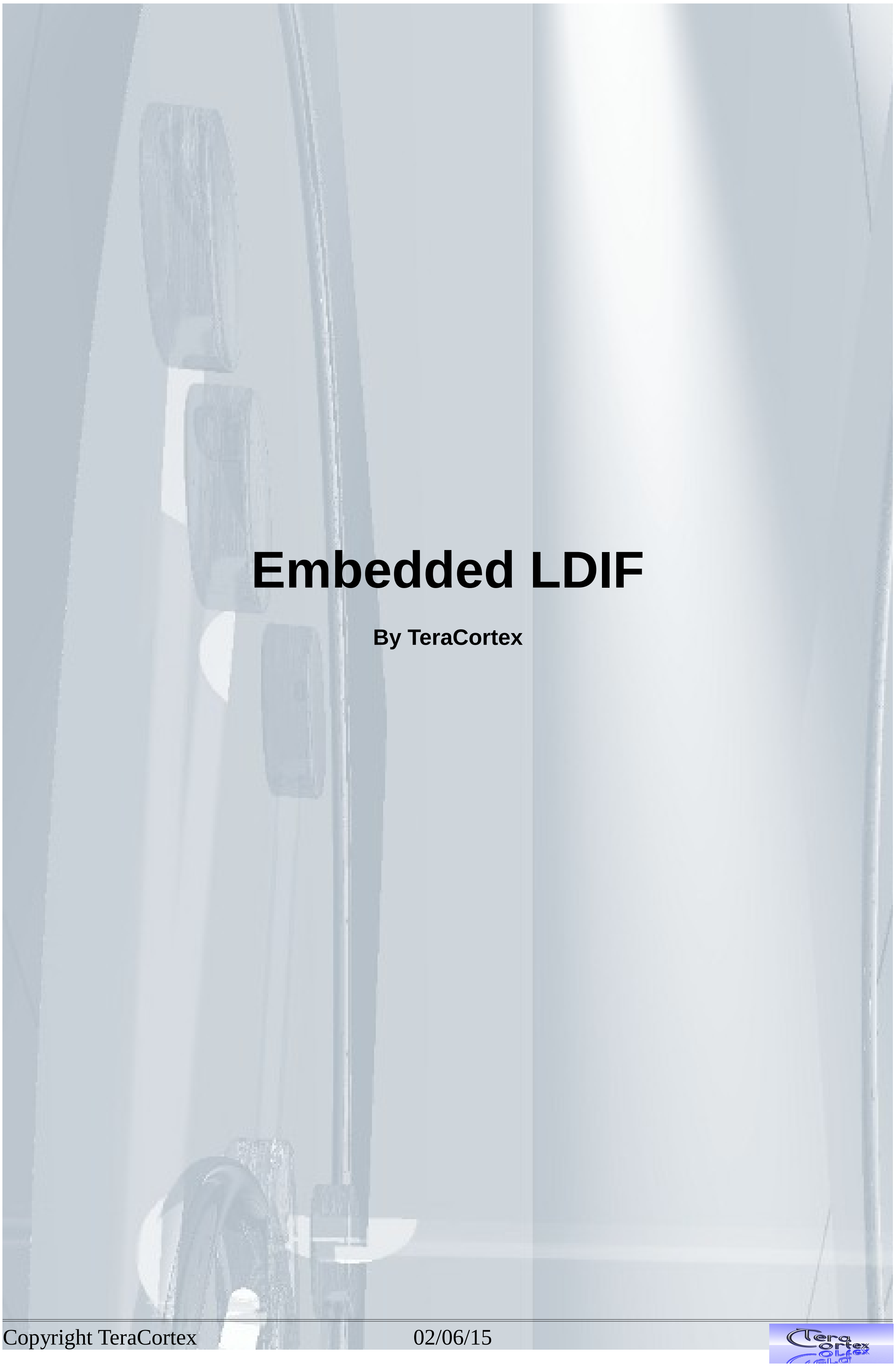

Embedded LDIF

By TeraCortex

Table of Contents

Independent Submission
INTERNET-DRAFT
Intended Status: Proposed Standard
Expires 2015/08/26
draft-hollstein-embedded-ldif-03.txt

Christian Hollstein
TeraCortex
February 2015

The Embedded LDAP Data Interchange Format (EMLDIF)

Status of This Memo

Abstract

RFC2849 and EXLDIF specify how LDAP operations can be represented in a text file. Client implementations may use this data to send sequences of requests to a LDAP server. This document specifies how values from the server's response can be propagated, displayed and used for decision taking in procedural logic. It enables EXLDIF with algorithmic behavior. The general method is to use EXLDIF embedded in a high level programming language like C/C++, Java and others.

