X/Open CAE Specification

Federated Naming:
The XFN Specification

X/Open Company Ltd.

X/Open CAE Specification

Federated Naming: The XFN Specification

ISBN: 1-85912-052-0

X/Open Document Number: C403

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

List of Figures

List of Tables

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to info-server@xopen.co.uk with the following in the Subject line:

```
request corrigenda; topic index
```

This will return the index of publications for which Corrigenda exist.

This Document

This document is a **CAE Specification**, except for those sections which are explicitly marked as at **Preliminary Specification** status. To enable clear identification of these *preliminary-status* sections, their titles, or page headers in the case of man-page definitions, include the word **Preliminary**. See also **Typographical Conventions** at the end of this Preface.

The **Federated Naming** specification defines programmatic interfaces for a federated naming service, and specifies the naming policies to be used in conjunction with a federated naming service. Collectively, these are called the XFN specification.

Intended Audience

This document is intended to serve two main classes of readers:

- software developers of applications that use naming services
- application and system software developers who wish to provide the XFN client interface or add support to federate a new naming system under the XFN interface.

Structure

This document is structured as follows:

- **Chapter 1** introduces the subject, and describes conformance criteria for compliant implementations
- **Chapter 2** defines the terminology used in this specification and describes the XFN model
- **Chapter 3** gives an overview of the XFN client interface, its organization, operations and their semantics
- **Chapter 4** describes XFN composite names, their syntax and resolution techniques
- **Chapter 5** presents the XFN policies
- **Chapter 6** gives reference manual pages for each routine in the XFN interface
- **Appendix A** describes the XFN protocols on two platforms: ONC+ and DCE

- **Appendix B** describes how XFN is mapped to a number of popular naming services: X.500, DNS, ONC / NIS+, and DCE / CDS.
- **Appendix C** describes guidelines and techniques for federating new and existing naming systems
- **Appendix D** describes policies for the enterprise namespace
- **Appendix E** describes integrating the file service within an XFN federation
- **Appendix F** describes techniques for extending the XFN interface
- **Appendix G** is a registry for identifiers and types used by XFN implementations
- **Appendix H** gives C header descriptions for the XFN interface.

Revision History

The revisions to XFN from XFN Preliminary Specification (July 1994) are summarised below.

- Add operations for attribute-based searching.
  Define basic search and extended search operations (and supporting interfaces). See Section 3.3 on page 27 for details.
- Add operations for object creation with attributes.
  Define two new operations to allow attributes to be associated with an object at the time the object is created. See Section 3.3 on page 27 for details.
- Add XFN link argument to operations in base attribute interface.
  Add a parameter to the relevant operations in the base attribute interface that indicates whether an XFN link which is bound to the terminal atomic name should be followed. See Section 3.3 on page 27 for details.
- Streamline Enterprise Policy (XFN-EP).
  XFN-EP was simplified to allow easier mapping to existing naming systems. Many of the descriptions were clarified as well.
  - Remove *site*, a concept less prevalent than users, hosts, and organizational units in most existing naming systems.
  - Make cleaner delineation between XFN-EP and examples of XFN-EP (to make clearer the normative parts of the policy).
  - Simplify bindings in the Initial Context to make them more flexible and easier to map to concepts in existing naming systems.
  - Move **Integrating File Service** section to separate appendix.
- Add *authoritative* argument to *fn_ctx_handle_from_ref*( ) and *fn_ctx_handle_from_initial*( )
  Add an argument to *fn_ctx_handle_from_ref*( ) and *fn_ctx_handle_from_initial*( ) that enables the client application to select the authoritativeness of the context with respect to information that the context returns from the naming service.
- Add operation for obtaining equivalent forms of names.
  See description of *fn_ctx_equivalent_name*( ) in Chapter 3 for details.
- Modify **Mapping XFN to X.500** description.
  In the Preliminary Specification, the Appendix on **Mapping XFN** lacked detailed definitions for the X.500 object classes and attributes needed to support XFN. In addition, operational aspects of XFN support in X.500 were absent. It is necessary to publish these definitions and describe these operations in the XFN Specifications to ensure the ability of XFN to

interoperate with any X.500 directory service. The modifications to this section include:

— **Overview** is renamed to **X.500 Overview** and reworded

— **Representation of XFN Concepts in X.500** is expanded

— **Name Syntax** is reworded

— **XFN References** is expanded to describe the X.500 object classes and attributes necessary to support XFN object references

— **String Encoding for XFN References** is added

— **Federating X.500 with Other Naming Systems** is expanded

— **Weak Separation** is reworded

— **Implicit Next Naming System Pointers** is expanded to describe the X.500 object classes and attributes necessary to support XFN next naming system references

— **Resolving Through X.500** is reworded

— **XFN API Function Mapping** is expanded.

