 7.1.

add examples

```
add givenName synonym "bill" = "William"
add organizationName noise word "and"
add organizationalUnitName truncated word "Div" # Division
```

assign

Assigns values to the DSA's operational parameters:

```
assign = "assign" parameter "=" value
     { parameter "=" value } ";"
```

The command sets each `parameter` in the argument list to the supplied `value`. It changes the value currently used by the DSA and saves the new value in the DSA's `cmsrv.cfg` file.

You can see the current values of the operational parameters using the `display` command. The DSA's operation parameters are described in *Chapter 3*.

This command is an alternative to the DSA Controller command `setwrite` (see page 10).

bind

Establishes a connection to the DSA using the authentication credentials provided:

```
bind = "bind" [ "with" ]
      [ simple-credential | strong-credentials ]
      [ "protocol" protocol ] "; "
simple-credentials =
      [ "username" username]
      [ "password" password ]
```

```
strong-credentials =
      { ”username” name | ”certificate” path }
      ”key” path ”to” name
```

An explicit bind is generally not required – Stream DUA will automatically bind to the DSA using the default bind credentials when necessary.

Simple authentication

A bind using simple authentication can be made using the simple-credentials form of the command. This takes the form of a username/password authentication.

Without any arguments, a simple-authentication bind is made with the default settings. The default settings can be modified using the `set` command with the `-u`, `-p` and `-a` options (see page 196).

Strong authentication

A bind using strong authentication can be made using the strong-credentials form of the command. This requires a Distinguished Name (DN) and secret key.

The DN can be provided in two ways:

- using the keyword `username` – the X.509 certificate matching the provided secret key must be present in the entry of the user identified by this DN.
- by identifying a user certificate – the DN is assumed to be the subject name in the certificate. The DN of the DSA must also be provided (using the `to` keyword) to form the bind token. Where the `capath` configuration-file parameter is provided, it may be used to construct the certification path for the user's certificate in the bind token.

The **secret key** must be provided using the keyword `key`, which identifies a file containing the PKCS-8 encoded private key.

For more information, see *Strong authentication* on page 112.

checkpoint

Renames the current query and update log files (the DSA starts writing log information to new log files):

```
checkpoint = “checkpoint” [ options ] “;”
```

The command renames each log file by appending the filename with a numeric extension. The extension is a hex encoding of a timestamp plus the number of times a checkpoint has occurred since the DSA was started. The following example is the filename for a check-pointed update log:

```
ulog-839e3fd4510
```

This naming convention allows filenames to be sorted according to when they were created.

The action of check-pointing the log files also occurs automatically whenever the `save` command is used; and when a `dump` command is invoked with a base object of the *root* entry.

compare

Tests whether a given value exists in an entry:

```
compare = "compare" name "to" type "=" value [ options ] ";"
```

The `name` is the full DN for the entry to be compared. The command returns either `true` or `false` according to whether the entry has an attribute of the given type with the given value.

For example:

```
compare {
      organizationName "Deltawing"
      / commonName "Fred Smith"
}
to
      userPassword = '414243444546'H
;
```

This example compares the `userPassword` in the entry { `organizationName "Deltawing" / commonName "Fred Smith" }` with the supplied value `'414243444546'` (that is, `"ABCDEF"`), returning `true` if the password matches.

delete

Deletes one or more entries from the database:

```
delete = "delete" [ "subtree" ] name [ content ]
         [ options ]";"
```

The `name` is the full DN of the entry to be deleted.

The `content` of the entry may be listed but is ignored. This facility is provided for the DSA update logs which will list the attributes of each deleted entry so that the operation can be reversed manually.

Only entries without subordinates can be deleted unless the keyword `subtree` is declared. If `subtree` is declared, Stream DUA deletes the entire subtree, which involves a series of searches. For these searches to succeed the DSA's `sizelimit` and `timelimit` must be high enough to allow all immediate subordinates to be retrieved in a single operation.

For example:

```
delete {
      organizationName "Deltawing"
      / organizationalUnitName "Finance"
};
```

This example deletes one entry, `{ organizationName "Deltawing" / organizationalUnit "Finance" }`.

display

Displays the DSA's status and the settings of operational parameters:

```
display = "display" ";"
```

dsa

Opens or closes the DSA's database or terminate the DSA:

```
dsa = "dsa" { "close" | "open" | "terminate" } ";"
```

The arguments are:

- `open` – opens the database if it is closed, and does nothing if it is already open.
- `close` – closes the database if it is open, and does nothing if it is already closed.
- `terminate` – terminates the DSA if it is running.

dump

Dumps the contents of a subtree as Stream DUA entry commands, LDIF content records, ELDIF content records, DSML version 1, or DSML version 2:

```
dump  = "dump" name
          [ key ]
          [ destination ]
          [ format ]
          [ fallback ]
          [ options ] ";"
key           = "with" "key" string
destination   = "to" path
format = "as" ("ldif" | "eldif" | "sdua-format" | "dsml-v1" | dsml-v1-
all-in-one" | "dsml-v2" | "dsml-v2-all-in-one" )[ "records" ]
fallback      = "with" "fallback" ("transfer-gser" | "transfer-rxer" |
"binary")
```

The command's arguments are:

`name`	The DN of the base entry for the subtree to be dumped.
`key`	If specified, this string is the key used to encrypt the `userPassword` attribute. If not specified, the default key is used.
`destination`	If specified, the DSA dumps to the **`path`**. (The `path` string must be enclosed in quotes.) If not specified, the DSA dumps to the path specified by the `dumpdir` parameter in the configuration file.
`format`	If specified, the DSA dumps in the format defined in the command. If not specified, the DSA dumps in Stream DUA format. (ELDIF is similar to LDIF, except that RXER encoded values are printed as text rather than in a BASE64 format. Note that ELDIF might not be compatible with non-ViewDS implementations.)

`Fallback` If specified, the DSA dumps in the format defined in the command. If not specified, the DSA dumps in Stream DUA format.

(ELDIF is similar to LDIF, except that RXER encoded values are printed as text rather than in a BASE64 format. Note that ELDIF might not be compatible with non-ViewDS implementations.)

Dumping in DSML allows information within ViewDS to be exported for use by XML enabled applications that support DSML. Note that DSML dumps cannot be reloaded into ViewDS as we only support the process of exporting DSML, not importing it.

The `dsml-v1` and `dsml-v2` format options will produce a set of dib files, each containing the DSML representation of up to 1000 entries each.

The `dsml-v1-all-in-one` and `dsml-v2-all-in-one` format options can be used to create a single file holding all of the dumped data.

dump notes

If the base entry for a dump is the DIT root and the directory data is being dumped in Stream DUA format, data is dumped to the following files:

File	Dumped to the file
`dib..root`	An `empty (filling)` command followed by a dump of the root entry.
`dib.00000`	All other entries are dumped to numbered files, starting with `dib.00000` and new a file created every 1000 entries. To allow fast reloading, the `empty (filling)` command is written at the start of the first `dib` file and a `fill` command at the end of the final file.

This command is only available to the super-administrator.

dump example

```
dump {/};
```

empty

Removes all entries from the database and leaves an empty root entry.

```
empty   = "empty" [ filling ] ";"
filling = "for" "filling"
```

The `filling` argument only has effect with the ViewDS fast-load utility (see page 11) and causes the DSA to defer building database index files until it receives a `fill` command (see page 185).

empty notes

When loading a database there is usually no need to use the `empty` command explicitly. This is because when the database is dumped from the root entry, the DSA encloses the dumped data between two commands:

- `empty (filling)` at the start of the first dump file
- `fill` at the end of the last dump file

Use the `empty` command with caution as it destroys all data in the database.

empty example

```
empty;
```

entry

The keyword `entry` is a synonym for the `insert` command (see page 185).

exit

Exits an interactive Stream DUA session:

```
exit = "exit"
```

It is equivalent to pressing Control-D.

fill

Builds database indexes:

```
fill = "fill" ";"
```

Use this command when building has been deferred by an `empty (filling)` command. Both commands only have an effect when the DSA is running as `vfload`.

The `empty (filling)` command switches the DSA into a mode where it defers building the database index files until a subsequent `fill` command is given. The `fill` command is normally given only after all entries have loaded. Deferring index building in this way results in a much faster load of the database.

A `fill` command at an inappropriate time (that is, after the database is indexed) is ignored.

With large a database, a `fill` command can take a long time to complete.

Until a `fill` command is given, the DSA will refuse to start up other than as `vfload`, and will usually return an empty result set in response to a `search` operation.

insert

Inserts a new entry in the database:

```
insert = ( "insert" | "entry" ) name
         [ content ] [ options ] ";"
```

The `name` is the full DN for the new entry.

The list of attributes given by the `content` argument should include the *object class* of the entry. For object classes that are subclasses, all object classes below `top` need to be supplied. If the object class is not given, the entry becomes an entry of `dseType {glue}` (see *Chapter 7*).

The keyword `entry` is a synonym for `insert`.

insert example 1

```
insert       {
             organizationName "Deltawing"
             / organizationalUnit "Finance"
}
with  {
          objectClass organizationalUnit,
          manager "Fred Smith",
```

```
          location "123 Collins Street, Melbourne 3000",
          telephoneNumber "(03)555-1234"
};
```

This example adds the organizational unit { `organizationName "Deltawing" / organizationalUnit "Finance" }` to the DIT and supplies values for the `objectClass`, `manager`, `location` and `telephoneNumber` attributes. The DSA will also add the attribute `organizationalUnit` and give it the value `Finance`. The superior entry { `organizationName "Deltawing" }` must already exist in the DIT.

insert example 2

```
insert        {
              organizationName "Deltawing"
              / organizationalUnit "Finance"
              / commonName "Fred Smith"
}
with  {
          objectClass organizationalPerson
          surname "Smith",
          givenName "Fred",
          location "123 Collins Street, Melbourne 3000",
          telephoneNumber "(03)555-5678"
};
```

This example adds a new organizational person to the DIT, and supplies values for the `objectClass`, `surname`, `givenName`, `location` and `telephoneNumber` attributes. The new entry will also be given the `commonName` attribute.

ldap-add

Uses LDAP protocol to insert a new entry in the database:

```
ldap-add = "ldap" "add" ldapdn
           [ ldap-content ]
           [ controls ] ";"
```

The `ldapdn` is the full DN for the new entry.

The list of attributes given by the `ldap-content` argument should include the *object class* of the entry. For object classes that are subclasses, all object classes below `top` need to be specified.

ldap-add example

```
ldap add "ou=Finance,o=Deltawing"
with  {
          objectClass "organizationalUnit",
          manager "Fred Smith",
          location "123 Collins Street, Melbourne 3000",
          telephoneNumber "(03)555-1234"
};
```

This example adds the organizational unit `"ou=Finance,o=Deltawing"` to the DIT and supplies values for the `objectClass`, `manager`, `location` and `telephoneNumber` attributes. The DSA will also add the attribute `ou (organizationalUnit)` and give it the value `Finance`. The superior entry `"o=Deltawing"` must already exist in the DIT.

ldap-compare

Uses LDAP protocol to determine whether a given value exists in an entry:

```
ldap-compare = "ldap" "compare" ldapdn "to"
                    ldap-type "=" ldap-value
                        [ controls ] ";"
```

The `ldapdn` is the full DN for the entry to be compared.

Either `true` or `false` is returned according to whether the entry has an attribute of the specified type with the specified value.

ldap-compare example

```
ldap compare
    "cn=Fred Smith,o=Deltawing"
to
    userPassword = '414243444546'H;
```

This example compares the `userPassword` in the entry `"cn=Fred Smith,o=Deltawing"` with the supplied value `'414243444546'` (that is, `"ABCDEF"`). It returns `true` if the password matches.

ldap-delete

Uses LDAP protocol to remove an entry from the database:

```
ldap-delete  = "ldap" "delete" ldapdn
                [ ldap-content ]
                    [ controls ] ";"
```

The `ldapdn` is the full DN of the entry to be deleted. Only entries without subordinates can be deleted.

For example:

```
ldap delete
    "ou=Finance,o=Deltawing"
;
```

ldap-modify

Uses LDAP protocol to modify the named entry:

```
ldap-modify  = "ldap" "modify" ldapdn
                [ ldap-changes ]
                    [ controls ] ";"
```

The `ldapdn` is the full DN of the entry to be modified. Modifications are performed in the order they appear in the command.

```
ldap-changes = [ "with" ] "changes" "{"
                    [ ldap-change
                    { "," ldap-change } ]
                        "}"
ldap-change      = change-add
                    | change-delvals
                    | change-delatt
                        | change-replace
change-add       = "add" ("attribute" | "values")
```

```
                          ldap-type { ldap-value }
change-delvals   = "delete" "values"
                          ldap-type { ldap-value }
change-delatt= "delete" "attribute"
                          ldap-type { ldap-value }
change-replace   = "replace" ldap-type
                     [ old-values ]
                          [ "add" { ldap-value } ]
old-values      = "(" "delete" { ldap-value } ")"
```

Each `ldap-change` is applied to the entry in turn.

The four alternatives are:

- `change-add` – adds the attribute of the specified type if it does not exist in the entry. Otherwise, it adds the listed values to the existing attribute. The choice of keyword, `attribute` or `values`, has no significance.
- `change-delvals` – removes only the listed values from the attribute of the specified type. If no values are listed the change is ignored.
- `change-delatt` – removes the attribute of the specified type. The values of the removed attribute may be listed but are ignored by Stream DUA. This facility is provided to allow reversal of the change recorded in the Stream DUA format update logs.
- `change-replace` – removes all the current values of the attribute of the specified type then adds the listed values, if specified. It is equivalent to a `change-delatt` followed by a `change-add` for the same attribute type. The removed values may be listed with `old-values` but these are ignored by Stream DUA. This is provided to allow reversal of the change recorded in the Stream DUA format update logs.

ldap-modify example

```
ldap modify
    "cn=Fred Smith,o=Deltawing"
with changes {
    remove values telephoneNumber "(03)543 2109",
    add values telephoneNumber "(03)543 9012",
    add attribute employeeNumber "1234" "9822"
};
```

This example modifies the entry `"cn=Fred Smith,o=Deltawing"`. It modifies the value of `telephoneNumber` and adds two new values of `employeeNumber`. Note that the `cn (commonName)` attribute value `Fred Smith` may not be modified with this command. You must use a rename command (for example, `ldap-rename`).

ldap-move

Uses LDAP protocol to move an entry (and any subordinates) to a new position in the DIT:

```
ldap-move = "ldap" "move" ldapdn "to" ldapdn
            [ delete-old ] [ controls ] ";"
```

The first `ldapdn` is the full DN of the entry to be moved; and the second `ldapdn` is the new name for the entry after the move.

ldap-move example

```
ldap move    "cn=Fred Smith,ou=Finance,o=Deltawing"
to             "cn=Fred Smith,ou=Marketing,o=Deltawing" ;
```

This example moves the entry `Fred Smith` so that it becomes a subordinate of the entry `"ou=Marketing,o=Deltawing"`.

ldap-rename

Uses the LDAP protocol to rename an entry. It only changes an entry's Relative Distinguished Name (RDN) and not its position in the DIT.

```
ldap-rename = "ldap" "rename" ldapdn
                  "to" relative-ldapdn
                  [ delete-old ]
                  [ controls ] ";"
relative-ldapdn = string | "(" rdn ")"
```

The `ldapdn` is the full DN of the entry to be renamed.