Copyright Notice

1. Overview

This document specfies how EXLDIF can be embedded and used in a high level programming language. Embedding adds the following capabilities to EXLDIF:

- Making use of all capabilities of the embedding language like object oriented or structured programming

- Decision taking based on the result codes or data content of server responses to previous requests

- Full support for LDAP transactions [RFC5805]

- Execution of EXLDIF requests inside of loops, branches

- Execution of EXLDIF requests inside of methods

- Dynamic replacement of attribute values and / or distinguished names by internal variables of the program or external environment variables

- Multiple connections per thread of execution

- Multiple threads per Embedded LDIF source file

- Request delay

- Asynchronous mode in conjunction with [QLENCONTROL]

The chapters 2 through 6 give a formal specification of the language elements. For comprehensive examples in the language C please refer to [EMLDIF-C]

1.1. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

Today's high level languages are either object oriented or structured. A common property is the ability to organize recurring functionality into some sort of callable entity. In object oriented languages these entities a called "methods". In structured language they are known as "functions", "procedures" or "subroutines". This specification uses the term "method" to refer to a callable entity regardless of particular programming languages.

1.2. General Implementation Design

The general method how to process an Embedded EXLDIF source code file is assumed to be independant from the the specific programming language used to embed EXLDIF. Implementations may vary but the functionalities below are expected a common appearance:

a) Replace any occurence of external environment variables by their real values.

b) Parse the source file and translate all EXLDIF records into an internal format suitable for LDAP protocol level encoding.

c) Generate an intermediate file with all EXLDIF code replaced by calls to methods that can be executed from within the chosen programming language. Give the calls appropriate arguments with data content coming from the internal representation of the parsed EXLDIF records.

From this point two different types of execution profile are possible.

1.2.1. Dynamically Linked Library (Type I)

Implementations SHOULD take this approach if the choosen programming language supports dynamic linking. The language specific compiler is called to translate the parsed and converted intermediate file into a dynamically attachable library. Then the library is linked with the running process and executed. With this design all steps are performed in a single sequence. This is the preferred method.

1.2.2. Binary Executable (Type II)

Implementations MUST take this approach if the choosen programming language does not support dynamic linking. The language specific compiler is called to translate the parsed and converted intermediate file into a binary executable. The generated binary is called right away or executed later independently. This method requires that the parser functionality (b) of chapter 1.2. is called once again in the generated binary. Otherwise the data structures representing the EXLDIF record content are not available.

2. Basic Definitions

This chapter specifies the basic structure of Embedded LDIF. Implementations MUST report any violations of the rules below as syntax error. If there are syntax errors, LDAP requests MUST NOT be sent to the server.

2.1. The EXLDIF Record

Any EXLDIF record starts with the keyword "dn:", possibly followed by the distinguished name value on the same line. Below this line a sequence of EXLDIF directives and / or EXLDIF comments appears as specified in [RFC2849] and [EXLDIF]. The first empty line below this sequence terminates the EXLDIF record. It is an integral part of the EXLDIF record syntax. An empty line is a line that solely consists of white space characters.

Each record inside an Embedded LDIF source code file has an implicit numeric identifier. Record identifiers are counted individually in each method top to bottom making records relative to the method they are located in. The upmost record in the method has the ID 0, the next one 1 and so on, making identifers an implicit property based on the record position in the method. There is no syntactic element to a assign an identifier explicitely to a particular record. Record identifiers are not affected by changes of declarations, statements, branches, or loops in the embedding program. They are affected by insertions or deletions of other records inside the same method.

2.2. EXLDIF Indentation

Embedded LDIF MAY be indented by a number of space characters (ASCII 0x20). Tabs are not allowed. Any directive inside a EXLDIF record MUST be indented by the same amount of spaces. Continued lines are indented by one more space, means: If "N" is the number of spaces from the begin of the line up to the "dn:" keyword, any directive or comment line inside this EXLDIF record MUST be indented by "N" spaces as well. Continued directive lines or comment lines inside this record MUST be indented by "N+1".