- Add names to reach global roots in Initial Context.
  Add the names `_dns` and `_x500` to the Initial Context so that one can name the roots of these respective namespaces.
- Drop status argument from destructors.
  Drop the **FN_status_t** parameter from the iterator destructors, to be consistent with other destructors in the interface. See *fn_ctx_list_names*( ) on page 152, *fn_ctx_list_bindings*( ) on page 151, *fn_attr_get_values*( ) on page 127, *fn_attr_multi_get*( ) on page 131.
- Accept NULL pointer in destructors.
  Specify that all destructors can accept NULL pointers as argument. See manual pages for data types with destructors.
- Accept NULL pointer as status argument.
  For those applications that do not care about the reason behind an operation failure, it can pass in a NULL pointer for *status* argument in the context and attribute operations, and a NULL pointer for *status* argument in the string-related operations.
- Add FN_E_ATTR_IN_USE status code.
- Add description of *fn_status_description*( ) in **FN_status_t** reference manual page.
- Add *fn_composite_name_from_str*( ).
  A common operation that applications perform is to convert a C-string to an **FN_composite_name_t** structure. This provides a single operation for achieving this instead of using multiple operations.
- Clarify compound name syntax model:

  — state that escapes and quotes are optional attributes

  — make pair up of multiple begin and end quotes possible by changing multivalued quotes attribute into two single-valued ones

  — add rule that escaping a non-meta character returns the non-meta character itself.

  See Section 3.8 on page 50 and reference manual page for XFN_compound_syntax.
- Specify semantics of *fn_ctx_rename*( ) with respect to attributes.
  Any attributes associated with the old name becomes associated with the new name. See

reference manual page for *fn_ctx_rename*( ).

- Specify semantics of *fn_ctx_bind*( ) with respect to attributes.
  In naming systems that support attributes and store the attributes along with the names, when binding a reference in non-exclusive mode, any attributes associated with the former binding are removed. See reference manual page for *fn_ctx_bind*( ).
- Separate **Integrating File Services** from the **Policies for the Enterprise Namespace** appendix:
  — make **Integrating File Services** into a separate appendix
  — delete reference to `/...` to prevent conflict with DCE.
- Drop FN_ID_ISO_OID_BER from list of **FN_identifier_t** formats.
  FN_ID_ISO_OID_BER is extraneous because of the presence of FN_ID_ISO_OID_STRING.
- Modify semantics of *fn_ctx_lookup_link*( ).
  Instead of failing when the name is not bound to an XFN link, *fn_ctx_lookup_link*( ) returns the reference bound to the given name. See Section 3.2.4.10 on page 24 and *fn_ctx_lookup_link*( ) on page 156.
- Correct error return of *fn_ctx_unbind*( ).
  fn_ctx_unbind( ) returns [FN_E_NAME_IN_USE] instead of [FN_E_OPERATION_NOT_SUPPORTED] when name cannot be unbound.
- Clarify descriptions in the **Registry of Types, Identifiers and Code Sets** appendix.
- Add entries in the **Registry of Types, Identifiers and Code Sets** appendix.
- Change opaque buffer *locale_info*, represented in **FN_string_t** and XFN syntax attributes, to be *language/territory* pair.
  A language / territory pair of registered identifiers is used to identify locale-specific character string representations.
- Update the **XFN Protocols** appendix to reflect XFN API changes betwen the Preliminary and CAE specification.
- Change XFN ONC protocol to allow for batching.
- Correct typographical errors.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  — command operands, command option-arguments or variable names, for example, substitutable argument prototypes
  — environment variables, which are also shown in capitals
  — utility names
  — external variables, such as *errno*

  — functions; these are shown as follows: *name*( ); names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.

- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C construct.
- The notation [ABCD] is used to identify an error value ABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font. Brackets shown in this font, `[ ]`, are part of the syntax and do *not* indicate optional items. In syntax the `|` symbol is used to separate alternatives, and ellipses (`. . .`) are used to show that additional arguments are optional.
- Shading is used to highlight sections of this specification which are at **Preliminary Specification** (not CAE Specification) status. Only the first (explanatory) paragraph of the affected section is shaded, to avoid excessive use of the shading feature.

Trade Marks

UNIX® is a registered trade mark of UNIX System Laboratories Inc. in the U.S.A. and other countries.

X/Open™ and the "X" device are trade marks of X Company Ltd. in the U.K. and other countries.

Acknowledgements

This document includes text excerpted and/or derived from the Solaris FNS Guide and other documents from SunSoft, with the permission of SunSoft.

Referenced Documents

The following documents are referenced in this specification:

Internet RFC 1034
: Domain Names — Concepts and Facilities. Mockapetris, P.V. November 1987.

Internet RFC 1035
: Domain names — Implementation and Specification. Mockapetris, P.V. November 1987.

ISO 646
: ISO 646: Information Processing Systems — ISO 7-Bit Coded Character Set for Information Exchange, 1991, International Reference Version.

ISO 10646
: ISO/IBC 10646: Information Processing Systems — Universal Multi-Octet Coded Character Set (UCS), 1993.

ISO ASN.1
: ISO 8824: Information Processing Systems — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1), 1990.

ISO BER
: ISO 8825: Information Processing Systems — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1), 1990.

ISO 8859-1
: ISO 8859-1: 1987, Information Processing Systems — 8-bit Single-byte Coded Graphic Character Sets.

ISO C
: ISO/IEC 9899: 1990, Programming Languages — C (which is technically identical to ANS X3.159-1989, Programming Language C).

CCITT X.500 (1988/1993)/ISO Directory
: ISO/IEC 9594: 1988, 1993, Information Technology — Open Systems Interconnection — The Directory.