The `relative-ldapdn` is the new RDN of the entry. The new RDN can be represented in the conventional character string format for an LDAP RDN, surrounded by quotes (for example, as a `string`); or in Stream DUA format, `rdn`, surrounded by parentheses.

ldap-rename example

```
ldap rename
    "cn=Fred Smith,o=Deltawing"
to
    "cn=Fred R. Smith"
delete old
;
```

This example renames the entry `Fred Smith` to `Fred R. Smith` and deletes the old value from the `cn (commonName)` attribute.

ldap-search

Uses LDAP protocol to search the database.

```
ldap-search = "ldap" "search" ldapdn
                search-arguments
                    [ controls ] ";"
search-arguments = [ scope ]
                [ "and" ldap-aliases ]
                "for" ldap-filter
                    [ ldap-selection ] limits
ldap-aliases = "neverDerefAliases"
                | "derefInSearching"
                | "derefFindingBaseObj"
                    | "derefAlways"
```

The `ldapdn` is the DN for the base entry from which to search.

The `ldap-aliases` argument specifies whether aliases in the search subtree and/or the base entry name are dereferenced. Aliases in the search subtree are

dereferenced if `derefInSearching` or `derefAlways` is specified. Aliases are dereferenced in locating the base entry if `derefFindingBaseObj` or `derefAlways` is specified. The default, if `ldap-aliases` is not specified, is `derefAlways`.

ldap-filter

The keyword `for` introduces a filter to apply to the search. An `ldap-filter` is a Boolean expression of `ldap-filter-item`. The `not` operator has highest precedence followed by `and` then `or`. Parentheses may be used to alter the order of evaluation.

```
ldap-filter = "(" ldap-filter ")"
                | ldap-filter-item [ "dnatt" ]
                | "not" ldap-filter
                | ldap-filter "and" ldap-filter
                | ldap-filter "or" ldap-filter
```

A `ldap-filter-item` is an assertion about the attribute values belonging to an entry. The keyword `dnatt` specifies that the assertion applies to the attribute values of the entry's DN addition to the normal attribute values.

ldap-filter-item

```
ldap-filter-item = ldap-type "=" ldap-value
                   | ldap-type "~=" ldap-value
                   | ldap-type ">=" ldap-value
                   | ldap-type "<=" ldap-value
                   | ldap-type "present"
                   | ldap-type "*=" ldap-substrings
                   | ldap-ext-match

ldap-ext-match = ldap-type ["using" ldap-rule] "matches" ldap-value
                         | "*" "using" ldap-rule "matches" ldap-value

ldap-substrings = [ ldap-value ] "*"
                   { ldap-value "*" }
                   [ ldap-value ]
```

The values of the specified attribute type are tested against the test value:

- **"="** – equal to
- **"~="** – approximately equal to
- **">="** – greater than or equal to
- **"<="** – less than or equal to

The keyword `present` tests for the existence of the specified attribute type. A substring search can be performed using one or more wildcards (`"*"`, which will match zero or more characters) to separate a group of values for an equality match. An extensible match can be obtained using the using `ldap-ext-match` form.

```
ldap-rule = rulename | object-identifier | ldap-oid
```

A `rulename` is a string identifying a matching rule, usually the same as the matching rule's conventional ASN.1 name. Names for the standard matching rules are built in (see *Appendix B*) and are case insensitive. The rule may also be identified by an `object-identifier` in modified ASN.1 value notation, or by an `ldap-oid` using the dotted decimal representation for LDAP.

A size or time limit may be imposed on the search request with the **`limits`** argument.

```
limits     = [ size-limit ] [ time-limit ]
size-limit = "sizeLimit" "=" number
time-limit = "timeLimit" "=" number
```

ldap-search example

```
ldap search
      "o=Deltawing"
for sn ~= "smith" and
      not (givenName = "B" * or givenName = * "t")
return { sn givenName telephoneNumber };
```

This example searches the subtree headed by `o=Deltawing` for all entries with a `sn` (`surname`) approximately matching `smith` and `givenName` that does not start with 'B' or end in 't'. The attributes `surname`, `givenName` and `telephoneNumber` are returned for the matching entries.

list

Lists the subordinates of an entry.

```
list = "list" name [options ] [ output ] ";"
```

The `name` is the full DN of the entry whose subordinates are to be listed. If `name` is empty braces then the subordinates of the DIT root entry are listed.

If `output` is specified, the result of the `list` command is written to the named file.

Only the RDNs of the subordinates are listed.

For example:

```
list { organizationName "Deltawing" };
```

This returns the list of immediate subordinates of the entry { `organizationName "Deltawing"` }.

modify

Modifies the named entry. The modifications are performed in the order listed in the request.

```
modify  = "modify" name [ changes ]
              [ options ] ";"
changes = [ "with" ] "changes"
              "{" [ change { "," change } ] "}"
change  = rematt | remvals | addatt | addvals
rematt  = "remove" "attribute" type { value }
remvals = "remove" "values" attribute
addatt  = "add" "attribute" attribute
addvals = "add" "values" attribute
```

The `name` is the full DN of the entry to be modified.

Each `change` is applied to the entry in turn. The four alternatives are:

- `rematt` removes the attribute of the specified type. The values of a removed attribute may be listed but are ignored by Stream DUA. This facility is provided to allow reversal of the change recorded in the Stream DUA format update logs.
- `remvals` removes the listed values of the specified type.

- `addatt` adds a new attribute.
- `addvals` adds additional values to an existing attribute.

modify example

This example modifies the entry { `organizationName "Deltawing" / commonName "Fred Smith" }`, modifying the value of `telephoneNumber` and adding two new values of `employeeNumber`. Note that the `commonName` attribute value `Fred Smith` may not be modified with this command: you must use a rename command.

```
modify {
    organizationName "Deltawing"
    / commonName "Fred Smith"
}
with changes {
    remove values telephoneNumber "(03)543 2109",
    add values telephoneNumber "(03)543 9012",
    add attribute employeeNumber "1234" "9822"
};
```

move

Moves entries (and any subordinates) to a new position in the DIT.

```
move = "move" name "to" name
          [ delete-old ] [ options ] ";"
```

The first `name` is the full DN of the entry to be moved. The second `name` is the new name for the entry after the move.

For example:

```
move {
      organizationName "Deltawing"
      / organizationalUnit "Finance"
      / commonName "Fred Smith"
}
to {
      organizationName "Deltawing"
      / organizationalUnit "Marketing"
      / commonName "Fred Smith"
};
```

This moves the entry `Fred Smith` so that it becomes a subordinate of the entry { `organizationName "Deltawing" / organizationalUnit "Marketing"` }.

quit

Quit an interactive session (the same as pressing Control-D).

```
Quit
```

read

Reads a single entry.

```
read = "read" name
           [ "modifyRights" ]
           [ selection ]
           [ options ] [ output ] ";"
```

The `name` is the full DN of the entry to be read.

The keyword `modifyRights` requests the DSA to provide information about what permissions the current user has to modify the entry being read. This information is returned as a list of items the user is permitted to modify.

For example:

```
read {
    organizationName "Deltawing"
}
return { manager modifyTimestamp updatersName };
```

This reads the entry { `organizationName "Deltawing" }` and returns the attributes `manager`, `modifyTimestamp` and `updatersName`.

```
read { organizationName "Deltawing" } return all;
```

This reads the entry { `organizationName "Deltawing" }` and returns all attributes, including operational attributes such as `modifyTimestamp` and `updatersName`.

```
read {
    organizationName "Deltawing"
    / commonName "Fred Smith"
}
return all user types only;
```

This reads the entry { `organizationName "Deltawing" / commonName "Fred Smith" }` and returns all user attribute types in the entry, but no attribute values and no operational attribute types.

register

Adds schema definitions to Stream DUA's knowledge of the Directory schema.

```
register = "register" attribute ";"
```

The `attribute` argument is one of the schema publication operational attributes described in *Chapter 4*.

This command is normally not required; Stream DUA reads the schema publication attributes from the DSA with the `set` (`set-base` or `set-context`) command, and acquires schema knowledge from any schema publication operational attributes it sees in passing (for example, while inserting entries). It is only needed when Stream DUA is used with remote non-ViewDS DSAs that do not publish their schema. The file `quipu` in the `config` directory contains schema definitions that allow Stream DUA to interwork with QUIPU DSAs.

remove

Deprecated.

rename

Renames entries. It has the effect of only changing an entry's RDN and not its position in the directory tree. It is a special case of the `move` command .

```
rename = "rename" name "to" "(" rdn ")"
             [ delete-old ] [ options ] ";"
```

The `name` is the full DN of the entry to be renamed.

The `rdn` is the new (last) RDN of the entry.

For example:

```
rename {
    organizationName "Deltawing"
    / commonName "Fred Smith"
}
to
    ( commonName "Fred R. Smith" )
delete old;
```

This renames the entry `Fred Smith` to the new name `Fred R. Smith`, deleting the old value from the `commonName` attribute.

save

Makes a safe copy of the database files for backup purposes. The directory remains available for normal operation, including updating, while the command is being processed; the saved files reflect the database at the start of processing the command. Note that it is *unsafe* to backup or copy the actual database (`ddm.*`) files while the DSA is running. This command is only available to the super-administrator.

```
save = "save" [ "to" path ] ";"
```

If **`path`** is specified, the DSA will save its files in this file system directory. If not specified, it will save them into the file system directory specified by the configuration-file parameter *savedir* (which defaults to `${VFHOME}/save`). The `path` string must be enclosed in quotes.

search

Performs a search on the database.

```
search = "search" name
         [ scope ] [ aliases ]
         [ "for" filter]
         [ "paged" pagedResultRequest]
         [ "matched" ]
         [ selection ]
         [ options ] ";"
aliases = "and" [ "not" ] "aliases"
```

The `name` is the DN for the base entry at which to begin the search.

The `aliases` argument can be used to specify whether or not aliases *in the search subtree* are dereferenced. Note that the **`options`** argument through the `dontDereferenceAliases` field only controls whether aliases are dereferenced in locating the base object. Aliases in the search subtree are normally dereferenced, but this default may be overridden with the `set (set-search)` command. The

`aliases` argument allows the default to be overridden for the single command with which it is used.

The keyword `"for"` introduces a **`filter`** to apply to the search. A `filter` is a Boolean expression of `filter-item`. The `not` operator has highest precedence, followed by `"and"`, then `"or"`. Parentheses may be used to alter the order of evaluation.

```
filter    =   "(" filter ")"
              | filter-item [ "dnatt" ]
              | "not" filter
              | filter "and" filter
                | filter "or" filter
```

A `filter-item` is an assertion about the attribute values belonging to an entry. The keyword `"dnatt"` specifies that the assertion applies to the attribute values of the entry's DN in addition to the normal attribute values.

```
filter-item  =   type "=" value
                 | type "~=" value
                 | type ">=" value
                 | type "<=" value
                 | type "present"
                 | type "*=" substrings
                     | extensible-match
extensible-match =   (type | "*" ) "using"
                         matching-rule
                         { "," matchingRule }
                                "matches" value
substrings   =   [ value ] "*"
                     { value "*" } [ value ]
```

The values of the specified attribute type are tested for being equal to (`"="`), approximately equal to (`"~="`), greater than or equal to (`">="`) or less than or equal to (`"<="`) the specified test value.

The keyword `"present"` tests for the existence of the specified attribute type. A substring search can be obtained by using one or more wildcards (`"*"`, which will match zero or more characters) to separate a group of values for an equality match. An extensible match can be obtained using the `using … matches` form, in which `matchingRule` is the name or object identifier of one of the built-in matching rules.

search notes

ViewDS DSAs can only handle extensible matches that specify the `type` and exactly one matching rule.

The keyword `"paged"` introduces a paged results request. A paged result request enables a subset of the result set matching the search to be returned. Furthermore, the result to a search containing a paged result request will include a `queryReference` which may be used to collect more of the result set of the same search. This allows a large result set to be retrieved from the DSA a `"page"` at a time instead of all at once. The `pagedResultRequest` is an ASN.1 type with the following definition:

```
PagedResultsRequest ::= CHOICE {
      newRequest SEQUENCE {
```

```
            pageSize            INTEGER,
            sortKeys            SEQUENCE OF SortKey OPTIONAL,
            reverse         [1] BOOLEAN DEFAULT FALSE,
            unmerged            [2] BOOLEAN DEFAULT FALSE
        },
        queryReference OCTET STRING
}
```

The "matched" keyword is used to request matched values only. When this keyword is used, the search result will only return values of the attribute types in the filter which actually matched some filter-item in the filter. For attribute types which are requested in the selection but are not mentioned in the filter, all values are returned as usual.

search examples

```
search { } for surname = "smith";
```

This searches the whole DIT (or the whole subtree below the prefix entry if a prefix has been set) for entries having the attribute surname with an exact value smith, returning all user attributes for any matching entries. Note that letter case is ignored when doing an equality match on the surname attribute.

```
search {
    organizationName "Deltawing"
}
for surname ~= "smith" and
    not (givenName = "B" * or givenName = * "t")
return { surname givenName telephoneNumber };
```

This searches the subtree headed by { organizationName "Deltawing" } for all entries having surname approximately matching smith and givenName not starting with 'B' or ending in 't'. The attributes surname, givenName and telephoneNumber are returned for the matching entries.

```
search {
    organizationName "Deltawing"
}
one level
for objectClass = organizationalUnit
return { organizationalUnitName manager };
```

This searches the immediate subordinates of { organizationName "Deltawing" } for entries with object class organizationalUnit, returning the attributes organizationalUnitName and manager for these entries.

```
search { }
for specificKnowledge present
return { specificKnowledge };
```

This searches the whole DIT for all entries having the specificKnowledge attribute, returning just that attribute for each entry.

set

Establishes a context for subsequent commands.

```
set = "set" ( set-base | set-context | set-prefix
                | set-search | options
                | set-username | set-password
```

```
        | set-protocol | set-timing
            | set-synchronization ) ";"
```

The command has the ten forms described below.

set-base

Reads all the schema information from the chain of entries from the root of the DIT to the entry with the nominated DN (`name`). When it starts up, the Stream DUA only understands the predefined attributes, so it must read the schema publication attributes to acquire an understanding of the user-defined attributes (see *Chapter 4*).

```
set-base = "base" [ "object" ] "to" name
```

set-context

The same as `set-base` but also searches for schema information in the subtree of the nominated entry.

The `set-context` alternative is the simplest command to load schema information. A `set` command with either `set-base` or `set-context` should be used as the first command to Stream DUA every time it is invoked. This is most conveniently done by including the command in the `sdua.startup` file (see page 6).

```
set-context  = "context" "to" name
```

set-prefix

Sets a value for the prefix implied at the start of a relative `name`. Its initial value is the DIT root.

```
set-prefix   = "prefix" [ "to" ] name
```

For example:

```
set prefix {
    organizationName "Deltawing"
    / organizationalUnit "Marketing"
};
```

set-search

Sets the default for dereferencing of aliases in the search subtree for the `search` command.

```
set-search   = "search" [ "not" ] "aliases"
```

set-username

Modifies the default username when binding to the DSA. It may be either a userName attribute value or a DN. It overrides the username set using the `-u` or `-a` command line options.

```
set-username = "username" username
```

set-password

Modifies the default password to use to bind to the DSA. It overrides the password set using the `-p` or `-a` command line options.

```
set-password = "password" password
```

For example:

```
set username "asherma";
```

```
set username {
    organizationName "Deltawing"
    / organizationalUnitName "Deltawing InfoSystems"
    / commonName "Andrew Sherman"
};
```

set-protocol

Modifies the default protocol used to bind to the DSA. The default protocol is normally DAP but may be changed to DAP Admin Protocol.

```
set-protocol = "protocol" protocol
```

set-timing

Enable or disable timing of operations sent to the DSA. When enabled, Stream DUA measures the elapsed time between sending the operation to the DSA and receiving a response and displays this information as a message at the beginning of the operation result.