2.3. EXLDIF Comments

EXLDIF records may contain comment lines. A comment line begins with a hash (ASCII 0x23, '#'). Comment lines MUST follow the same indentation rules as non - comment lines of the EXLDIF record. Outside of LDIF records comments MUST be in the style of the embedding program.

2.4. Embedding Program Comments

Comments in the style of the embedding program MUST NOT be placed inside the EXLDIF record. This applies also to the terminating empty line of a EXLDIF record.

2.5. The EXLDIF Record Set

A EXLDIF record set is the maximum sequence of LDIF records separated from each other by one or more empty lines or comments in the style of the embedding program. It starts with the first (maybe indented) EXLDIF record below a code line or comment of the embedding program. The EXLDIF record set ends when a non - empty line follows an empty line and the non - empty line is not a dn-spec.

2.6. The Program Block

High level programming languages have the concept of structuring sequences of declarations and statements into "blocks". Blocks can be nested. Normally there are syntactic elements to start and terminate blocks, e.g. curly brackets (C, C++, Java) or keywords (BEGIN, END in Algol, Pascal) or indentation is used to express a block structure. The embedded program MUST support the concept of such blocks. A particular EXLDIF record set MUST completely be contained inside a block of the program. Block nesting is of course allowed.

2.7. Conflicting Syntax Elements

It cannot be ruled out that programming languages offer syntax elements that conflict with Embedded LDIF syntax. Of particular interest is the dn-spec production:

dn: ...

It is the keyword from which an Embedded LDIF parser can recognize the start of a EXLDIF record. There is a conflict if the embedding program syntax allows for such a (possibly indented) keyword at the begin of a line being part of the program code. In these cases the particular language SHOULD be avoided for use with Embedded LDIF. In any case programmers MUST NOT use syntax elements of the programming language at positions that lead to conflicts with Embedded LDIF syntax.

3. Embedded LDIF Operations

3.1. Relation to existing documents

This specification relies on [RFC2849], [EXLDIF] and [QLENCONTROL].

[EXLDIF] contains the following definition of a change record:

```
changerecord = "changetype:" FILL
               (change-add / change-delete /
               change-modify / change-moddn /
               operation-bind / operation-unbind /
               operation-compare / operation-search /
               operation-extended / operation-abandon)
```

This is extended as follows:

```
changerecord = "changetype:" FILL
                (change-add / change-delete /
                change-modify / change-moddn /
                operation-bind / operation-unbind /
                operation-compare / operation-search /
                operation-extended / operation-abandon /
                operation-conncet / operation-disconnect /
                operation-response)
```

The ABNF forms below make use of ABNF definitions already presented in [RFC2849] and [RFC3986]. Of particular interest are:

- hostport	The hostport from Section 5 of [RFC3986]

- DIGIT	DIGIT definition from [RFC2849]
- FILL	FILL definition from [RFC2849]
- SEP	SEP definition from [RFC2849]

Either of the additional operations is covered in the following chapters. For the sake of simplicity the keywords "dn" and "changetype" are used for the additional operations despite the fact that they have nothing to do with LDAP. Their semantics are on TCP level.

3.2. CONNECT

```
CONNECT         = "dn:" SEP operation-connect

operation-connect  = "connect"  SEP
                   "connection:" FILL 1*restricted-url SEP

restricted-url     = scheme "://" ([hostport] / filename)

scheme             = ("ldap" / "ldaps" / "file")
```

The connection information is used to establish a TCP connection to the server. If the port part of hostport is ommitted, implementations SHOULD examine the scheme. If the scheme is "ldap" they SHOULD try to connection to the port 389. If the scheme is "ldaps", they SHOULD try to connect to the port 636. If the scheme is "file" the value "filename" refers to a file in the machine's file system. In this case the LDAP messages are dumped BER encoded to the given file. Multi threaded implementations MUST either add automatically a numeric suffix to the file name according to the number of the thread that emits the output stream or synchronize concurrent access of different threads to the same output file.

3.3. DISCONNECT

```
DISCONNECT       = "dn:" SEP "disconnect" SEP
```

3.4. RESPONSE

```
RESPONSE         = "dn:" SEP operation-response

operation-response = "response" SEP
                     "responses:" FILL 1*DIGIT SEP
```

The value for the "responses" keyword is an integer. It MUST give the number of responses the client expects to receive from the server.