ISO Country Codes
: ISO 3166: 1988, Codes for the Representation of Names of Countries.

DCE RFC 40.1
: S. Martin O'Donnell (OSF), ''OSF Character And Code Set Registry'', June 1994, available per email request to cs_registry@osf.org

X/Open DCE RPC
: X/Open Preliminary Specification, October 1993, X/Open DCE : Remote Procedure Call (ISBN: 1-872630-95-2).

X/Open DCE Directory
: X/Open Preliminary Specification, December 1993, X/Open DCE: Directory Services (ISBN: 1-85912-012-P314).

Solaris FNS Guide
: Solaris Federated Naming Service Guide, September 1993.

Solaris NIPG
: Solaris Network Interface Programming Guide for Solaris 2.3, October 1993.

Administering NIS+

All About Administering NIS+, Ramsey, R. SunSoft Press (ISBN: 0-13-068800-2).

Chapter 1

Introduction

1.1 Motivation

A fundamental facility in any computing system is the naming service — the means by which names are associated with objects, and by which objects are found given their names. The naming service provides operations for:

- associating (binding) names to objects
- resolving names to objects
- removing bindings, listing names, renaming and so on.

In traditional systems, a naming service is seldom a separate service. It is usually integrated with another service, such as a file system, directory service, database, desktop, mail system, spreadsheet, or calendar. For example, a file system includes a naming service for files and directories; an X.500 directory service combines a naming service with an information service; and a spreadsheet has a naming service for cells and macros.

1.1.1 Uniform Naming Interface

There is a great degree of diversity and incompatibility amongst existing naming services, due in part to different requirements of scalability, performance, granularity, naming policy, and in part because naming is often embedded in specific services, such as the file system, and accessed only through that service's interface. Not only do the naming interfaces differ widely but the essential naming operations are obscured.

At present, there is no naming interface that includes the basic naming operations that any naming service can support. Applications use the interface of a specific service that provides naming functionality. This means that the application is not portable to an environment where the equivalent naming functions are provided by a different service. Even within the same environment, it is not easy to replace the underlying naming service with a different one. The resulting impediments to building well-integrated distributed systems are substantial.

A standard naming interface that supports the basic naming functionality can help overcome these problems. The standard naming interface would define a uniform interface to a large class of naming systems fitting a general model.

The use of a standard naming interface will help improve the modularity and flexibility of the distributed computing platform. It will allow the basic naming operations supported by different types of naming services to be applied uniformly without requiring changes to the client.

1.1.2 Composite Name Support

Most applications that were originally conceived and developed for single machine environments use only a single naming service. Even when these applications are evolved to work in a distributed environment, they have very limited access to objects in the network. Historically, a small percentage of applications use composite names — names that span multiple naming systems — for accessing remote objects. The UNIX commands *mount* and *rcp* are examples of such applications. Microsoft *Excel* for the Apple Macintosh is another. Each application defines its own composition rule for names, parses the composite names, and performs resolution of composite names. Composition rules differ from one application to another. For example, *rcp* in UNIX uses composite names such as **sylvan:/usr/smith/memo**, which has two components: the host name **sylvan** and the file pathname **/usr/smith/memo**. Microsoft *Excel* for the Apple Macintosh uses composite names such as **HardDisk:Finance:Budgets:Fiscal 1992:Household!B4** which has two components: the pathname of the spreadsheet **HardDisk:Finance:Budgets:Fiscal 1992:Household**, and the spreadsheet cell name, **B4**.

The user must remember which applications permit composite naming and which do not. For example, in UNIX, the composite name **sylvan:/tmp/foo** is accepted by the command *rcp*, but not by the command *cp*. The user must also remember the composition rules. Different composition rules can be found among different applications, even in the same operating system. Applications which support composite names on their own can use only a small and specific set of naming services, and must be changed whenever a new type of naming service is added.

Incorporating composite naming into the operating system permits any application to use composite names. Some operating systems have incorporated composite naming at the system level. Two examples are Sun Microsystems Automounter and the OSF DCE Naming Service. However, these approaches are limited to a fixed number of naming services and are restricted to services that integrate them.

A federated naming system is an aggregation of autonomous naming systems that cooperate to offer a standard interface for resolution for composite names. It has the following advantages over the more traditional specialised approaches to handling composite names:

- A single uniform naming interface is provided to clients.
- The addition of new types of naming services does not require changes to applications, or to existing member naming systems.
- Any number of naming services can be added, and composite names may have a large number of components.
- By enabling flexible composition of names from different naming systems, federated naming facilitates coherent naming by encouraging shared contexts and shared names.

1.1.3 Naming Policy in Federations

The computing environment of an enterprise typically consists of several naming services. There are naming services that provide contexts for naming common entities in an enterprise such as organisations, physical sites, human users and computers. Naming services are also incorporated in applications offering services such as file service, mail service, printer service, and so on. These different naming services are the result of different requirements. Yet, from a user's perspective, there exist several natural and logical relationships between these naming services. For example, it is natural to think of naming a variety of services such as mail, appointment calendar, files, and so on., in the context of a user. It is also natural to think of a user in the context of a department, within a division of an enterprise. Meaningful names can be composed using useful arrangements of naming services reflecting these relationships.