For example:

```
# Started Wed Apr 19 14:00:56 2000, Execution Time: 0.030000
```

The execution time is measured in seconds. The default setting for timing is `"off"`.

```
set-timing   = "timing" "=" ( "on" | "off" )
```

set options

Sets default values for options (see *options* on page 218) that apply to subsequent operations. It is overridden by an explicit `options` on a command. This command is most useful in a file.

For example:

```
set options {
      serviceControls {
          options { localScope },
          priority high,
          timeLimit 2,
          attributeSizeLimit 1000
      }
}
```

set-synchronization

Synchronization is configured by an extension to the `set` command:

```
set-synchronization  =    "synchronization" [ "context" ]
                                      ( "none" | value ) ";"
```

The `value` is of the `SynchronizationContext` ASN.1 type in Stream DUA value-notation format shown below.

```
SynchronizationContext ::= SEQUENCE {
      subtree    [1] DistinguishedName OPTIONAL,
      types      [2] SEQUENCE OF AttributeType OPTIONAL,
      classes    [3] SEQUENCE OF SynchronizedClass,
      timestamp  [4] GeneralizedTime OPTIONAL,
      group      [5] DirectoryString { ub-name } OPTIONAL
}
```

When a `SynchronizationContext` is set, Stream DUA attempts to synchronize the content in any subsequent entry command with a corresponding entry in the directory. Note that this behaviour only applies to the Stream DUA (or vfload) entry command or LDIF entry record. It does not apply to the otherwise-equivalent insert command.

The synchronizing behaviour is limited to the duration of the Stream DUA (or vfload) process. It can be switched off before the process terminates by setting the synchronization context to `"none"`. A new synchronization context can be set at any time (in which case, the previous synchronization context is discarded).

Directory entries to be synchronized are either identified by DN or by a search on their attribute values. For a search, the subtree component can be used to specify the base object for the search (if it is not used, the base object is the root).

The components of `SynchronizationContext` are as follows:

`subtree` Specifies the base object for a search (if unused, the base object is the root).

`types` Lists the attribute types to be synchronized regardless of the object class of an input entry.

`classes` Provides additional information for synchronizing entries of specific structural object classes. However, any entry specified in the input will be synchronized even if its structural object class is not listed by this component.

If the classes component is used, an input entry should contain either an objectClass or structuralObjectClass attribute.

`timestamp` If the timestamp component is present, a synchronized entry will have the value of its `synchronizationTimestamp` attribute set to the value of this component.

The value of the timestamp component has no particular relevance, although setting it to a value close to the current time makes sense.

If synchronization is performed over multiple runs of the Stream DUA (or vfload), each run should use the same value for the timestamp component. Likewise, if a run is repeated after fixing errors, the same value for the timestamp component should be used.

In typical use, any entries that have a different `synchronizationTimestamp` value will be deleted by a subsequent filtered delete subtree command (see *synchronization* on page 201). The reason for this is that any entry that has a `synchronizationTimestamp` value is being synchronized from an external source. However, if the value hasn't been updated to the current timestamp, then a record corresponding to that entry no longer appears in the external source data (it has been deleted).

The `synchronizationTimestamp` attribute is a predefined, built-in operational attribute permitted in any entry.

`group` If the group component is present, a synchronized entry will have the value of its `synchronizationGroup` attribute set to the value of this component.

The group component is provided for situations where disjoint collections of directory entries are:

- synchronized from multiple, independent sources; and
- the collections cannot be easily distinguished from each other by a filtered delete subtree command (for example, because they appear in the same subtree and use the same object classes).

The `synchronizationGroup` attribute is a predefined, built-in operational attribute permitted in any entry.

SynchronizationClass

The SynchronizationClass ASN.1 type is defined as follows:

```
SynchronizedClass ::= SEQUENCE {
      structuralObjectClass [0] OBJECT-CLASS.&id,
      types [1] SEQUENCE OF AttributeType OPTIONAL,
      identifiers [2] SEQUENCE OF AttributeType OPTIONAL,
      nameAuthority [3] NameAuthority OPTIONAL,
      newEntryPermitted [4] BOOLEAN DEFAULT TRUE,
      newEntryTypes [5] SEQUENCE OF AttributeType OPTIONAL
}

NameAuthority ::= ENUMERATED {
      none,
      onlyRDN,
      fullDN
}
```

The components are as follows:

`structural ObjectClass` — A `SynchronizedClass` instance applies to entries with the structural object class identified by the `structuralObjectClass` component.

There can be a maximum of one `SynchronizedClass` instance for each structural object class.

`types` — This component lists the attribute types to be synchronized. They are in addition to the attribute types specified by the types component of the enclosing `SynchronizationContext` instance.

If an attribute type in an input entry is not listed in either of the types components, it is ignored.

`identifiers` — If the identifiers component is absent or empty, an entry in the input is synchronized with the entry in the directory with the same DN. If no entry exists with the same DN, the entry in the input is added to the directory (unless the `newEntryPermitted` component is present and set to `FALSE`).

This is also the default behaviour for an entry whose structural object class does not have a `SynchronizedClass` instance.

`newEntryTypes` This component lists additional attribute types from the input entry that are to be included when a new entry is added. It is ignored when the `newEntryPermitted` component is present and set to `FALSE`.

The `objectClass` attribute and any identifier attributes are automatically included in a new entry.

`nameAuthority` This component only applies if the identifiers component is present and is not empty.

If a matching entry is found in the directory and the value of the `nameAuthority` component is:

- `none` (or the nameAuthority component is absent) – the DN of the entry in the directory is preserved.
- `onlyRDN` – the RDN of the entry in the directory is changed to the RDN of the entry in the input. (The entry in the directory retains its superior entry.)
- `fullDN` – the entry in the directory is moved and renamed so that it has the DN of the entry in the input.

The synchronization procedure does not have the functionality to avoid conflicts when entries are moved and renamed. These conflicts produce errors that have to be fixed by editing and rerunning the input.

synchronization example

The following is an example of the first step when synchronizing a directory with a collection of entries:

```
set synchronization context
      {
          subtree {
              organizationName "Deltawing"
          },
          types {
              objectClass
          },
          classes {
              {
                  structuralObjectClass organizationalUnit,
                  types {
                      organizationalUnitName,
                      telephoneNumber
                  }
              },
              {
                  structuralObjectClass organizationalPerson,
                  types {
                      commonName,
                      surname,
                      title,
```

```
                    telephoneNumber,
                    employeeNumber
                },
                identifiers {
                    employeeNumber
                },
                nameAuthority fullDN
            }
        },
        timestamp "20080101120000"
    }
;
```

The next step is to delete all entries with a `synchronizationTimestamp` attribute that does not match the value of the timestamp component in the `SynchronizationContext` instance.

The delete subtree command can include a search filter to delete a subset of entries in a subtree. The extended BNF for delete command is as follows:

```
delete = "delete" [ "subset" filter [ "from" ] ]
[ "subtree" ] name [ content ] [ options ] ";"
```

The obsolete synchronized entries can be deleted using the filtered subtree delete command. For example:

```
delete
    subset (synchronizationTimestamp present and
        not synchronizationTimestamp = "20080101120000")
    from subtree { organizationName "Deltawing" }
;
```

The above example tests for the presence of the `synchronizationTimestamp` attribute. This is necessary because an entry without a `synchronizationTimestamp` attribute will satisfy the ‘not’ filter item.

The delete subtree command can take an arbitrary filter, which is useful when there are multiple independent input sources. For example, consider a directory that includes staff entries synchronized from an independent HR database. This same directory includes other kinds of entries that are also synchronized, but from another source.

In this example scenario, the delete subtree command might be as follows:

```
delete
    subset (objectClass = organizationalperson and
        synchronizationTimestamp present and
        not synchronizationTimestamp = "20080101120000")
    from subtree { organizationName "Deltawing" } ;
```

If the same staff entries were drawn from multiple external sources, then the group component of the `SynchronizationContext` instance could be used. The delete subtree command might be as follows:

```
delete
    subset (synchronizationGroup = "HR Staff Records" and
        not synchronizationTimestamp = "20080101120000")
    from subtree { organizationName "Deltawing" }
```

```
;
```

In this example, the test for the presence of the `synchronizationTimestamp` attribute is unnecessary. This is because any entry that has the required value for `synchronizationGroup` will also have a `synchronizationTimestamp` attribute.

synchronization notes

Note that the Stream DUA also takes notice of any values of schema publication attributes that may be used in update operations, or that are returned in the results of read and search operations.

Also note that the dump operation places a suitable `set` command at the start of each `dib.*` file.

show

Displays the current setting for `prefix`, `search`, `options`, `username` or `protocol`. It is most useful in interactive mode.

```
show  =   "show" ( "prefix" | "search"
          | "options" | "username"
           | "protocol" ) ";"
```

For example: `show prefix;`

source

Transfers the source of the command input from the current source to the named file.

```
source = "source" path ";"
```

The `path` is the name of a file from which to read commands; it must be enclosed in quotes. Commands are processed until an end of file or an `exit` or `quit` command, the processing resumes with the current input source. The `source` command can be nested.

unbind

Unbinds the Stream DUA from the DSA.

```
unbind = "unbind" ";"
```

The `unbind` command is performed implicitly prior to a `bind`, `exit` or `quit` command, but can also be given explicitly. It is not an error to request this operation if Stream DUA is already unbound.

userlist

Displays information about all currently bound users of the DSA.

```
userlist = "userlist" ";"
```

verify

Requests the DSA to verify its database integrity. The DSA performs a number of integrity checks on its database, and reports problems to its error log, as well as a success or failure message to the Stream DUA.

```
verify = "verify" ";"
```

With large a database this command can take a long time to complete.

xldap-add

Inserts a new entry into the database using the XLDAP protocol.

```
xldap-add =   "xldap" "add" name
                  [ xldap-content ]
                      [ xldap-controls ] ";"
```

The `name` is the full DN for the new entry.

The list of attributes given by the **`content`** argument should include the *object class* of the entry. For object classes that are subclasses of other classes, all object classes below `top` need to be given.

xldap-add example

```
xldap add {
      organizationName "Deltawing"
      / commonName "James Clarke"
}
with {
      objectClass organizationalPerson person,
      surname "Clarke",
      givenName "James",
      userName "jclarke",
      userPassword "testpass"
}
controls {
      {
          controlType { 1 3 6 1 4 1 42 2 27 8 5 1 },
          criticality FALSE
      }
};
```

This example adds a new organizational person, James Clarke, to the DIT and supplies attribute values for the `objectClass, surname, givenName, userName` and `userPassword`. It contains a Password Policy Control (see *Chapter 6*).

xldap-compare

Tests for a given value in an entry using the XLDAP protocol.

```
xldap-compare = "xldap" "compare" name "to" ldap-type "=" value
                      [ xldap-controls ] ";"
```

The `name` is the full DN for the entry to be compared. Either `true` or `false` is returned according to whether the entry has or does not have an attribute of the given type with the given value.

For example:

```
xldap compare {
      organizationName "Deltawing"
      / commonName "Fred Smith"
}
to
    userPassword = '414243444546'H
;
```

This compares the `userPassword` in the entry `{ organizationName "Deltawing" / commonName "Fred Smith" }` with the supplied value `'414243444546'H` (that is, `"ABCDEF"`), returning `true` if the password matches.

xldap-delete

Removes an entry from the database using the XLDAP protocol.

```
xldap-delete = "xldap" "delete" name
               [ xldap-content ]
               [ xldap-controls ] ";"
```

The `name` is the full DN of the entry to be deleted. Only entries with no subordinates may be deleted.

For example:

```
xldap delete {
      organizationName "Deltawing"
      / organizationalUnit "Finance"
}
;
```

This example will remove the organizational unit “Finance” if it exists and does not have any subordinates.

xldap-modify

Modifies the named entry using the XLDAP protocol. The modifications are performed in the order listed in the request.

```
xldap-modify =   "xldap" "modify" name
                 [ xldap-changes ]
                 [ xldap-controls ] ";"
```

The `name` is the full DN of the entry to be modified.

```
xldap-changes    =   [ "with" ] "changes" "{"
                     [ xldap-change
                     { "," xldap-change } ]
                     "}"
xldap-change =   change-add
                 | change-delvals
                 | change-delatt
                 | change-replace
change-add   = "add" ("attribute" | "values")
                 ldap-type { value }
change-delvals = "delete" "values" ldap-type { value }
change-delatt  = "delete" "attribute" ldap-type { value }
change-replace = "replace" ldap-type
                 [ old-values ]
                 [ "add" { value } ]
old-values     = "(" "delete" { value } ")"
```

Each `xldap-change` is applied to the entry in turn. The four alternatives are:

- `change-add` adds the attribute of the specified type if it doesn’t already exist in the entry, otherwise it adds the listed values to the existing attribute. The choice of keyword, either `"attribute"` or `"values"`, carries no significance.

- `change-delvals` removes only the listed values from the attribute of the specified type. If no values are listed, the change is ignored.
- `change-delatt` removes the attribute of the specified type. The values of the removed attribute may be listed but are ignored by Stream DUA. This facility is provided to allow reversal of the change recorded in the Stream DUA format update logs.
- `change-replace` removes all the current values of the attribute of the specified type then adds the listed values, if any. It is equivalent to a `change-delatt` followed by a `change-add` for the same attribute type. The removed values may be listed with `old-values` but these are ignored by Stream DUA. This facility is provided to allow reversal of the change recorded in the Stream DUA format update logs.

xldap-modify example:

```
xldap modify {
      organizationName “Deltawing”
      / commonName “Fred Smith”
}
with changes {
      remove values telephoneNumber "(03)543 2109",
      add values telephoneNumber "(03)543 9012",
      add attribute employeeNumber "1234" "9822"
};
```

This example modifies the entry `{ organizationName “Deltawing” / commonName “Fred Smith” }`, modifying the value of `telephoneNumber` and adding two new values of `employeeNumber`. Note that the `cn (commonName)` attribute value `Fred Smith` may not be modified with this command: you must use a rename command (for example, `xldap-rename`).

xldap-move

Moves entries (and any subordinates) to a new position in the DIT using the XLDAP protocol.

```
xldap-move   =   “xldap” “move” name “to” name
                     [ delete-old ] [ xldap-controls ] “;”
```

The first `name` is the full DN of the entry to be moved. The second `name` is the new name for the entry after the move.