3.5. Result References

A reference is a means to refer to results of requests already sent to the server. In time it points always into the past. In terms of its position in the Embedded LDIF source code file it may point to previous or subsequent records or even to a record in a different file. The server MUST already have responded to the request and the client MUST have processed the response before a reference can have any effect. There are three types of references. The reference syntax is as follows:

```
REFERENCE = FILE ":" CLASS ":" METHOD ":" RECORD ":" THREAD ":"
            INSTANCE
```

FILE The name of one of the Embedded LDIF source code input files that are currently processed. If FILE is empty, the current input file is meant.

CLASS The name of a class definition in the referred input FILE. This is only relevant for object - oriented embedding programs that have intrinsic support for object classes. In structured programming languages CLASS is empty.

METHOD The name of a method in the referenced CLASS if CLASS is used. In structured programming languages METHOD refers to a function, procedure or subroutine inside the referred FILE.

RECORD 1*DIGIT

This is the identifier of the referenced record.

THREAD 1*DIGIT [-1 ... N]

This is the identifier of a thread in the in client execution runtime, if the implementation supports multiple parallel threads. Otherwise it is always zero. If negative it refers to the own thread.

INSTANCE 1*DIGIT [0 ... N]

This identifies a particular executed instance of the referenced record. Records may be executed inside loops where each invocation yields a different result. INSTANCE is a means to access such particular results.

3.5.1. Connection Identifier

connection-identifier = "connectionId(" REFERENCE ")"

This reference MUST point to a record of type "connect". It makes the referenced connection available to the referencing record. Implementations MUST send the referencing LDAP request to the given connection and receive the response(s) from there. The syntax of records using connection-identifier is specified in chapter 4.

3.5.2. Transaction Identifier

```
transaction-identifier = "transactionId(" REFERENCE ")"
```

This reference MUST point to a record of type extended. The record must contain an extended request suitable to invoke a "transaction begin" operation. This reference MUST NOT be used except as value of a transaction control. For LDAP transactions please refer to [RFC5805].

The syntax for controls using this reference is:

```
"control:" FILL trans-oid 1*SPACE "true:" connection-identifier

trans-oid = ( "1.3.6.1.1.21.2" / propriatary-oid )

propriatary-oid   Object identifier used in propriatary
                  implementations of transaction semantics.
```

3.5.3. Message Identifier

```
message-identifier     = "messageId(" REFERENCE ")"
```

This reference MUST point to a record representing a type of request that can be abandoned. It MUST NOT be used except as value of the "messageId:" directive of an abandon record. Please refer to [RFC4511] for a list of request types that can be abandoned.

The syntax for the "messageId:" directive of abandon requests using this reference is:

```
"messageId:" FILL message-identifier
```

4. Changetype Line Parameters

The behavior of EXLDIF records can be modified by a small set of parameters. These parameters are concatenated at the "changetype" lines:

```
changeline = "changetype:" FILL
                changetype
                0*1(1*SPACE DELAY)
                0*1(1*SPACE CONNID)
                0*1(1*SPACE RESULT) SEP

changetype = ("add"      / "delete"  / "modify"  / "moddn"      /
              "bind"     / "unbind"  / "compare" / "search"     /
              "extended" / "abandon" / "conncet" / "disconnect" /
              "response"
              )
```

4.1. DELAY

DELAY = 1*DIGIT

This parameter is an integer giving the request delay in milli seconds. Implementations MUST apply this delay before the request encoded in this EXLDIF record is sent to the LDAP server.

4.2. CONNID

CONNID = connection-identifier

4.3. RESULT

```
RESULT = "result(" result-variable ":" result-count ":"
         result-objects ")"
```

result-variable	The name of a a local variable that will contain a list of results. The syntax of local variable names is specific to the embedding program language.

```
result-count    = 1*DIGIT
```

This is an integer giving the maximum number of record invocations for which a result will be stored.

```
result-objects   = 1*DIGIT
```

This is an integer giving the maximum number of objects that will be stored as response to a search request per invocation of the record.

Implementation MUST store the results up to the limits of result count and object count (search requests only) in result-variable. From there results can be accessed by the embedding program. Typically result-variable will be some sort of array, linked list or other appropriate arrangement of structured data. The precise data model layout is specific to the embedding language.