If applications are not provided with a common set of policies, naming across applications will not be logically coherent, even within a homogeneous distributed environment. In addition, without a common set of policies across distributed computing platforms, applications will have to deal with gratuitous differences in policies. This not only reduces application portability but also complicates applications that need to work when spanning more than one type of distributed computing platform.

1.2 Scope of Specifications

The primary service provided by a federated naming system is to map a *composite name* to a *reference*. A composite name is composed of name components from one or more naming systems. A reference consists of one or more communication endpoints. An additional service provided by a federated naming system is to provide access to attributes associated with named objects. This extension is to satisfy most applications' additional naming service needs without cluttering the basic naming service model.

This document contains a specification of programming interfaces for a federated naming service. Application programming interfaces are specified as well as RPC interfaces. In addition, this document specifies naming policies to be used in conjunction with a federated naming service. These specifications are called the XFN specifications.

The primary goals of XFN are:

- to provide a federated naming service interface comprising a set of common naming operations
- to provide sufficient policy infrastructure to allow applications to construct and use composite names uniformly.

XFN does not specify administrative interfaces. The administrative models of different individual naming services vary too widely to permit a useful generic treatment. These are outside the scope of XFN.

1.3 Relationship to Other Services

XFN provides a uniform interface to a basic set of naming service operations and a means to federate naming systems. The interface is intended to be implemented over a number of existing naming services, using their existing programming interfaces and protocols, as well as with new naming services in the future.

1.4 Lineage

The XFN model has been based on the federated naming model ("Ivy") originally developed by Sun Microsystems Inc. from 1989 to 1991. In 1991, Sun Microsystems provided a description of Federated Naming requirements in response to X/Open's interest in this. In May 1993, as a result of this request, the champions of Federated Naming (Sun, OSF, HP, Siemens Nixdorf, IBM and DEC) agreed to produce the XFN specifications contained herein.

The XFN model has evolved from the Ivy model in two main aspects. First, it has added a generalised attribute interface. This aspect has been influenced by the Remote Network Directory developed by Hewlett-Packard in 1989 and appeared as OSF RFC 18 in 1992. Second, the XFN model specifies a string representation for composite names which is different from that specified in Ivy.

Readers interested in the revisions to XFN from the **XFN Preliminary Specification** (July 1994) may refer to the **Revision History** in the **Preface**.

1.5 Conformance

XFN conformance has four main facets:

- XFN API
- XFN composite name string representation
- XFN naming policies
- XFN reference and address.

In addition, there are optional components:

- XFN protocols
- XFN context implementations
- XFN enterprise policies.

1.5.1 XFN API

An XFN conformant naming service must provide the following interfaces (Chapter 3):

- base context interface
- base attribute interface
- extended attribute interface
- abstract data types of objects passed across the base context, base attribute and extended attribute interfaces.

These are specified as C interfaces in the manual pages (Chapter 6). These manual pages are the definitive source of the API. Conformance to the XFN API is based on the definitions in these manual pages.

The lookup operation must be enabled. Enabling the remaining operations in the APIs is optional. However, resolution implicit in any of the other operations must be enabled.

If the naming system being provided is a non-terminal naming system in the federation, support must be provided to resolve composite names through this naming system.

The following are the optional subcomponents of the XFN API:

- compound name object (FN_compound_name_t)
- XFN Standard Syntax Model for compound names.

1.5.2 XFN Composite Name String Representation

An XFN compliant naming system must specify the string representation of names from its namespace and how these names will be composed in the string form of an XFN composite name. Composition adhere to the rules specified in Section 4.1.2 on page 56.

The minimum requirement for all XFN implementations is to support the portable representation of ISO 646 (same encoding as ASCII) for communication of name strings. All other representations are optional. See Section 2.5 on page 17.

1.5.3 XFN Naming Policies

Naming systems can exist at various levels in a computing environment, providing a range of naming scope (Section 5.1 on page 71).

For naming systems with a global scope in naming, XFN conformance means that access to either or both X.500 and Internet Domain Naming System (DNS) must be provided by supplying in the Initial Context a binding of both the atomic names ‘‘`...`’’ and ‘‘`/...`’’ to the root of either or both system (Section 5.3 on page 73). In addition, if DNS is supported, the atomic name `_dns` must be present in the Initial Context and be bound to the root of the DNS namespace. If X.500 is supported, the atomic name `_x500` must be present in the Initial Context and be bound to the root of the X.500 namespace.

It is permissible for XFN conformant naming systems to add policies in addition to those in this specification.

1.5.4 XFN Reference and Address

Identifiers for the XFN reference types and address types must be supported as described in Section 3.6.3 on page 42.

An XFN implementation that supports XFN links must use the reference type defined in Appendix G.

1.5.5 XFN Protocols (Optional)

An XFN implementation is not required to support any particular XFN protocol but if a protocol for a specific platform is supported, it must comply with the behaviour specified in Appendix A.

In any protocol implementation, the lookup operation must be enabled. Enabling the remaining operations in the protocol is optional. However, resolution implicit in any of the other operations must be enabled.

If the naming system that exports the protocol is a non-terminal naming system in the federation, support must be provided to resolve composite names through this naming system.