For example:

```
xldap move {
      organizationName “Deltawing”
      / organizationalUnit “Finance”
      / commonName “Fred Smith”
}
to {
      organizationName “Deltawing”
      / organizationalUnit “Marketing”
      / commonName “Fred Smith”
};
```

This moves the entry `Fred Smith` so that it becomes a subordinate of the entry `{ organizationName “Deltawing” / organizationalUnit “Marketing” }`.

xldap-rename

Renames entries using the XLDAP protocol. It has the effect of only changing an entry's RDN and not its position in the directory tree. It is a special case of the `xldap-move` command .

```
xldap-rename =   “xldap” “rename” name
                     “to” “(“ rdn “)”
                     [ delete-old ]
                     [ xldap-controls ] “;”
```

The `name` is the full DN of the entry to be renamed. The `rdn` is the new (last) RDN of the entry.

xldap-rename example

```
xldap rename {
      organizationName “Deltawing”
      / commonName “Fred Smith”
}
to    ( commonName “Fred R. Smith” )
delete old
;
```

This renames the entry `Fred Smith` to the new name `Fred R. Smith`, deleting the old value from the `cn (commonName)` attribute.

xldap-search

Performs a search on the database.

```
xldap-search =   “xldap” “search” name
                     search-arguments
                     [ xldap-controls ] “;”

search-arguments =   [ scope ]
                         [ “and” xldap-aliases ]
                         “for” xldap-filter
                         [ ldap-selection ] limits

xldap-aliases=   “neverDerefAliases”
                     | “derefInSearching”
                     | “derefFindingBaseObj”
                     | “derefAlways”
```

The `name` is the DN for the base entry at which to begin the search.

The `xldap-aliases` argument can be used to specify whether or not aliases in the search subtree and/or the base entry name are dereferenced. Aliases in the search subtree are dereferenced if `derefInSearching` or `derefAlways` is specified, otherwise they are not dereferenced. Aliases are dereferenced in locating the base entry if `derefFindingBaseObj` or `derefAlways` is specified, otherwise they are not dereferenced. The default, if `xldap-aliases` is not specified, is `derefAlways`.

The keyword **`"for"`** introduces a filter to apply to the search. An `xldap-filter` is a Boolean expression of `xldap-filter-item`. The `"not"` operator has highest

precedence, followed by **"and"**, then **"or"**. Parentheses may be used to alter the order of evaluation.

```
xldap-filter =   "(" xldap-filter ")"
                    | xldap-filter-item [ "dnatt" ]
                    | "not" xldap-filter
                    | xldap-filter "and" xldap-filter
                    | xldap-filter "or" xldap-filter
```

A `xldap-filter-item` is an assertion about the attribute values belonging to an entry. The keyword `"dnatt"` specifies that the assertion applies to the attribute values of the entry's DN in addition to the normal attribute values.

```
xldap-filter-item    =   ldap-type "=" value
                         | ldap-type "~=" value
                         | ldap-type ">=" value
                         | ldap-type "<=" value
                         | ldap-type "present"
                         | ldap-type "*=" xldap-substrings
                         | xldap-ext-match
xldap-ext-match = ldap-type ["using" xldap-rule] "matches" value
                  | "*" "using" xldap-rule "matches" value
xldap-substrings =   [ value ] "*"
                     { value "*" }
                     [ value ]
```

The values of the specified attribute type are tested for being equal to (`"="`), approximately equal to (`"~="`), greater than or equal to (`">="`) or less than or equal to (`"<="`) the specified test value.

The keyword `"present"` tests for the existence of the specified attribute type. A substring search can be obtained by using one or more wildcards (`"*"`, which will match zero or more characters) to separate a group of values for an equality match. An extensible match can be obtained using the using `xldap-ext-match` form.

```
xldap-rule = rulename | object-identifier
```

A `rulename` is a string identifying a matching rule, usually the same as the matching rule's conventional ASN.1 name. Names for the standard matching rules are built in (see *Appendix B*) and are case insensitive. The rule may also be identified by an `object-identifier` in modified ASN.1 value notation.

A size or time limit may be imposed on the search request with the `limits` argument.

```
limits     = [ size-limit ] [ time-limit ]
size-limit = "sizeLimit" "=" number
time-limit = "timeLimit" "=" number
```

xldap-search example

```
xldap search {
      organizationName "Deltawing"
}
for sn ~= "smith" and
    not (givenName = "B" * or givenName = * "t")
return { sn givenName telephoneNumber };
```

This searches the subtree headed by `{ organizationName "Deltawing" }` for all entries having sn (surname) approximately matching `smith` and `givenName` not starting with 'B' or ending in 't'. The attributes surname, `givenName` and `telephoneNumber` are returned for the matching entries.

Stream DUA notation

This appendix describes the following aspects of the Stream DUA:

- ASN.1 value notation
- Input language notation and common arguments
- Service controls

ASN.1 value notation

The Stream DUA uses a data specification language used extensively in the X.500 Recommendations called ASN.1 (Abstract Syntax Notation 1). ASN.1 is suitable for specifying complex data types (for example, attribute syntaxes) and their values. Stream DUA uses ASN.1 to specify attribute values and the `options` argument.

ASN.1 type and value notation is described in *ITU Recommendation X.680*, and it is illustrated by examples in this chapter.

Note the following:

- A value of a SEQUENCE or SET is a list of the component values enclosed in braces.
- A value of a BIT STRING is a list of the named bits enclosed in braces.
- The identifier is always required for each component of a SEQUENCE or SET, or for the actual chosen alternative within a CHOICE.
- There must be a colon between the identifier of an alternative of a CHOICE and its value.
- The XML Enabled Directory specification defines an ASN.1 type, `AnyType`, the values of which are able to hold arbitrary content of XML elements (see *draft-legg-xed-glue-xx.txt*). Values of `AnyType` are represented by SDAU in ASN.1 value notation according to the ASN.1 type definition of `AnyType` (a CHOICE of a SEQUENCE) which tends to obscure the XML content. However, the representation of `AnyType` values in ELDIF, which `sdua` also accepts, is much closer to the original XML notation.

Stream DUA format

The Stream DUA accepts some modifications to standard ASN.1 value notation to simplify its use as an input/output language for the directory. In particular, a different notation is optionally used for values of restricted string types (for example, `PrintableString`, `TeletexString)`, and values of type `OCTET STRING`, `DirectoryString`, `ORAddress`, and `ORName`, and must be used for values of type `DistinguishedName`. The special notation for these syntaxes is described below, together with examples of many of the attribute syntaxes supported by ViewDS.

In addition, two special formats are available for representing syntaxes which are unknown to Stream DUA, or which represent values which are to be hidden from human users. These formats are best illustrated by example:

```
BER:'130421222324'H       # BER encoding of the PrintableString "1234"
```

```
ENCRYPTED:'5415D9CB40F65D893C1BB21C91B177ED'H
                                # a value hidden from human users
```

The `ENCRYPTED` form is used by the DSA when dumping directory data that contains values of the `userPassword` attribute, in order to prevent disclosure of the actual value. The DSA and Stream DUA must both use the same encryption key.

BIT STRING

A `BIT STRING` value can be given in binary or hexadecimal notation, but can usually be given using named bits as a comma-separated list in braces.

```
'00110000'B                     # using binary notation
{dapProtocol, dspProtocol}          # using named bits
```

BOOLEAN

A `BOOLEAN` value is given as `TRUE` or `FALSE`.

DistinguishedName

A `DistinguishedName` value must be given as a *name* enclosed in braces. The syntax for *name* is described later in this chapter.

```
{   # shorthand format
      organizationName "Deltawing"
      / organizationalUnit "Deltawing InfoSystems"
}
{     # sdua output format
      organizationName "Deltawing"
      / organizationalUnitName "Deltawing InfoSystems"
}
```

DirectoryString

A value of type `DirectoryString` may be represented as a value of type `PrintableString`, `TeletexString`, `BMPString`, `UniversalString` or `UTF8String` (that is, the `CHOICE` tag `printableString:`, `teletexString:`, `bmpString:`, `universalString:` or `uTF8String:` can be omitted).

On input it will be automatically converted to a `DirectoryString` according to the following algorithm:

- If the string contains characters other than UNICODE characters, it is a UniversalString.
- Otherwise if the string contains characters other than Latin1 (ISO 8859-1) characters, it is a BMPString.
- Otherwise if the string contains characters other than Printable characters it is a TeletexString.
- Otherwise it is a `PrintableString`.

On output by Stream DUA or from a DSA `dump`, the tags are omitted if they can be reconstructed exactly by the above algorithm.

Therefore:

- A `PrintableString` is always dumped without a tag.
- A `TeletexString` is dumped without a tag if it contains any non-printable string characters and no non-Latin1 characters.

- A `BMPString` is dumped without a tag if it contains any non-Latin1 characters and no non-UNICODE characters.
- A `UniversalString` is dumped without a tag if it contains any non-UNICODE characters.

Examples:

```
"Mr Fred Smith"             # DirectoryString with an implicit CHOICE
teletexString: "ABCDEF"     # DirectoryString with an explicit CHOICE
```

ENUMERATED

An `ENUMERATED` value can be given as an integer, but is usually given as a named integer. Just specify the integer value's name:

```
nssr
```

FacsimileTelephoneNumber

A `FacsimileTelephoneNumber` value consists of a printable string containing a telephone number in international (E.123) format followed by an optional `BIT STRING` giving Group 3 fax non-basic parameters. The latter string is generally omitted, but its possible presence requires a more complex syntax for `Facsimile-TelephoneNumber`.

```
{
      telephoneNumber "+1-212-667-9801"
}
```

Note that the `telephoneNumber` attribute also requires the number to be given in E.123 format. However, ViewDS does not enforce this in either case.

GeneralizedTime

A `GeneralizedTime` value specifies either a local time or a coordinated universal time (UTC) as a string in ISO 8601 format. It has three forms: (a) local time, (b) UTC time, (c) local time with indicated offset from UTC time. Timestamps generated by ViewDS always use format (c).

Note that the seconds or minutes and seconds digits may be omitted, and that a decimal point and fractional seconds may be given after the integral seconds digits.

```
"19951106210627"          # (a): local time 9:06:27 p.m. 6 Nov 1995
"19951106210627Z"         # (b): UTC time 9:06:27 p.m. 6 Nov 1995
"19951106210627+1000"     # (c): local time 9:06:27 p.m. 6 Nov 1995
                          # with local time 10 hours ahead of UTC
```

OBJECT IDENTIFIER

An `OBJECT IDENTIFIER` value can be given as a list of OID components in braces, or as a symbolic name (provided the name is known to Stream DUA):

```
{2 5 4 3}
```

```
commonName
```

OCTET STRING

An `OCTET STRING` value may be given in hexadecimal notation, or it if contains only ASCII printable characters (space through tilde), may alternatively be given as a quoted string:

```
'414243444546'H         # OCTET STRING in hexadecimal notation
"ABCDEF"                # OCTET STRING as a quoted string
```

ORAddress

An `ORAddress` value may be given using standard ASN.1 value notation, or alternatively as a `TeletexString` whose value is the address converted to a string in accordance with *X.402 (1995) Annex F*.

```
{     # standard ASN.1 value notation
      standard-attributes {
                  admd-name printable:"Telememo",
          prmd-name printable:"Delta Auto",
          organization-name "Deltawing",
          personal-name {
              surname "Smith",
              given-name "Marie"
          },
          org-unit-names {
              "Delta Automotive"
          }
      }
}
"G=Marie; S=Smith; OU=Delta Automotive; O=Deltawing; " +
      "P=Delta Auto; A=Telememo"
                                      # as a string in X.402 Annex F
format
```

The *X.402 Annex F* format permits the attributes to be given in any order, and permits either `;` or `/` to be used as a delimiter between attributes.

ORName

An `ORName` value consists of an `ORAddress` with an optional `DistinguishedName` added as the last value of the SEQUENCE. It may optionally be given as an *X.402 Annex F* format `TeletexString` followed by a `DistinguishedName` value.

For example:

```
{
      "G=Marie; S=Smith; OU=Delta Automotive; O=Deltawing; " +
          "P=Delta Auto; A=Telememo",
      {
          organizationName "Deltawing"
          / organizationalUnitName "Deltawing Automotive Ltd."
          / organizationalUnitName "Sales"
          / commonName "Marie Smith"
      }
}
```

PostalAddress

A `PostalAddress` value is a sequence of up to six strings of no more than 30 characters each:

```
{
      "GPO Box 8",
      "Melbourne",
      "Victoria 3000"
}
```

Restricted String Types

A value of a restricted string type (for example,, `PrintableString`, IA5String, `TeletexString`, `BMPString`, `UniversalString` or `UTF8String)` is represented as a sequence of one or more juxtaposed *substrings* separated by '+' signs. The substrings are concatenated to form a single string.

Each substring must fit wholly onto a single line and is one of the following:

- A sequence of characters in the local code page (currently ISO Latin1 (ISO 8859-1)) enclosed in matching quotes, any of ', " or `. If the string contains a quote of the same type as the enclosing quotes then the embedded quote must be escaped by duplicating it in the string.
- One of the named characters `lf` (representing 10, a line feed) or `cr` (representing 13, carriage return).
- A sequence of hexadecimal characters enclosed in single quotes (`'`) followed by one of `H`, `T61`, `BMP` or `UCS`. The hexadecimal characters specify one or more characters of the string in the local code page (`H`), in the T.61 character set (`T61`), in the BMP character set (`BMP`) or in the UCS character set (`UCS`). In the case of T.61, 4 hex characters are required for each actual character, the first two specifying the code set, and the final two the position in the code set. In the case of BMP, 4 hex characters are again required for each actual character and specify the 16-bit BMP value of the character. In the case of UCS, 8 hex characters are required for each actual character and specify the 32-bit UCS value of the character.

Note that the characters allowed for a `PrintableString` are `A-Z`, `a-z`, `0-9`, *space*, plus the following: `'()+,-./:=?`

Examples:

```
'Mr Fred O''Hara'
"Marketing Manager for Mel"    # String split over two lines
      + "bourne region"
"hello" + lf + "there"         # String with embedded linefeed
teletexString:"ABCDEF"         # DirectoryString of type TeletexString
"ABC" + '2345'BMP              # BMPString of 4 chars, last is '2345'
```

Note that Access Presence uses a different representation for characters that cannot be represented in the local-code page. Such a character is represented in-line using an escape notation: a backslash followed by the 4 hexadecimal digits of its Unicode representation. A genuine backslash and the concatenation character for multi-valued attributes of that type are also escaped with a backslash if they occur in the string.

Open Type

The governing type and colon for a value of an ASN.1 open type are always omitted. An attribute value embedded in an access control item is an example of an open type. In such a case, the attribute value is represented in the usual way for Stream DUA without any need to identify the governing ASN.1 type.

Input language notation and common arguments

Input to the Stream DUA is a sequence of commands, each terminated by a semicolon (except for `quit` and `exit`). Spaces, tabs, and newlines are not significant.

The language accepted by Stream DUA is defined using extended Backus-Naur Form:

`|` separates alternative expressions,

`()` groups the enclosed expressions,

`{}` means zero, one or more repetitions of the enclosed expression and

`[]` means the enclosed expression is optional.

Literal text (for example, a keyword in the language) is surrounded by double quotes.

Any text following a '`#`' up to the end of the line is treated as a comment and ignored.

At the top level, the Stream DUA input language is described by the following BNF:

```
input     = { command }
command   = dap-request | admin-request | ldap-request | dua-command
dap-request = bind | compare | insert | list | modify | move | read |
              rename | delete | search | unbind
admin-request = add | checkpoint | display | dsa | dump | empty | fill
                  | remove | save | userlist | verify
ldap-request = ldap-add | ldap-compare | ldap-delete | ldap-modify |
                  ldap-move | ldap-rename | ldap-search
xldap-request = xldap-add | xldap-compare | xldap-delete |
              xldap-modify | xldap-move | xldap-rename | xldap-search
dua-command = assign | encode | exit | quit | register |
                set | show | source
```

These commands are described below.