5. Value Replacement

There are two types of value replacement: A static one and a dynamic one. Both of them use the variable name production:

```
var-name =  1*( ALPHA / DIGIT / "-" / "_" )
```

5.1. *Static Value Replacement by Environment Variables*

An environment variable takes the following form

```
envvar   = ( "${" var-name "}" / "$" var-name )
```

The first form SHOULD be used when the first character after the variable name matches the var-name character set. The second form MAY be used when this condition is false. Environment variables SHOULD have been exported on operating system level in order to take effect. Implmentations MUST try to resolve the environment variable and replace each such occurence in the source code file by the real value. Failure to do so SHOULD be reported as an error.

From the point of LDAP request processing these variables are static. Their value does not change until the program terminates.

5.2. *Dynamic Value Replacement by Local Variables*

Local variables can be used to alter the behavior of EXLDIF records. EXLDIF attribute values are replaced with the values of variables of the embedding program. Replacement takes place dynamically while LDAP requests are processed. The mechanism is based on a variable representation specified in [POSIX] for the "printf" library element. Embedded LDIF uses a subset of the "printf" functionality:

```
dynvar = "%" *DIGIT ( "d" / "s" / "f" ) "_" var-name "%"
```

With this syntax variables values can be represented as integer numbers, strings or floating point numbers either in fixed length or variable length format. Programmers SHOULD use the conversion appropriate for the data type of the variable.

6. Asynchronous Mode

Embedded LDIF support asynchronous mode by means of [QLENCONTROL] and the RESPONSE record type. [QLENCONTROL] gives a number of requests to be sent in a row without waiting for an individual response. Responses MUST be awaited for after the last request in the asynchronous queue has been sent. For this purpose the RESPONSE record must be used. The numeric value for the "responses:" directive MUST match the number of requests in the asynchronous queue as given in [QLENCONTROL].

7. Security Considerations

In addition to the security issues of LDIF files [RFC2849] Extended LDIF may contain authentication information used for BIND operations. This sensitive data MUST NOT be displayed to unauthorized people.

In Embedded LDIF it is pretty easy to create a program firing millions of requests to a LDAP server in short time frame. Such denial of service attacks are illegal. Their prevention is not in scope of this specification.

General security considerations [RFC4510], especially those associated with update operations [RFC4511], apply to this extension.

8. IANA Considerations

There are no new object identifiers associated with this specification.

9. Acknowledgments

The author gratefully acknowledges the contributions made by Internet Engineering Task Force participants.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997.

[RFC2849] Good, G., "The LDAP Data Interchange Format (LDIF) - Technical Specification", RFC 2849, June 2000.

[RFC3986] Berners-Lee, T., Ed., Masinter, L., Ed., "Uniform Resource Identifier (URI): Generic Syntax", RFC 3986, January 2005

[RFC4510] Zeilenga, K., Ed., "Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map", RFC 4510, June 2006.

[RFC4511] Sermersheim, J., Ed., "Lightweight Directory Access Protocol (LDAP): The Protocol", RFC 4511, June 2006.

[POSIX] IEEE Std 1003.1, 2003 Edition

10.2. Informative References

[RFC5805] Zeilenga, K., "Lightweight Directory Access Protocol (LDAP) Transactions", RFC 5805, March 2010.

[EXLDIF] Hollstein, C., "The Extended LDAP Data Interchange Format (EXLDIF)", draft-hollstein-extended-ldif-03.txt, February 2015.

[EMLDIF-C] Hollstein, C., "The Embedded LDAP Data Interchange Format for C (EMLDIF-C)", draft-hollstein-embedded-ldif-c-03.txt, February 2015.

[QLENCONTROL] Hollstein, C., "LDAP Queue Length Control", draft-hollstein-queuelength-control-03.txt, February 2015

Author's address

Christian Hollstein
TeraCortex
Hopfenbrede 2
D-49179 Ostercappeln

E-Mail: chollstein@teracortex.com
Phone: 0049 / 5473 / 9933
Mobile: 0049 / 160 / 96220958

Appendix A. Changes

Added in chapter 3.5 (Result References) the option "-1" to refer to resources of the own thread.
Changed for dynamic value replacement (chapter 5.2) the separator between the format specifier and the variable name from ":" to "_". This is necessary to avoid conflicts in dynamic value replacements of extensible search filters which might contain colons (":") anyway.