An XFN conformant naming system that exports an XFN protocol using DCE IDL must use the protocol specified in Section A.1. It must provide the following interfaces:

- base context interface
- base attribute interface
- context management interface
- attribute search interface.

The following interface is optional:

- context location interface.

An XFN conformant naming system that exports an XFN protocol using ONC RPCL must use the protocol specified in Section A.2.

1.5.6 XFN Context Implementations (Optional)

Implementations that provide the XFN API for the Internet Domain Name Service must conform to the specification described in Section B.1.

Implementations that provide the XFN API for the CCITT X.500 must conform to the specification described in Section B.2.

Implementations that provide the XFN API for the DCE CDS must conform to the specification described in Section B.3.

Implementations that provide the XFN API for the ONC NIS+ must conform to the specification described in Section B.4.

1.5.7 XFN Enterprise Policies (Optional)

For naming systems with an enterprise-wide scope in naming, an XFN conformant system should use the policies indicated in Appendix D. This includes providing an enterprise namespace with the structure described, and using the syntax of names described to name objects within the enterprise.

Not all policies in Appendix D are applicable in all environments. For example, some environments may have no notion of users. In an environment where only some of the policies are meaningful, the parts of the policies that are meaningful are used and those parts that are not meaningful are not used. There may be gradations of support of XFN enterprise policies among systems. The purpose of the conformance statement and questionnaire for the XFN Enterprise Policies of any particular system is to enable the application developer to decide when to use these policies or some environment-specific policies.

It is permissible for XFN conformant naming systems to add policies for naming objects within an enterprise in addition to those described in Appendix D.

1.6 Typographical Conventions

The typographical conventions used in this specification are defined at the end of the **Preface**.

In particular, this specification contains certain material which is assigned X/Open **Preliminary Specification** status (not **CAE Specification** status).

Where this applies to a section, the title of the affected section includes the words **Preliminary Specification**. Where it applies to a man-page definition, the word **PRELIMINARY** appears in the centre of each page header for that definition. In addition, shading is used in the first paragraph of each affected section to highlight the *preliminary specification* status of those sections.

Chapter 2

Model and Definitions

This chapter provides an overview of the XFN naming model and a description of the elements in the model.

2.1 Definitions

2.1.1 Names, References, Bindings, Contexts

Every name is generated by a set of syntactic rules called a *naming convention.*

An *atomic name* is an indivisible component of a name, as defined by the naming convention.

A *compound name* represents a sequence of one or more atomic names composed according to the naming convention.

For example, in UNIX pathnames, atomic names are ordered from left to right, and are delimited by slash ('/') characters. The UNIX pathname **usr/local/bin** is a compound name representing the sequence of atomic names, **usr**, **local**, and **bin**. In names from the Internet Domain Naming System (DNS) , atomic names are ordered from right to left, and are delimited by dot ('.') characters. Thus, the DNS name **sales.Wiz.COM** is a compound name representing the sequence of atomic names, **COM**, **Wiz**, **sales**.

A naming convention enables the definition of functions for parsing any compound name to produce its sequence of atomic names. Two basic name parsing functions are *first*( ), that returns the first atomic component of name, and *rest*( ), that returns the remainder of name after *first*( ) is removed. The following examples illustrate the *first*( ) and *rest*( ) functions of two different naming conventions:

$first_{UNIX}$(usr/local/bin) = usr
$rest_{UNIX}$(usr/local/bin) = local/bin
$first_{DNS}$(sales.Wiz.COM) = COM
$rest_{DNS}$(sales.Wiz.COM) = sales.Wiz

The naming convention also determines equivalence amongst names. For example, the naming convention for UNIX pathnames regards case as distinguishing, whereas the convention for Internet DNS is case insensitive. So the UNIX pathname **home/smith/memo** is not equivalent to the name **home/Smith/memo**, while the Internet DNS name **sales.Wiz.COM** is equivalent to the name **sales.wiz.com**.

The *reference* of an object contains one or more communication endpoints (*address*).

The association of an atomic name with an object reference is called a *binding.* For simplicity, an object reference and the object it refers to are sometimes used interchangeably.

A *context* is an object whose state is a set of bindings with distinct atomic names. Every context has an associated naming convention. A context provides a lookup (resolution) operation, which returns the reference bound to an object, and may provide operations such as for binding names, unbinding names, listing bound names.

An atomic name in one context object can be bound to a reference to another context object of the same type, called a *subcontext,* giving rise to compound names. In the earlier UNIX pathname example, the atomic name **local** is bound in the context of **usr** to a directory context (and subcontext) in which **bin** is bound. Resolution of compound names proceeds by looking up

each successive atomic component in each successive context. The reader will find a familiar model in UNIX file naming, where directories serve as contexts, and pathnames may be compound names.

A *naming system* is a connected set of contexts of the same type (having the same naming convention) and providing the same set of operations with identical semantics. In UNIX, for example, the set of directories in a given file system (and the naming operations on directories) constitute a naming system.

A *naming service* is the service offered by a naming system. It is accessed using its interface.

A *namespace* is the set of all names in a naming system.

2.1.2 Composite Names, Federated Naming Systems

A *composite name* is a name that spans multiple naming systems. It consists of an ordered list of zero or more *components*. Each component is a string name from the namespace of a single naming system.