Common command arguments

Arguments common to two or more Stream DUA commands are described below.

type

The `type` argument specifies a single attribute type.

```
type = typename | object-identifier
```

A `typename` is a string identifying the attribute, usually the same as the attribute's conventional ASN.1 name. Type names for the standard attributes are built in (see *Appendix B*). The synonyms `C`, `O`, `OU`, `CN`, `S` for `countryName`, `organizationName`, `organizationalUnitName`, `commonName`, and `surname` are also built in. Typenames for user-defined attributes are taken from the `name` component of the attribute's `attributeTypes` value (any of the names may be used). `sdua` reads these in response to a **`set`** command. Type names are case insensitive.

An `object-identifier` is represented using ASN.1 value notation, i.e. as a sequence of integers enclosed in braces (see *Chapter 4*).

Examples:

```
organizationName
OU
```

```
commonName
{ 2 5 4 4 }
```

selection

The `selection` argument specifies the attribute types to be returned in the result of a `read` or `search` command.

```
selection = "return" [ category ]
                [ "types" [ "only" ] ]
                [ types ]

category    = all | user | operational
all         = "all"
user        = "all" "user"
operational = "all" "operational"
types       = "{" { type } "}"
```

If the `selection` argument is omitted or `user` is specified then all *user* attributes of the entry are returned. If `operational` is specified, all *operational* attributes are returned. If `all` is specified then all user and operational attributes are returned.

The implied attribute types selected by `category` can be augmented with a specific list of attribute `types`, between braces and separated by white-space.

If `category` is absent, and `types` is absent or is an empty list of attribute types, then no attributes are returned.

The `selection` can also include the keywords `"types" "only"` which results in the attribute types only (that is, no values) being returned.

For example, to return the attributes `manager`, `modifyTimestamp` and `updatersName`:

```
return { manager modifyTimestamp updatersName }
```

To return all attributes, including operational attributes such as `modifyTimestamp` and `updatersName`:

```
return all
```

To return all user attribute types in the entry, but no attribute values and no operational attribute types:

```
return all user types only
```

This specifies that no attributes are to be returned:

```
return { }
```

ldap-selection

The `ldap-selection` argument specifies the attribute types to be returned in the result of an `ldap-search` or `xldap-search` command.

```
ldap-selection = "return" [ category ]
                [ "types" [ "only" ] ]
                [ types ]
category    = all | user | operational
all         = "all"
user        = "all" "user"
operational = "all" "operational"
types       = "{" { ldap-type } "}"
```

If the `ldap-selection` argument is omitted or `user` is specified then all *user* attributes of the entry are returned. If `operational` is specified, all *operational* attributes are returned. If `all` is specified then all user and operational attributes are returned.

The implied attribute types selected by `category` can be augmented with a specific list of attribute `types`, between braces and separated by white-space.

If `category` is absent, and `types` is absent or is an empty list of attribute types, then no attributes are returned.

The `selection` can also include the keywords `"types" "only"` which results in the attribute types only (that is, no values) being returned.

Examples

To return the attributes `userCertificate;binary` and `modifyTimestamp`:

```
return { "userCertificate;binary" modifyTimestamp }
```

To return all attributes including operational attributes:

```
return all
```

To return all user attribute types in the entry, but no attribute values and no operational attribute types:

```
return all user types only
```

To specify that no attributes are to be returned:

```
return { }
```

octet-string

An `octet-string` argument is an arbitrary string of 8-bit octets presented in the same form as the modified ASN.1 value notation for a value of the OCTET STRING syntax.

For example:

```
'414243444546'H
```

value

A value is an attribute or assertion value.

```
value = asn1value | ber | from
ber   = "BER" ":" octet-string
from  = "FROM" [ "BER" ":" ] path
```

A `value` is typically expressed using modified ASN.1 value notation (see above). The `ber` notation may be used to express the value in the hexadecimal representation of its BER encoding.

If the attribute or assertion syntax is OCTET STRING, BIT STRING, UTF8String, Audio, Fax or JPEG, then optionally, values may be taken from an external file using the **`from`** notation, where **`path`** specifies a filename relative to the current working directory, and the attribute value is taken to be the entire contents of the file with that name. For syntaxes other than OCTET STRING, BIT STRING, UTF8String, Audio, Fax or JPEG, the `"BER"` keyword is used in the `from` notation to indicate that the file content is the BER encoding of the value according to its attribute syntax.

NOTE: The syntax of the value must agree with that specified in the attribute definition. The attribute syntax of built-in attributes can be determined from the file `${VFHOME}/setup/schema.defs`. The attribute syntax of user-defined attributes depends on the user's specification – see *Chapter 4*.

attribute

An `attribute` contains an attribute type and a list of one or more attribute values separated by white-space.

```
attribute = type value { value }
```

For examples:

```
telephoneNumber "+61 3 253 1234" "13 2200"
photo FROM "fred.jpg"
userCertificate FROM BER:"fred.cer"
```

rdn

An `rdn` (Relative Distinguished Name) is a sequence of one or more attribute type and value pairs separated by white-space.

```
rdn = type value { type value }
```

For example:

```
organizationName "Deltawing"
```

name

A distinguished `Name` is a sequence of zero, one or more RDNs each separated by a slash, the sequence itself optionally starting with a slash, enclosed in braces.

```
name = "{" [ rdn ] { "/" rdn } "}"
```

Names beginning with a slash (after the left brace) are absolute names (relative to the DIT root). Names without an initial slash are relative to the *prefix* which was set with the `set (set-prefix)` command. The absolute name of the DIT root is a single slash, i.e. `{/}`; its relative name relative to a prefix which is the DIT root is an empty string between the braces, i.e. `{ }`.

NOTE: Initially the prefix is the DIT root, so there is no difference between names beginning with an initial slash and ones without an initial slash. The `set (set-prefix)` command is most useful when `sdua` is used in interactive mode.

For example:

```
{ organizationName "Deltawing" / organizationalUnit "Deltawing
InfoSystems" }
```

content

The `content` argument specifies a comma separated list of attributes.

```
content = "with" "{"
          [ attribute { "," attribute } ]
      "}"
```

path

The `path` argument is a string of characters enclosed in double quote characters specifying the path to a file. If a relative path is specified, it is determined relative to the file system directory `sdua` was started in.

output

The `output` argument specifies a file to receive the result of a `read`, `list` or `search` command.

```
output = "into" path
```

options

Specifies a value for the `CommonArguments` item from the Directory Access Protocol.

```
options      = "options" options-expr
options-expr    = asn1value | parameter { parameter }
parameter    = param [ "=" option-value ]
option-value = on | off | number
on           = "1" | "on" | "true"
off             = "0" | "off" | "false"
```

The `options` argument is mostly used to supply a value for the `serviceControls` component of `CommonArguments` (see next section). This component has syntax `ServiceControls`, and appears as the first member of the `CommonArguments` SET, all other members of that SET being optional. An `asn1value` for `options` can therefore be specified by providing a value for `serviceControls` and enclosing it in an additional pair of braces. *Be careful not to confuse it with the* `options` *field within* `serviceControls`.

For example:

```
{     serviceControls {
          options {localScope},
          timeLimit 2,
          sizeLimit 30
      }
}
```

To simplify the setting of the serviceControls in `options`, any of the following keywords may be used as `param`:

```
preferChaining
chainingProhibited
localScope
dontUseCopy
dontDereferenceAliases
subentries
copyShallDo
manageDSAIT
priority
timeLimit
sizeLimit
scopeOfReferral
attributeSizeLimit
schemaChecking
```

The first eight of these parameters may take an `option-value` of `on` or `off`. If no `option-value` is provided then **`on`** is assumed. The next five parameters must have an `option-value` and expect it to be an integer `number`.

And the last parameter:

```
SchemaChecking ::= ENUMERATED {
      none,
```

```
      ignoreUserModifiableFlag,
      all
}
```

The setting of options using `asn1value` results in all existing settings being replaced with the new commonArguments. The `param` settings will only affect the indicated field in the current `commonArguments`.

string

The `string` argument is an arbitrary string of characters enclosed in double quote characters (`"`). Double-quote characters that appear in the string itself are escaped by being repeated.

Examples:

```
"This is a string."
"This string has a quoted dollar sign ""$""."
```

password

A `password` is a quoted character string representing a user's password.

```
password = string
```

protocol

```
protocol = "dap" | "admin"
```

The `protocol` may be defined as either `"dap"` for the DAP Protocol or `"admin"` for the DAP Admin Protocol.

username

The `username` argument identifies a user of the directory service.

```
username = name | string
```

The `username` may be specified as a DN using the `name` alternative or as a character string enclosed in double quote characters. If the second option is used, sdua will use the string to carry out a getMyDN search to determine the DN to use in the bind. The getMyDN search is carried out during processing of the `bind` command.

delete-old

The `delete-old` argument is used by the `move`, `rename`, `ldap-move`, `ldap-rename`, `xldap-move` and `xldap-rename` commands.

```
delete-old = "delete" "old"
```

If `delete-old` is specified and the last RDN of the new name is different from the last RDN of the old name, the entry's values corresponding to the old RDN are removed. Otherwise they remain as non-naming attributes in the entry.

ldap-oid

An `ldap-oid` is an OBJECT IDENTIFIER value in the conventional dotted decimal representation for LDAP, surrounded by double quotes.

For example:

```
"2.5.13.0"
```

controls

The `controls` argument specifies a list of LDAP controls for any of the `ldap-request` commands.

```
controls = "controls" [ control { "," control } ]
control  = "{" "controlType" ldap-oid
               [ criticality ]
               [ control-value ] "}"
criticality   = "," "criticality"
                    ( "TRUE" | "FALSE" )
control-value = "," "controlValue" octet-string
```

scope

The `scope` argument specifies which entries are searched by a `search`, `ldap-search` or `xldap-search` command.

```
scope = "only" | "one" "level"
```

If the `scope` is omitted, the whole of the subtree below the entry named in the `search`, `ldap-search` or `xldap-search` command is searched. The keyword `"only"` restricts the search to the named entry itself while the keywords `"one" "level"` restrict the search to the subordinates of the named entry. A search for the entry only is similar to the `read` command, and a one level search is similar to the `list` command.

ldap-type

The `ldap-type` argument specifies a single attribute description in an LDAP operation.

```
ldap-type = type | string
```

An attribute type without an LDAP attribute option can be represented by a `type` argument. An attribute type with attribute options must be surrounded by double quotes. The attribute options are separated by semicolons. An attribute type represented as an LDAP dotted decimal object identifier must also be surrounded by double quotes.

Examples:

```
commonName
cn
{ 2 5 4 3 }
"cn;binary"
"2.5.4.3"
```

ldap-value

An ldap-value is an attribute or assertion value in the LDAP protocol.

```
ldap-value = string | octet-string | value  |"FROM" path
```

Values in the LDAP protocol usually appear as human-readable character strings.

If the value belongs to a human-readable syntax in LDAP, then it can be represented as a quoted character string. If the value belongs to an unreadable syntax in LDAP, then it is represented as an octet-string.

Values using the LDAP binary encoding are represented in the usual format for an `sdua value`. An LDAP value may be taken from an external file using the "`FROM`"

notation, where **`path`** specifies a filename relative to the current working directory, and the value is taken to be the entire contents of the file with that name. This final alternative is particularly useful when the LDAP value contains line breaks (for example, if it is an XML document).

ldap-content

The `ldap-content` argument specifies a list of attributes.

```
ldap-content = "with" "{"
                 [ ldap-attribute
                 { "," ldap-attribute } ] "}"
ldap-attribute = ldap-type ldap-value
               { ldap-value }
```

An `ldap-attribute` contains an attribute description specifying the attribute type and a list of one or more attribute values separated by white-space.

Examples:

```
telephoneNumber "+61 3 253 1234" "13 2200",
postalAddress "250 Bay Street$Brighton$Victoria"
"postalAddress;binary" {
      "250 Bay Street",
      "Brighton",
      "Victoria" }
"userCertificate;binary" FROM BER:"fred.cer"
```

ldapdn

An `ldapdn` is the DN of an entry in an **`ldap-request`**.

```
ldapdn = string | name
```

An `ldapdn` can be represented in the conventional character string format for an LDAP DN, surrounded by quotes (that is, as a `string`) or in the usual format for a Stream DUA `name`.

Examples:

```
"cn=Fred Smith,o=Deltawing"
{ organizationName "Deltawing" / commonName "Fred Smith" }
```

xldap-controls

The `xldap-controls` argument specifies a list of XLDAP controls for any of the `xldap-request` commands.

```
xldap-controls = "controls" "{" [ xldap-control { "," xldap-control } ] "}"
xldap-control  = "{" "controlType" object-identifier
                 [ criticality ]
                 [ control-value ] "}"
criticality    = "," "criticality"
                ( "TRUE" | "FALSE" )
control-value = "," "controlValue" "request:" value
```

xldap-content

The `xldap-content` argument specifies a list of attributes.

```
xldap-content = "with" "{"
                   [ xldap-attribute
                       { "," xldap-attribute } ]
```

```
                    "}"
xldap-attribute = ldap-type value { value }
```

`xldap-attribute` contains an attribute type and a list of one or more attribute values separated by white-space.

Examples:

```
telephoneNumber "+61 3 253 1234" "13 2200",
postalAddress "250 Bay Street$Brighton$Victoria"
"postalAddress;binary" {
    "250 Bay Street",
    "Brighton",
    "Victoria"
}
"userCertificate;binary" FROM BER:"fred.cer"
{ 2 5 4 12 } "My Title"
```

Service controls

Most operations sent by Stream DUA to the DSA include service controls (set via the `options` argument). Service controls are standard requests to modify the service which the DSA would otherwise perform.

The ASN.1 definition of `ServiceControls` is as follows (from X.500).

```
ServiceControls  ::= SET {
      options               [0] BIT STRING {
      preferChaining        (0),
      chainingProhibited    (1),
      localScope            (2),
      dontUseCopy           (3),
      dontDereferenceAliases(4),
      subentries            (5),
      copyShallDo           (6),
      manageDSAIT           (8) }  DEFAULT {},
      priority                  [1] INTEGER {low(0), medium(1),
                                high(2)}  DEFAULT medium,
      timeLimit                 [2] INTEGER  OPTIONAL,
      sizeLimit                 [3] INTEGER  OPTIONAL,
      scopeOfReferral           [4] INTEGER {dmd(0), country(1)}
                                OPTIONAL,
      attributeSizeLimit        [5] INTEGER  OPTIONAL,
      manageDSAITPlaneRef       [6] SEQUENCE {
          dsaName               Name,
          agreementID           AgreementID }
AgreementID ::= SEQUENCE {
      identifier        INTEGER,
      version           INTEGER }
```

The fields are described below.

`preferChaining`	Not used by ViewDS. ViewDS DSAs always try to chain unless prevented by either of the next two service controls (see *Chapter 7*).
`chainingProhibited`	Requests the DSA not to chain under any circumstances.
`localScope`	Requests the DSA to chain only to DSAs within the 'local scope' (defined in ViewDS using the `dsaCollaborators` attribute described in *Chapter 3*).
`dontUseCopy`	Requests the DSA to ignore replicated copies of an entry and chain the request to the supplier of the replicated information instead.
`dontDereference Aliases`	Requests the DSA to not to dereference aliases: if the named entry is an alias entry, it is returned rather than the entry to which it refers.
`copyShallDo`	Requests the DSA to use replicated copies of entries, even when it would normally determine the copies to be unsuitable for evaluation of the request.
`manageDSAIT`	A 1997 extension to X.500 supported by ViewDS. If asserted, DSA information tree management is enabled.
`priority`	Not used by ViewDS.
`timeLimit`	Specifies the time within which an operation is to be completed. If the operation cannot be carried out within this time, the DSA returns a service error.
`sizeLimit`	Specifies the maximum number of entries to be returned. If this limit is exceeded, the DSA may return some of the requested entries (up to this limit) or it may return an error.
`scopeOfReferral`	Not used by ViewDS.
`attributeSizeLimit`	Specifies the largest size of any attribute value to be included in the returned entry information. If an attribute exceeds this limit, all of its values are omitted from the returned entry information. See *Rec. X.511 | 9594-3 clause 7.6*.
`manageDSAITPlaneRef`	A 1997 extension to X.500 supported by ViewDS. It is used to specify a specific replication plane to act on.

Appendix B
Supported schema

This chapter specifies the pre-defined schema supported by ViewDS.

Introduction

This chapter specifies selected aspects of the pre-defined schema available in any ViewDS installation. ViewDS's schema is highly extensible (only the set of supported matching rules is fixed).

The pre-defined schema covers all of X.500, all of X.400, and the ViewDS extensions. It is used when adding initial information to a ViewDS DSA (for example, when setting up a subschema administrative point entry), and as a cross-check that schema publication information added by the user for standard schema objects is in fact correct.

The pre-defined schema is supplied in text format (the GSER encoding of the `SET OF Attribute` ASN.1 type) and their file names have the .txt extension. For backward compatibility purposes, the pre-defined schema is supported in binary format.

The following pre-defined schema is installed with the ViewDS Management Agent:

- `ACP133.txt` – schema defined in ACP133 edition B.
- `ACP133C.txt` – schema defined in ACP133 edition C.
- `Clearswift.txt` – schema defined for Clearswift Directory Bastion test bed.
- `Corporate.txt` – schema for the Deltawing demonstration directory supplied with ViewDS.
- `NADF.txt` – schema defined for North American Directory Forum (NADF) profile.
- `RFC1274.txt` – using COSINE and Internet X.500 schema.
- `RFC2247.txt` – using Domains in LDAP/X.500 Distinguished Names.
- `RFC2307.txt` – an Approach for using LDAP as a Network Information Service.
- `RFC2798.txt` – definition of the inetOrgPerson LDAP Object Class.
- `RFC4523.txt` – LDAP Schema Definitions for X.509 Certificates.
- `RFC4524.txt` – COSINE LDAP/X.500 Schema (obsoletes RFC 1274, updates RFC 2247 and RFC 2798).
- `X.500-Default.txt` – standard X.500 schema defined in X.520 and X.521.
- `X.509-PKI.txt` – standard X.500 schema defined in X.509.

Object identifier prefixes

The object identifiers in the tables that follow are specified as a symbolic prefix and a numeric suffix. The symbolic prefix represents a sequence of numbers, and the actual

object identifier is formed by concatenating that sequence with the sequence forming the numeric suffix. The prefixes are as follows:

Prefix	Definition
ds	2 5
mhs	2 6
ldap	1 3 6 1 4 1 1466
vf	1 3 32 0 1
ads	1 2 36 79672281 1
vds	1 3 6 1 4 1 21473 5
sun-at	1 3 6 1 4 1 42 2 27 8 1

Attribute and assertion syntaxes

The ViewDS DSA supports all attribute and assertion syntaxes defined in X.520 (1993) and Annex C of X.402 (1988). It also supports all attribute and assertion syntaxes for operational attributes defined in X.501, X.509, X.518, and X.525, and a number of ViewDS-specific attribute and assertion syntaxes.

Attribute syntaxes as used in X.500 (1988) have been superseded for X.500 (1993) with use of straight ASN.1 in the attribute definitions and the definition of standard matching rules. The supported attribute and assertion syntaxes are sufficient to conform with X.520 (1988).

The set of supported attribute and assertion syntaxes is as follows.

Standard syntaxes

The ViewDS DSA supports the following built-in attribute and assertion syntaxes:

Attribute or Assertion syntax	Reference
ASN.1 built-in syntaxes	
ANY	X.680
BIT STRING	X.680
BMPString	X.680
BOOLEAN	X.680
CHARACTER STRING	X.680
EMBEDDED PDV	X.680
ENUMERATED	X.680
EXTERNAL	X.680
GeneralizedTime	X.680
GeneralString	X.680
GraphicString	X.680
IA5String	X.680
INTEGER	X.680
NULL	X.680
NumericString	X.680
ObjectDescriptor	X.680
OBJECT IDENTIFIER	X.680
OCTET STRING	X.680
PrintableString	X.680

Attribute or Assertion syntax	Reference
REAL	X.680
TeletexString	X.680
UTCTime	X.680
VideotexString	X.680
VisibleString	X.680
UniversalString	X.680
UTF8String	X.680
Directory syntaxes	
AccessPoint	X.518 Annex A
ACIItem	X.501 Annex D
AlgorithmIdentifier	X.509 Annex A
AttributeTypeDescription	X.501 Annex C
Certificate *	X.509 Annex A
CertificateAssertion	X.509 Annex A
CertificateExactAssertion	X.509 Annex A
CertificateList *	X.509 Annex A
CertificateListAssertion	X.509 Annex A
CertificateListExactAssertion	X.509 Annex A
CertificatePair	X.509 Annex A
CertificatePairAssertion	X.509 Annex A
CertificatePairExactAssertion	X.509 Annex A
ComponentFilter	RFC 3687
ConsumerInformation	X.501 Annex E
CountryName	X.520 Annex A
DestinationIndicator	X.520 Annex A
DirectoryString{}	X.520 Annex A
DistinguishedName	X.501 Annex B
DITContentRuleDescription	X.501 Annex C
DITStructureRuleDescription	X.501 Annex C
DSEType	X.501 Annex E
EnhancedGuide	X.520 Annex A
FacsimileTelephoneNumber	X.520 Annex A
FamilyEntries	X.511 Annex A
Guide	X.520 Annex A
InfoSyntax	X.509 Annex A
InternationalISDNNumber	X.520 Annex A
LDAPSchemaDefintiion	RFC 2927
LDAPSyntaxDescription	draft-ietf-ldapbis-syntaxes-xx.txt
MasterAndShadowAccessPoints	X.525 Annex A
MatchingRuleDescription	X.501 Annex C
MatchingRuleUseDescription	X.501 Annex C
NameAndOptionalUID	X.501 Annex D
NameFormDescription	X.501 Annex C
ObjectClassDescription	X.501 Annex C

Attribute or Assertion syntax	Reference
OctetSubstringAssertion	X.520 Annex A
PkiPath	X.509 Annex A
PolicySyntax	X.509 Annex A
PostalAddress	X.520 Annex A
PreferredDeliveryMethod	X.520 Annex A
PresentationAddress	X.520 Annex A
ProtocolInformation	X.520 Annex A
RelativeDistinguishedName	X.501 Annex B
SubjectPublicKeyInfo	X.509 Annex A
SubstringAssertion	X.520 Annex A
SubtreeSpecification	X.501 Annex B
SupplierAndConsumers	X.501 Annex E
SupplierInformation	X.501 Annex E
SupplierOrConsumer	X.501 Annex F
SupportedAlgorithm	X.509 Annex A
TelephoneNumber	X.520 Annex A
TeletexTerminalIdentifier	X.520 Annex A
TelexNumber	X.520 Annex A
Time	X.509 Annex A
UniqueIdentifier	X.501 Annex D
X121Address	X.520 Annex A
X.400 attribute syntaxes	
DLSubmitPermission	X.411
ORAddress	X.411
ORName	X.411
RequestedDeliveryMethod	X.411
XED Syntaxes	
AnyType	draft-legg-xed-rxer-xx.txt
IdentifiedSchema	draft-legg-xed-schema-xx.txt
SchemaIdentity	draft-legg-xed-schema-xx.txt
PKCS attribute syntaxes	
CertificationRequest	PKCS #10
Certs	PKCS #7
ContentInfo	PKCS #7
Data	PKCS #7
DigestedData	PKCS #7
EncryptedMACData	PKCS #7
EnvelopedData	PKCS #7
FileDetails	PKCS #9
IndirectDataContent	PKCS #7
PBES2-params	PKCS #5
PBEParameter	PKCS #5
PBKDF2-params	PKCS #5
PBMAC1-params	PKCS #5

Attribute or Assertion syntax	Reference
PKCS7TextParameters	PKCS #7
PKCS9String	PKCS #9
RC2-CBC-Parameter	PKCS #5
RC5-CBC-Parameters	PKCS #5
SignedData	PKCS #7
SignedPublicKeyAndChallenge	PKCS #10
QUIPU and other attribute syntaxes	
ACLSyntax	QUIPU source code
AttributeCertificate *	ANSI X9.57
Audio	QUIPU source code
AuthenticationPolicySyntax	QUIPU source code
BootParameterSyntax	RFC 2307
Call	QUIPU source code
CcMailAttributes	Nortel
DSAQualitySyntax	QUIPU source code
EDBInfoSyntax	QUIPU source code
InheritedAttribute	QUIPU source code
JPEG	QUIPU source code
ListACLSyntax	QUIPU source code
MsMailAttributes	Nortel
NISNetgroupTripleSyntax	RFC 2307
OtherMailbox	RFC 1274
QuipuProtectedPassword	QUIPU source code
SearchACLSyntax	QUIPU source code
SecurityPolicy	QUIPU source code
TreeStructureSyntax	QUIPU source code

* The DSA preserves and returns the original uploaded encoding for these attributes.

ViewDS-specific syntaxes

The ViewDS DSA supports the following built-in ViewDS-specific attribute and assertion syntaxes, used with ViewDS operational attributes:

ViewDS-specific syntax	Reference
AnonymousPrivilege	*Chapter 6*
AnonymousPrivilegeAssertion	
ATTRIBUTE.&id (an OBJECT IDENTIFIER for an attribute type)	*X.501 Annex B*
AttributePresentation	*Chapter 6*
AttributeTypeExtension	*Chapter 5*
AuthenticationLevel	*X.501 Annex E*
BaseObjectIndexing	*Chapter 5*
CaseIgnoreList	
ConsumerStatus	*Chapter 8*
DSACollaborator	*Chapter 8*
DSACollaboratorAssertion	

ViewDS-specific syntax	Reference
DUABanners	*Chapter 6*
EntrustCAInfo	
Fax	
GeneralDate	*Time type definitions* (page 231)
GeneralDateTime	*Time type definitions* (page 231)
GeneralTimeOfDay	*Time type definitions* (page 231)
HierarchyNameSpecification	*Chapter 6*
IndexDescription	*Chapter 5*
KeySize	*X.509 Annex F*
MailPreference	
MATCHING-RULE.&id	*X.501 Annex B*
(an OBJECT IDENTIFIER for a matching rule)	
ModifyRight	
Name	*X.501 Annex B*
NAME-FORM.&id	*X.501 Annex B*
(an OBJECT IDENTIFIER for a name form)	
NewSubordinateModifyRights	
OBJECT-CLASS.&id	*X.501 Annex B*
(an OBJECT IDENTIFIER for an object class)	
ObjectClassPresentation	*Chapter 6*
ODataDate	*Time type definitions* (page 231)
ODataDateTimeOfDay	*Time type definitions* (page 231)
ODataDateTimeOffset	*Time type definitions* (page 231)
OTHER-OBJECT-IDENTIFIER	
(an OBJECT IDENTIFIER that is not for an attribute type, matching rule, object class or name form)	
PasswordEncryption	*Chapter 6*
PermittedNewSubordinates	
PlaneKnowledge	
PlaneRef	*Chapter 8*
Privilege	*Chapter 6*
RemoteAlias	*Chapter 8*
ResolvedDistinguishedName	*Chapter 6*
SASLDigestMD5Assertion	
SearchOptions	*Chapter 6*
SEPType	*Chapter 8*
ShadowStatusIdentifier	
SupplierOrConsumerInformationAssertion	
SupplierStatus	*Chapter 8*
UnsignedCertificate	
UserConfig	*Chapter 6*
UserGroup	*Chapter 6*
UserEntitlement	*Chapter 6*
WordList	*Chapter 5*

Time type definitions

The date and time syntaxes are derived from the ASN.1 TIME type described in Amendment 3 to ITU-T Rec. X.680:2002 | ISO/IEC 8824-1:2002.

The ASN.1 definitions are as follows:

```
GeneralDateTime ::= TIME {
      (SETTINGS "Basic=Date-Time Date=YMD Midnight=Start")
}

GeneralDate ::= TIME {
      (SETTINGS "Basic=Date Date=YMD")
}

GeneralTimeOfDay ::= TIME {
      (SETTINGS "Basic=Time Midnight=Start")
}

ODataDateTimeOffset ::= TIME {
      (SETTINGS "Basic=Date-Time Date=YMD Time=HMSF12 Midnight=Start" ^
      (SETTINGS "Local-or-UTC=Z" | SETTINGS "Local-or-UTC=LD"))
}

ODataDate ::= TIME {
      (SETTINGS "Basic=Date Date=YMD Local-or-UTC=L Midnight=Start")
}

ODataTimeOfDay ::= TIME {
      (SETTINGS "Basic=Time Time=HMSF12 Local-or-UTC=L Midnight=Start")
}
```

Matching rules

Standard matching rules

The ViewDS DSA supports all matching rules defined in the X.500-series (1993) and X.402 (1988) Recommendations. These are as follows:

Matching Rule	OID	Reference
objectIdentifierMatch	ds 13 0	X.501 Annex B
distinguishedNameMatch	ds 13 1	X.501 Annex B
X.520 matching rules		
caseIgnoreMatch	ds 13 2	X.520 Annex A
caseIgnoreOrderingMatch	ds 13 3	X.520 Annex A
caseIgnoreSubstringsMatch	ds 13 4	X.520 Annex A
caseExactMatch	ds 13 5	X.520 Annex A
caseExactOrderingMatch	ds 13 6	X.520 Annex A
caseExactSubstringsMatch	ds 13 7	X.520 Annex A
numericStringMatch	ds 13 8	X.520 Annex A
numericStringOrderingMatch	ds 13 9	X.520 Annex A
numericStringSubstringsMatch	ds 13 10	X.520 Annex A
caseIgnoreListMatch	ds 13 11	X.520 Annex A
caseIgnoreListSubstringsMatch	ds 13 12	X.520 Annex A
booleanMatch	ds 13 13	X.520 Annex A