A *federated naming system* is an aggregation of autonomous naming systems that *cooperate* to support name resolution of composite names through a standard interface. Each member of a federation has autonomy in its choice of operations other than name resolution.

A *federated naming service* is the service offered by a federated naming system.

A *federated namespace* is the set of all possible names generated according to the policies that govern the relationships among member naming systems and their respective namespaces.

In a federated naming system, a *naming system boundary* is the point where the namespace under the control of one member of the federation ends, and where the namespace under the control of the next member of the federation begins.

Composite name resolution is the process of resolving a name that spans multiple naming systems.

When one naming system is federated with another naming system, the naming system that is involved first during composite name resolution is called the *superior* naming system. The naming system that is involved next, after resolution through the superior naming system has completed, is called the *subordinate* naming system.

2.2 Elements of the XFN Model

2.2.1 XFN Composite Names

An *XFN composite name* is a composite name that has the syntactic and structural properties defined by XFN. XFN defines the basic semantics of composite name resolution, in a manner analogous to compound name resolution. Individual naming systems determine the resolution of each component. The composite name resolution process is described in more detail in Section 4.3 on page 63.

An XFN composite name is represented by the type **FN_composite_name_t**. Operations are provided to manipulate the type as a list of string components. For example, the composite name:

```
( sales.Wiz.COM, usr/local/bin )
```

has two components, a DNS name (**sales.Wiz.COM**) and a UNIX pathname (**usr/local/bin**).

XFN also defines a standard string form for composite names, and provides functions to translate from the standard string form to the structural form, and vice-versa. These are described in detail in Section 4.1 on page 55.

2.2.2 XFN References

An *XFN reference* consists of a type and a list of addresses. The type at this level is intended to identify the type of object. An address is an item which can be used with some communication mechanism to invoke operations on an object or service. Multiple addresses are intended to identify multiple communication endpoints for a single conceptual object or service. These multiple addresses may be required because the object is distributed or because the object can be accessed through more than one communication mechanism. Although the XFN reference representation does not have a size limit, XFN recommends that they contain address information of objects, rather than the objects themselves.

An address in an XFN reference consist of an opaque data buffer and an address type identifier. The address type determines the format and interpretation of the address data. Together, the address's type and data should specify the scheme to reach the object. Specifically, they determine a communication mechanism, an interface over this communication mechanism, and the object's address for this communication mechanism. Many specific schemes are possible. XFN does not specify a particular scheme in general. However, XFN does specify the interpretation of certain types of addresses encountered during the resolution of composite names. Appendix G contains a list of these address types.

Because the types of addresses and the communication mechanisms that can be represented are not restricted, the XFN specifications do not imply any specific properties of addresses such as their stability, validity, or reachability. The ability of a client to lookup a name carries no guarantee that the client can use the returned reference. The client may not have support for any of the necessary communication mechanisms, or may lack the necessary network connectivity to reach the address. The address may be invalid from that origin, or stale; these issues are the province of convention between the name's binder, the clients, and the service provider specified in the address.

In a context, a name must be bound to a reference. Some naming services may not always have reference information for all names in their contexts; for those names, such naming services may return a special reference whose type indicates that the name is not bound to any address. This reference may be naming service-specific or it may be the conventional NULL reference defined in Appendix G.

2.2.3 XFN Context

An *XFN context* is a context that exports the XFN context interface described in Section 3.2 on page 20.

In an XFN context, an atomic name is bound to an XFN reference.

2.2.4 XFN Initial Context

Every XFN name is interpreted relative to some context, and every XFN naming operation is performed on a context object. The XFN interface provides a function that allows the client to obtain an *initial context* object that provides a starting point for resolution of composite names.

XFN policies described in Chapter 5 and optional additional policies described in Appendix D specify a set of names that the client can expect to find in this context and the semantics of the bindings. This provides the initial pathway to other XFN contexts.

2.2.5 XFN Links

An *XFN link* is a special form of XFN reference which has a composite name as an address. Like any other type of XFN reference, an XFN link is bound to an atomic name in an XFN context.

Normal resolution of names in context operations always follows XFN links. See Section 4.4 on page 69 for details of how resolution is affected by links.

A link is bound to a name using the normal bind operation, and unbound using the normal unbind operation. Operations are provided for constructing a link from a composite name. Since normal resolution always follows links, a separate operation is provided to lookup the link itself.

Many naming systems support a native notion of link that may be used within the naming system itself. XFN does not specify whether there is any relationship between such native links and XFN links.

2.2.6 XFN Attributes

Each named object is associated with a set of zero or more *XFN attributes.* Each attribute in the set has a unique *attribute identifier*, an *attribute syntax*, and a set of zero or more distinct *attribute values.*

As is the case with XFN references, the XFN attribute representation does not have a size limit. However, XFN recommends that XFN attributes be used to store attribute information about an object, rather than store the object itself.

XFN defines the base attribute interface for examining and modifying the values of attributes associated with existing named objects. These objects may be contexts or other types of objects. XFN also defines additional interfaces for doing attribute-based searches.

XFN specifies that the attribute set associated with a context contains the attribute *fn_syntax_type*, which describes how the context parses compound names (see Section 3.8 on page 50).

2.3 XFN Usage and Implementation Models

This section describes how the XFN interface can be used and implemented. The usage model is from a client's perspective. The implementation model is from a service provider's perspective.