Matching Rule	OID	Reference
integerMatch	ds 13 14	X.520 Annex A
integerOrderingMatch	ds 13 15	X.520 Annex A
bitStringMatch	ds 13 16	X.520 Annex A
octetStringMatch	ds 13 17	X.520 Annex A
octetStringOrderingMatch	ds 13 18	X.520 Annex A
octetStringSubstringsMatch	ds 13 19	X.520 Annex A
telephoneNumberMatch	ds 13 20	X.520 Annex A
telephoneNumberSubstringsMatch	ds 13 21	X.520 Annex A
presentationAddressMatch	ds 13 22	X.520 Annex A
uniqueMemberMatch	ds 13 23	X.520 Annex A
protocolInformationMatch	ds 13 24	X.520 Annex A
uTCTimeMatch	ds 13 25	X.520 Annex A
uTCTimeOrderingMatch	ds 13 26	X.520 Annex A
generalizedTimeMatch	ds 13 27	X.520 Annex A
generalizedTimeOrderingMatch	ds 13 28	X.520 Annex A
integerFirstComponentMatch	ds 13 29	X.520 Annex A
objectIdentifierFirstComponentMatch	ds 13 30	X.520 Annex A
directoryStringFirstComponentMatch	ds 13 31	X.520 Annex A
wordMatch	ds 13 32	X.520 Annex A
keywordMatch	ds 13 33	X.520 Annex A
facsimileNumberMatch	ds 13 63	X.520 Annex A
facsimileNumberSubstringsMatch	ds 13 64	X.520 Annex A
Knowledge matching rules		
accessPointMatch	ds 14 0	X.501 Annex E
masterAndShadowAccessPointsMatch	ds 14 1	X.501 Annex E
supplierOrConsumerInformationMatch	ds 14 2	X.501 Annex E
supplierAndConsumersMatch	ds 14 3	X.501 Annex E
X.400 matching rules		
orAddressElementsMatch	mhs 4 8 13	X.402
orAddressMatch	mhs 4 8 14	X.402
orNameElementsMatch	mhs 4 8 16	X.402
oRNameExactMatch	mhs 5 0	X.402
orNameMatch	mhs 4 8 17	X.402
oRNameSingleElementMatch	mhs 4 8 18	X.402
oRAddressSubstringElementsMatch	mhs 4 8 15	X.402
oRNameSubstringElementsMatch	mhs 4 8 19	X.402
X.509 (1997) matching rules		
algorithmIdentifierMatch	ds 13 40	X.509 (1997)
certificateExactMatch	ds 13 34	X.509 (1997)
certificateListExactMatch	ds 13 38	X.509 (1997)
certificateListMatch	ds 13 39	X.509 (1997)
certificateMatch	ds 13 35	X.509 (1997)
certificatePairExactMatch	ds 13 36	X.509 (1997)
certificatePairMatch	ds 13 37	X.509 (1997)

Matching Rule	OID	Reference
LDAP matching rules		
caseExactIA5Match	ldap 109 114 1	RFC 2252
caseIgnoreIA5Match	ldap 109 114 2	RFC 2252
caseIgnoreIA5SubstringsMatch	ldap 109 114 3	RFC 2798
componentFilterMatch	ads 13 2	RFC 3687
rdnMatch	ads 13 3	RFC 3687
presentMatch	ads 13 5	RFC 3687
allComponentsMatch	ads 13 6	RFC 3687
directoryComponentsMatch	ads 13 7	RFC 3687
XED matching rules		
schemaIdentityMatch	ads 13 10	draft-legg-xed-schema-xx.txt

ViewDS-specific matching rules

Matching Rule	OID	Reference
anonymousPrivilegeMatch	vf 13 12	Below
asn1Match	vf 13 0	Below
asn1OrderingMatch	vf 13 1	Below
defaultApproximateMatch	vf 13 8	Below
defaultEqualityMatch	vf 13 4	Below
defaultGreaterOrEqualMatch	vf 13 7	Below
defaultLessOrEqualMatch	vf 13 6	Below
defaultPresentMatch	vf 13 3	Below
defaultSubstringsMatch	vf 13 5	Below
dsaCollaboratorMatch	vf 13 2	Below
dsaPasswordMatch	vf 13 13	Below
indexDescriptionMatch	vf 13 19	Below
literalStringMatch	vds 13 2	Below
literalStringOrderingMatch	vds 13 3	Below
literalStringSubstringsMatch	vds 13 31	Below
oldGeneralWordMatch	vf 13 17	Below
sepTypeMatch	vf 13 14	Below
shadowStatusMatch	vf 13 16	Below
undefinedMatch	ads 13 8	Below
saslDigestMD5Match	ads 13 9	Below
dateTimeMatch	vds 13 7	Below
dateTimeOrderingMatch	vds 13 8	Below
timeOfDayMatch	vds 13 9	Below
timeOfDayOrderingMatch	vds 13 10	Below
dayOfWeekMatch	vds 13 11	Below
dayOfWeekOrderingMatch	vds 13 12	Below

Each is described below.

anonymousPrivilegeMatch

An `anonymousPrivilegeMatch` is true if two values of `anonymousPrivilege` have at least one common bit set in both the `protocol` and `credentialType` bit strings. Its assertion syntax is `AnonymousPrivilegeAssertion`.

```
AnonymousPrivilegeAssertion ::= SET {
      protocol       [0] BIT STRING,
      credentialType [1] BIT STRING }
```

asn1Match

An `asn1Match` is true if two values have the same ASN.1 encoding, ignoring differences in set ordering. Its assertion syntax is an open type.

asn1OrderingMatch

An `asn1OrderingMatch` is true if the attribute value being compared is determined to be 'less than' the presented value. Primitive ASN.1 components (INTEGER, OCTET STRING etc) are compared using normal rules for their type. SET, SEQUENCE, and CHOICE types are compared by comparing their components in definition order. A missing OPTIONAL component or a later CHOICE is considered to be less than a present component or an earlier CHOICE. A missing DEFAULT component is compared as though it is a missing OPTIONAL component. The ASN.1 constructs SET OF and SEQUENCE OF are compared by comparing member values in turn. A SET OF is first reordered so lesser values lie ahead of greater values. If one SET OF or SEQUENCE OF is a prefix of the other, the shorter is considered to less than the longer. The assertion syntax of this rule is an open type.

defaultEqualityMatch

A `defaultEqualityMatch` is true if the values match for equality using the default equality matching rule for the attribute type. Its assertion syntax is the same as the syntax of the attribute to which it is applied.

defaultSubstringsMatch

A `defaultSubstringsMatch` is true if a `substrings` match applied to the attribute would return true. Its assertion syntax is same as the `strings` component of the `substrings` component of `FilterItem`.

defaultLessOrEqualMatch

A `defaultLessOrEqualMatch` is true if a `lessOrEqual` match applied to the attribute would return true. Its assertion syntax is the same as the syntax of the attribute to which it is applied.

defaultGreaterOrEqualMatch

A `defaultGreaterOrEqualMatch` is true if a `greaterOrEqual` match applied to the attribute would return true. Its assertion syntax is the same as the syntax of the attribute to which it is applied.

defaultApproximateMatch

A `defaultApproximateMatch` is true if an `approximateMatch` match applied to the attribute would return true. Its assertion syntax is the same as the syntax of the attribute to which it is applied.

defaultPresentMatch

A `defaultPresentMatch` is true if the attribute type is present in the entry. Its assertion syntax is `NULL`.

dsaCollaboratorMatch

A `dsaCollaboratorMatch` is true if two values of `dsaCollaborators` have DSA names that match its assertion syntax is `DSACollaboratorAssertion`.

```
DSACollaboratorAssertion ::= SET {
      dsa-name       [0] Name,
      password       [1] OCTET STRING }
```

dsaPasswordMatch

A `dsaPasswordMatch` is true if two values of `dsaCollaborators` have DSA names that match and passwords that match. Its assertion syntax is `DSACollaboratorAssertion`.

indexDescriptionMatch

An `indexDescriptionMatch` is true if the `type` and `index` fields of the assertion value match the respective fields in an attribute value with syntax `IndexDescription`. The assertion syntax of this matching rule is `IndexDescription`.

literalStringMatch

A `literalStringMatch` is true if the string attribute value and string assertion value are the same length and corresponding characters have the same Unicode code point. No transformations are applied to the strings before comparison.

literalStringOrderingMatch

A `literalStringOrderingMatch` is true if the string attribute value is lexicographically before the string assertion value, ordering on Unicode code points. No transformations are applied to the strings before comparison.

literalStringSubstringsMatch

The `literalStringSubstringsMatch` uses the `SubstringAssertion` syntax. A `literalStringSubstringsMatch` is `true` if there is a partitioning of the attribute value (into portions) such that the specified substrings (`initial`, `any`, `final`) match different portions of the value in the order of the substrings' sequence:

- `initial`, if present, matches the first portion of the value.
- `final`, if present, matches the last portion of the value.
- `any`, if present, matches some arbitrary portion of the value.

For a component of substrings to match a portion of the attribute value the corresponding characters must have the same Unicode code point. No transformations are applied to the strings before comparison.

oldGeneralWordMatch

An `oldGeneralWordMatch` is true if the assertion string matches an attribute string according to the rules for general word matches (see F.510 and the X.500 (1999) working document). Its assertion syntax is `GeneralWordAssertion`.

```
GeneralWordAssertion ::= SEQUENCE {
      string             DirectoryString {ub-match},
```

```
      sequenceMatchType  ENUMERATED {
                             sequenceExact (0),
                             sequenceDeletion (1),
                             sequenceRestrictedDeletion (2),
                             sequenceRotation (3),
                             sequenceRotationAndDeletion (4) }
                             DEFAULT sequenceExact,
      wordMatchTypes     SEQUENCE OF ENUMERATED {
                             wordExact (0),
                             wordTruncated (1),
                             wordPhonetic (2) } DEFAULT { wordExact },
      characterMatchType ENUMERATED {
                             characterExact (0),
                             characterCaseIgnore (1),
                             characterMapped (2) }
                             DEFAULT characterExact}
```

sepTypeMatch

A `sepTypeMatch` is true if two values of sepType are the same enumerated value of the `SEPType` syntax. Its assertion syntax is `SEPType`.

shadowStatusMatch

A `shadowStatusMatch` is true if two values of type ShadowStatus refer to the same DSA and have the same identifier. Its assertion syntax is `ShadowStatusIdentifier`.

```
ShadowStatusIdentifier ::= SEQUENCE {
      dsaName        [0] Name,
      identifier     [1] INTEGER
}
```

undefinedMatch

An `undefinedMatch` is always undefined (i.e. neither `True` nor `False`). This matching rule has been included in order to handle the chaining of search operations which have been converted from LDAP to DAP, where some attribute types and/or matching rules used in the search filter are not recognized in the X.500 schema. X.500 filter evaluation requires these parts of the filter to be evaluated as undefined and this matching rule provides a mechanism to pass this information on to other DSAs through DSP even though we cannot provide the object identifier for the attribute or matching rule which failed conversion.

saslDigestMD5Match

The `saslDigestMD5Match` matching rule is used to allow ViewDS to chain LDAP SASL "DIGEST-MD5" bind requests to other DSAs, using the distributed operations, without the need to transmit clear-text passwords over the network. It accomplishes this by collecting the SASL bind arguments and the server details required to verify the SASL authentication request and passing this information on to the DSA, that has knowledge of the clear-text password for the identity requesting authentication, using a search operation. The response to this search operation will include an operational attribute holding the information required to form a valid SASL response to this authentication request.

The assertion syntax for this matching rule is:

```
SASLDigestMD5Assertion ::= SEQUENCE {
      username           [0] OCTET STRING,
      realm              [1] OCTET STRING OPTIONAL,
      nonce              [2] OCTET STRING,
      cnonce             [3] OCTET STRING,
      nonce-count        [4] OCTET STRING OPTIONAL,
      qop                [5] OCTET STRING OPTIONAL,
      digest-uri         [6] OCTET STRING OPTIONAL,
      response           [7] OCTET STRING,
      maxbuf             [8] OCTET STRING OPTIONAL,
      charset            [9] OCTET STRING OPTIONAL,
      authzid            [10]OCTET STRING OPTIONAL
}
```

The fields in this syntax correspond to the SASL "DIGEST-MD5" fields described in *RFC2831 Using Digest Authentication as a SASL Mechanism*.

dateTimeMatch

The equality matching rule for the `currentDateTime` and `userDateTime` operational attributes.

When matching the `currentDateTime` attribute:

- If the assertion value is a UTC time (that is, `Z` is the last character) or it has a time-zone difference, the comparison is on UTC times.
- If the assertion value is a local time (that is, no `Z` at the end or time-zone difference), only the local part of the attribute value is compared. This allows assertions against either UTC time or the server's local time using the one operational attribute.

Values of the `userDateTime` attribute are always local times and are compared as such.

dateTimeOrderingMatch

The ordering matching rule for `currentDateTime` and `userDateTime` operational attributes.

timeOfDayMatch

The equality matching rule for the `currentTimeOfDay` and `userTimeOfDay` operational attributes. It compares two time of day values in ISO 8601 format for equality. It operates in the same way as `dateTimeMatch` with regard to time zones.

timeOfDayOrderingMatch

The ordering matching rule for the `currentTimeOfDay` and `userTimeOfDay` operational attributes.

dayOfWeekMatch

The equality matching rule for the `currentDayOfWeek` and `userDayOfWeek` operational attributes.

dayOfWeekOrderingMatch

The ordering matching rule for the `currentDayOfWeek` and `userDayOfWeek` operational attributes. Monday is the first day of the week and has the least value; Sunday is the last day of the week and has the highest value.

User attributes

The ViewDS DSA can store any attribute that is specified using a supported attribute syntax and supported matching rules. In particular, it can store all attributes of X.520 (1988) and X.520 (1993), and all attributes of Annex C of X.402 (1994). Built-in definitions are as follows.

Standard user attributes

Standard Attribute	OID	Reference
System attributes		
objectClass	ds 4 0	X.501 Annex B
aliasedEntryName	ds 4 1	X.501 Annex B
knowledgeInformation	ds 4 2	X.520 Annex A
Naming attributes		
commonName	ds 4 3	X.520 Annex A
surname	ds 4 4	X.520 Annex A
serialNumber	ds 4 5	X.520 Annex A
name	ds 4 41	X.520 Annex A
givenName	ds 4 42	X.520 Annex A
initials	ds 4 43	X.520 Annex A
generationQualifier	ds 4 44	X.520 Annex A
uniqueIdentifier	ds 4 45	X.520 Annex A
dnQualifier	ds 4 46	X.520 Annex A
Geographic attributes		
countryName	ds 4 6	X.520 Annex A
localityName	ds 4 7	X.520 Annex A
collectiveLocalityName	ds 4 7 1	X.520 Annex A
stateOrProvinceName	ds 4 8	X.520 Annex A
collectiveStateOrProvinceName	ds 4 8 1	X.520 Annex A
streetAddress	ds 4 9	X.520 Annex A
collectiveStreetAddress	ds 4 9 1	X.520 Annex A
houseIdentifier	ds 4 51	X.520 Annex A
Organizational attributes		
organizationName	ds 4 10	X.520 Annex A
collectiveOrganizationName	ds 4 10 1	X.520 Annex A
organizationalUnitName	ds 4 11	X.520 Annex A
collectiveOrganizationalUnitName	ds 4 11 1	X.520 Annex A
Explanatory attributes		
title	ds 4 12	X.520 Annex A
description	ds 4 13	X.520 Annex A
searchGuide	ds 4 14	X.520 Annex A

Standard Attribute	OID	Reference
businessCategory	ds 4 15	X.520 Annex A
enhancedSearchGuide	ds 4 47	X.520 Annex A
Postal attribute		
postalAddress	ds 4 16	X.520 Annex A
collectivePostalAddress	ds 4 16 1	X.520 Annex A
postalCode	ds 4 17	X.520 Annex A
collectivePostalCode	ds 4 17 1	X.520 Annex A
postOfficeBox	ds 4 18	X.520 Annex A
collectivePostOfficeBox	ds 4 18 1	X.520 Annex A
physicalDeliveryOfficeName	ds 4 19	X.520 Annex A
collectivePhysicalDeliveryOfficeName	ds 4 19 1	X.520 Annex A
Telecommunications atts		
telephoneNumber	ds 4 20	X.520 Annex A
collectiveTelephoneNumber	ds 4 20 1	X.520 Annex A
telexNumber	ds 4 21	X.520 Annex A
collectiveTelexNumber	ds 4 21 1	X.520 Annex A
teletexTerminalIdentifier	ds 4 22	X.520 Annex A
collectiveTeletexTerminalIdentifier	ds 4 22 1	X.520 Annex A
facsimileTelephoneNumber	ds 4 23	X.520 Annex A
collectiveFacsimileTelephoneNumber *	ds 4 23 1	X.520 Annex A
x121Address	ds 4 24	X.520 Annex A
internationalISDNNumber	ds 4 25	X.520 Annex A
collectiveInternationalISDNNumber	ds 4 25 1	X.520 Annex A
registeredAddress	ds 4 26	X.520 Annex A
destinationIndicator	ds 4 27	X.520 Annex A
preferredDeliveryMethod	ds 4 28	X.520 Annex A
OSI attributes		
presentationAddress	ds 4 29	X.520 Annex A
supportedApplicationContext	ds 4 30	X.520 Annex A
protocolInformation	ds 4 48	X.520 Annex A
Relational attributes		
member	ds 4 31	X.520 Annex A
owner	ds 4 32	X.520 Annex A
roleOccupant	ds 4 33	X.520 Annex A
seeAlso	ds 4 34	X.520 Annex A
distinguishedName	ds 4 49	X.520 Annex A
uniqueMember	ds 4 50	X.520 Annex A
Security attributes		
userPassword	ds 4 35	X.509 Annex A
userCertificate	ds 4 36	X.509 Annex A
cACertificate	ds 4 37	X.509 Annex A
authorityRevocationList	ds 4 38	X.509 Annex A
certificateRevocationList	ds 4 39	X.509 Annex A
crossCertificatePair	ds 4 40	X.509 Annex A

Standard Attribute	OID	Reference
X.400 attributes		
mhs-deliverable-content-length	mhs 5 2 0	X.402 Annex C
mhs-deliverable-content-types	mhs 5 2 1	X.402 Annex C
mhs-deliverable-eits	mhs 5 2 2	X.402 Annex C
mhs-dl-archive-service	mhs 5 2 12	X.402 Annex C
mhs-dl-members	mhs 5 2 3	X.402 Annex C
mhs-dl-policy	mhs 5 2 13	X.402 Annex C
mhs-dl-related-lists	mhs 5 2 14	X.402 Annex C
mhs-dl-submit-permissions	mhs 5 2 4	X.402 Annex C
mhs-dl-subscription-service	mhs 5 2 15	X.402 Annex C
mhs-message-store-dn	mhs 5 2 5	X.402 Annex C
mhs-or-addresses	mhs 5 2 6	X.402 Annex C
mhs-or-addresses-with-capabilities	mhs 5 2 16	X.402 Annex C
mhs-supported-attributes	mhs 5 2 10	X.402 Annex C
mhs-supported-automatic-actions	mhs 5 2 8	X.402 Annex C
mhs-supported-content-types	mhs 5 2 9	X.402 Annex C
mhs-supported-matching-rules	mhs 5 2 11	X.402 Annex C
mhs-undeliverable-eits	mhs 5 2 17	X.402 Annex C

ViewDS-specific user attributes

ViewDS defines a small number of user attributes which are used by Access Presence.

ViewDS-Specific User Attribute	OID	Reference
sortSubs	vf 4 0	Technical Reference Guide: User Interfaces
hierarchyName	vf 4 1	Technical Reference Guide: User Interfaces
unabbreviatedHierarchyName	ads 4 0	Technical Reference Guide: User Interfaces

Operational attributes

The operational attributes supported by ViewDS are of two kinds: standard operational attributes defined in the X.500 Recommendations, and ViewDS-specific operational attributes that have meaning only for components of ViewDS.

Standard operational attributes

Standard Operational Attribute	OID	Reference
Information model		
createTimestamp	ds 18 1	X.501 Annex B
modifyTimestamp	ds 18 2	X.501 Annex B
creatorsName	ds 18 3	X.501 Annex B
modifiersName	ds 18 4	X.501 Annex B
administrativeRole	ds 18 5	X.501 Annex B
subtreeSpecification	ds 18 6	X.501 Annex B
collectiveExclusions	ds 18 7	X.501 Annex B

Standard Operational Attribute	OID	Reference
subschemaTimestamp	ds 18 8	X.501 (1997) Annex B
hasSubordinates	ds 18 9	X.501 (1997) Annex B
subschemaSubentry	ds 18 10	X.501 (1997) Annex B
accessControlSubentries	ds 18 11	X.501 (1997) Annex B
collectiveAttributeSubentries	ds 18 12	X.501 (1997) Annex B
Schema administration		
dITStructureRules	ds 21 1	X.501 Annex C
dITContentRules	ds 21 2	X.501 Annex C
matchingRules	ds 21 4	X.501 Annex C
attributeTypes	ds 21 5	X.501 Annex C
objectClasses	ds 21 6	X.501 Annex C
nameForms	ds 21 7	X.501 Annex C
matchingRuleUse	ds 21 8	X.501 Annex C
structuralObjectClass	ds 21 9	X.501 Annex C
governingStructureRule	ds 21 10	X.501 Annex C
definitions	xed 21 0	draft-legg-xed-schema-xx.txt
Basic access control		
accessControlScheme	ds 24 1	X.501 Annex D
prescriptiveACI	ds 24 4	X.501 Annex D
entryACI	ds 24 5	X.501 Annex D
subentryACI	ds 24 6	X.501 Annex D
userGroups	ads 18 5	Chapter 6
LDAP password policy		
pwdAttribute	sun-at 1	Chapter 6
pwdMinAge	sun-at 2	Chapter 6
pwdMaxAge	sun-at 3	Chapter 6
pwdInHistory	sun-at 4	Chapter 6
pwdCheckSyntax	sun-at 5	Chapter 6
pwdMinLength	sun-at 6	Chapter 6
pwdExpireWarning	sun-at 7	Chapter 6
pwdGraceLoginLimit	sun-at 8	Chapter 6
pwdLockout	sun-at 9	Chapter 6
pwdLockoutDuration	sun-at 10	Chapter 6
pwdMaxFailure	sun-at 11	Chapter 6
pwdFailureCountInterval	sun-at 12	Chapter 6
pwdMustChange	sun-at 13	Chapter 6
pwdAllowUserChange	sun-at 14	Chapter 6
pwdSafeModify	sun-at 15	Chapter 6
pwdChangedTime	sun-at 16	Chapter 6
pwdAccountLockedTime	sun-at 17	Chapter 6
pwdExpirationWarned	sun-at 18	Chapter 6
pwdFailureTime	sun-at 19	Chapter 6
pwdHistory	sun-at 20	Chapter 6
pwdGraceUseTime	sun-at 21	Chapter 6

Standard Operational Attribute	OID	Reference
pwdReset	sun-at 22	Chapter 6
pwdPolicySubentry	sun-at 23	Chapter 6
pwdMaxIdle	sun-at 26	Chapter 6
pwdLastSuccess	sun-at 29	Chapter 6
DSA operational atts		
dseType	ds 12 0	X.501 Annex D
myAccessPoint	ds 12 1	X.501 Annex D
superiorKnowledge	ds 12 2	X.501 Annex D
specificKnowledge	ds 12 3	X.501 Annex D
nonSpecificKnowledge	ds 12 4	X.501 Annex D
supplierKnowledge	ds 12 5	X.501 Annex D
consumerKnowledge *	ds 12 6	X.501 Annex D
secondaryShadows *	ds 12 7	X.501 Annex D

* Standard operational attributes marked with a * are supported only to the extent that ViewDS can accept an attempt to add, modify, or read a value of the attribute; ViewDS does not support the semantics defined in the X.500 Recommendations.

ViewDS-specific operational attributes

The operational attributes implemented by the ViewDS DSA are as follows:

ViewDS-Specific Operational Attribute	OID	Reference
DSA attributes		
attributeTypeExtensions	vf 21 0	Chapter 5
dsaCollaborators	vf 12 0	Chapter 6
anonymousPrivilege	vf 12 1	Chapter 6
sepType	vf 12 2	Chapter 7
supplierStatus	vf 12 6	Chapter 7
consumerStatus	vf 12 7	Chapter 7
resolvedDistinguishedName	vf 18 0	Technical Reference Guide: User Interfaces
updatersName	vf 18 1	Technical Reference Guide: User Interfaces
userName	vf 18 2	Chapter 6
protectedFEALPassword	vf 18 3	Chapter 6
localSubschemaTimestamp	vf 18 11	Chapter 5
wordList	vf 18 13	Chapter 5
remoteAlias	vf 18 14	Chapter 7
passwordModifyTimestamp	vf 18 15	Chapter 6
passwordExpiry	vf 18 16	Chapter 6
passwordEncryption	vf 18 17	Chapter 6
privilege	vf 24 0	Chapter 6
numberOfMasterEntries	ads 12 0	Chapter 5
numberOfShadowEntries	ads 12 1	Chapter 4
proxyAgent	ads 24 0	Chapter 6
hierarchyNameSpecification	ads 18 2	Technical Reference Guide: User Interfaces

ViewDS-Specific Operational Attribute	OID	Reference
currentDateTime	vds 18 2	Chapter 4
currentTimeOfDay	vds 18 3	Chapter 4
currentDayOfWeek	vds 18 3	Chapter 4
userDateTime	vds 18 6	Chapter 4
userTimeOfDay	vds 18 7	Chapter 4
userDayOfWeek	vds 18 8	Chapter 4
userTimeZone	vds 18 5	Chapter 4
viewDSMatchQuality	vds 18 11	Chapter 4
viewDSSimpleMatchQuality	vds 18 12	Chapter 4
viewDSPasswordQuality	vds 18 21	Chapter 6
viewDSPasswordDistance	vds 18 22	Chapter 6
automaticIndexing	ads 21 2	Chapter 5
viewdsBuiltInSyntaxes	vds 21 6	Below
viewdsBuiltInSchema	vds 21 7	Below
viewdsBuiltInAttributeTypes	vds 21 12	Below
DUA configuration atts		
duaBanners	vf 18 4	Technical Reference Guide: User Interfaces
attributePresentation	vf 18 5	As above
objectClassPresentation	vf 18 6	As above
searchOptions	vf 18 7	As above
userEntitlement	vf 18 8	As above
userConfig	vf 18 9	As above
defaultEntitlement	vf 18 12	As above

viewdsBuiltInSyntaxes

An operational attribute that can be read from any entry to return a list of syntaxes supported by this instance of ViewDS. Intended for internal ViewDS use only.

viewdsBuiltInSchema

An operational attribute that can be read from any entry to return a list of built-in schema definitions supported by this instance of ViewDS. The schema definitions returned include all of the built in syntaxes, matching rules, attribute types, object classes and name forms. Intended for internal ViewDS use only.

viewdsBuiltInAttributeTypes

An operational attribute that can be read from any entry to return a list of built in attribute type definitions supported by this instance of ViewDS. This attribute uses the standard AttributeTypeDefinition syntax, whose values are the schema definitions of all built in attribute types. Intended for internal ViewDS use only.

Object classes

The ViewDS DSA has built-in definitions of the following object classes.

Supported Object Classes	OID	Reference
Directory object classes		
top	ds 6 0	X.521 Annex A
alias	ds 6 1	X.521 Annex A
country	ds 6 2	X.521 Annex A
locality	ds 6 3	X.521 Annex A
organization	ds 6 4	X.521 Annex A
organizatonalUnit	ds 6 5	X.521 Annex A
person	ds 6 6	X.521 Annex A
organizationalPerson	ds 6 7	X.521 Annex A
organizationalRole	ds 6 8	X.521 Annex A
groupOfNames	ds 6 9	X.521 Annex A
residentialPerson	ds 6 10	X.521 Annex A
applicationProcess	ds 6 11	X.521 Annex A
applicationEntity	ds 6 12	X.521 Annex A
dSA	ds 6 13	X.521 Annex A
device	ds 6 14	X.521 Annex A
strongAuthenticationUser	ds 6 15	X.521 Annex A
certificationAuthority	ds 6 16	X.521 Annex A
groupOfUniqueNames	ds 6 17	X.521 Annex A
subentry	ds 17 0	X.501 Annex B
accessControlSubentry	ds 17 1	X.501 Annex B
collectiveAttributeSubentry	ds 17 2	X.501 Annex B
subschema	ds 20 1	X.501 Annex C
X.400 object classes		
mhs-distribution-list	mhs 5 1 0	X.402
mhs-message-store	mhs 5 1 1	X.402
mhs-message-transfer-agent	mhs 5 1 2	X.402
mhs-user	mhs 5 1 3	X.402
mhs-user-agent	mhs 5 1 4	X.402

Name forms

The ViewDS DSA has built-in definitions of the following name forms. Note that it supports RDNs containing multiple AVAs.

Standard Name Forms	OID	References
countryNameForm	ds 15 0	X.521 Annex A
locNameForm	ds 15 1	X.521 Annex A
sOPNameForm	ds 15 2	X.521 Annex A
orgNameForm	ds 15 3	X.521 Annex A
orgUnitNameForm	ds 15 4	X.521 Annex A
personNameForm	ds 15 5	X.521 Annex A
orgPersonNameForm	ds 15 6	X.521 Annex A

orgRoleNameForm	ds 15 7	X.521 Annex A
gONNameForm	ds 15 8	X.521 Annex A
resPersonNameForm	ds 15 9	X.521 Annex A
applProcessNameForm	ds 15 10	X.521 Annex A
applEntityNameForm	ds 15 11	X.521 Annex A
dSANameForm	ds 15 12	X.521 Annex A
deviceNameForm	ds 15 13	X.521 Annex A

Appendix C OpenSSL and SSLeay licensing

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

OpenSSL License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgment: “This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (http://www.openssl.org/)”
4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org.
5. Products derived from this software may not be called “OpenSSL” nor may “OpenSSL” appear in their names without prior written permission of the OpenSSL Project.
6. Redistributions of any form whatsoever must retain the following acknowledgment:

   “This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/)”

PROJECT OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Original SSLeay License

Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved.

This package is an SSL implementation written by Eric Young (eay@cryptsoft.com).

The implementation was written so as to conform with Netscapes SSL.

This library is free for commercial and non-commercial use as long as the following conditions are adhered to. The following conditions apply to all code found in this distribution, be it the RC4, RSA, lhash, DES, etc., code; not just the SSL code. The SSL documentation included with this distribution is covered by the same copyright terms except that the holder is Tim Hudson (tjh@cryptsoft.com).

Copyright remains Eric Young's, and as such any Copyright notices in the code are not to be removed.

If this package is used in a product, Eric Young should be given attribution as the author of the parts of the library used.

This can be in the form of a textual message at program start-up or in documentation (online or textual) provided with the package.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgement: "This product includes cryptographic software written by Eric Young (eay@cryptsoft.com)"

   The word 'cryptographic' can be left out if the routines from the library being used are not cryptographic related.

4. If you include any Windows specific code (or a derivative thereof) from the apps directory (application code) you must include an acknowledgement:

   “This product includes software written by Tim Hudson (tjh@cryptsoft.com)”

THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ”AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

The licence and distribution terms for any publicly available version or derivative of this code cannot be changed. That is, this code cannot simply be copied and put under another distribution licence (including the GNU Public Licence).

Appendix D Open XML SDK licensing

The MIT License (MIT)

Copyright (c) Microsoft Corporation

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,

FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.