2.3.1 Basic Usage Model

Most clients of the XFN interface will only be interested in lookups. Their usage of the XFN interface will amount essentially to:

- obtaining the Initial Context
- looking up one or more names relative to the Initial Context.

Once the client obtains a desired reference, it constructs a client-side representation of the object from the reference. This need not involve code within the application layer. For example, RPC services can provide clients with a means of constructing client-side handles from a composite name for the service, or from a reference containing an RPC address for the service. After getting this handle, the client performs all further operations on the object or service by supplying the handle.

This is the basic model of how the XFN interface is expected to be used. The XFN enterprise policies presented in Appendix D further encourage a bind/lookup model for how services and clients may rendezvous through the use of the naming service.

Applications may also use federated naming services indirectly through other interfaces. For example, consider a UNIX application that obtains a filename that it later supplies to the UNIX *open*( ) function. If the system provides XFN support for resolution of filenames, the application need not be aware that the strings it deals with are composite names rather than the traditional local pathnames. Some applications can thereby support the use of composite names without modification.

2.3.2 Basic Implementation Model

The XFN specifications do not dictate a specific way of providing support for the XFN interface. The implementor has great flexibility in terms of how to provide implementations of the XFN service for new and existing naming systems in different environments, based on different requirements in those environments. A potential implementor may find the basic model described below useful in order to understand how to provide the required functionality. This model and additional models are described in more detail in Appendix C, which provides the implementor guidelines on how different configurations of XFN with new and existing naming systems might be implemented.

The XFN client interface can be implemented in a layered manner. Figure 2-1 shows examples of this approach for a few existing naming services.

The top layer (labeled XFN API) is the XFN client interface with which callers interact directly. It implements the types of parameters used in the XFN interface.

In the bottom layer (labeled Context Implementation) are implementations of the XFN interface over specific naming services. Each such implementation over a specific naming service is based on knowledge of that naming service and essentially maps the XFN interface onto that service. It is expected that most of these implementations will work over existing naming services using the existing naming service interfaces and protocols.

Alternatively, the bottom layer could be a generic implementation that supports the XFN interface using a client/server model instead of a naming service-specific implementation. For example, there could be an XFN/DCE service to which the client invokes XFN requests. No

particular naming service-specific implementation or generic implementation is a mandatory component of XFN.

The middle layer (labeled XFN Framework) implements operations that span naming systems using interfaces presented in the bottom layer. This layer is responsible for guiding composite name resolution and invoking the proper operations for each context as required.

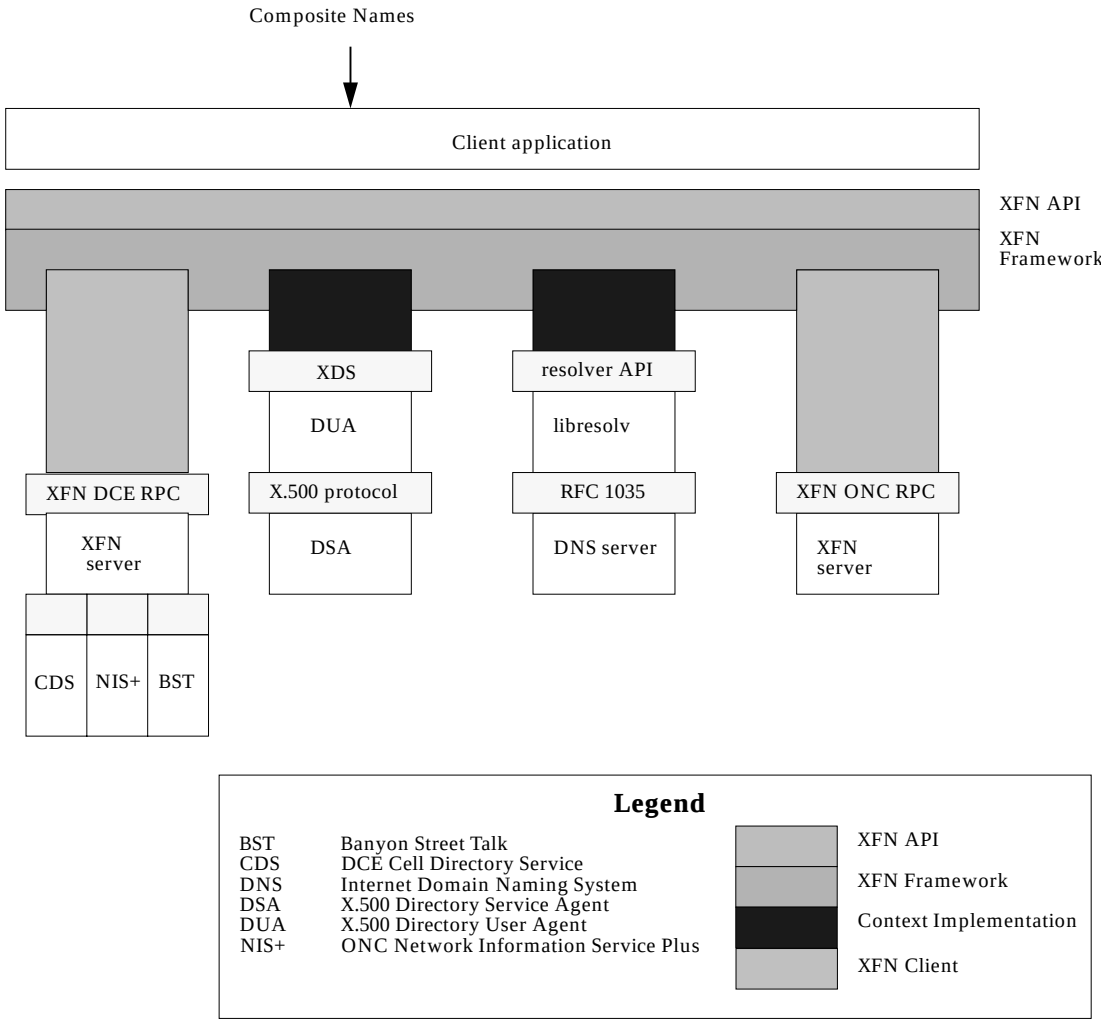

Figure 2-1 Example of an Implementation Model

2.4 Relationship of XFN to other Naming-related Services

2.4.1 Security

XFN does not define a security model nor a common security interface for contexts. Security-related operations, such as those required for authentication or access control, fall into the category of administrative interfaces which are expected to differ widely amongst federation members. Any single choice of security model at the XFN layer is prone to be fundamentally incompatible with the security provided by other direct interfaces, thereby giving rise to inconsistent protection that is susceptible to compromise. Consequently, the security administration interface is kept entirely separate from the federated naming service interface, outside the scope of XFN.

XFN does, however, provide means by which security can be integrated with specific XFN implementations. Operations that fail due to security-related problems can indicate in the status code the nature of the failure.

Authentication

Authentication is to be handled in the modules that implement the service interfaces for each particular member naming system. It occurs as part of the communication that occurs between the client and the naming service. Significant engineering issues remain when multiple security mechanisms and administrative domains are involved. These problems can be addressed on a one-to-one basis, or via a general federated security model, outside the scope of XFN.

XFN provides the status code [FN_E_AUTHENTICATION_FAILURE] to indicate that an operation failed due to authentication errors.

Access Control

Given the ability to authenticate the principal making a service request, a context service provider must then decide whether this principal should be granted or denied the request. Access control is to be handled through additional interfaces in the specific contexts. This can be done by having operations that control default authorisation at context creation and binding creation times, and having interfaces that modify the current authorisation settings. This is analogous to the *umask* and *chmod* scheme for POSIX.1 files.

The access control model is outside the scope of XFN.

XFN provides the status codes [FN_E_CTX_NO_PERMISSION] and [FN_E_ATTR_NO_PERMISSION] to indicate that an operation failed due to access control errors. In the case that the principal is not authorised to know that access has been denied due to permission problems, the status code [FN_E_NAME_NOT_FOUND] or [FN_E_NO_SUCH_ATTRIBUTE] is returned.

2.4.2 Caching

Caching techniques are primarily used for performance improvement. Any aspect of the mechanism that manifests to the client interface is specific to the implementation. Each underlying naming service typically has its own caching mechanism, with its own specific strategies for caching and cache invalidation. Hence, caching at the federation level is not necessary, nor is it likely to be substantially helpful if naming service-specific caches can be reached without doing expensive operations. Caching services such as prefix caching at component boundaries may be supported but should be designed so as not to counter the intended semantics provided by naming service-specific caches.

Caching interfaces for context service providers are outside the scope of XFN.

2.4.3 Replication

Naming services typically use replication to improve fault tolerance and possibly provide better failure semantics through group masking of the underlying server failures.

The XFN model is designed to accommodate and maximise the intended benefits of different replication models. For example, the semantics of the unbind operation are defined so as to not need precise serialising capabilities — semantics that can be readily supported by most existing replication schemes.

XFN references provide a flexible means by which replication can be supported. An XFN reference can have multiple addresses. Some naming services can choose to use this capability to support replication. Others might encode service location information in one address, and use the facilities provided by the client side of the naming service for service location and selection.

Interfaces for administering replication are outside the scope of XFN.

2.5 XFN and Internationalisation

Within XFN, different representations (encodings) may be used to represent character strings. The possible representations can be classified into one of the following:

Portable representation
: This is the set of characters listed in Section 4.1.2 on page 56. The encoding used for the portable representation is ISO 646 (same encoding as ASCII). Use of characters within the portable representation guarantees that the character strings are handled correctly by all implementations. All XFN implementations are required to support the portable characters for component names.

Locale-specific representation
: This is the set of locale-specific characters whose representation is implementation-dependent. Use and proper handling of strings that consist of locale-specific characters is implementation-dependent. Support for locale-specific representations is optional for XFN implementations.

Universal representation
: This is a single format into which all character strings, and thus, all characters within all strings, can be converted. The encoding used for the universal representation is UTF-8 (also known as File System Safe UCS Transformation Format, FSS-UTF) form of ISO 10646. Use and proper handling of strings consisting of characters in the universal representation is implementation-dependent. Support for universal representations is optional for XFN implementations.

For all representations, XFN provides the ability to associate locale-specific information with strings. Specifically, this information includes the encoding (code set/character set), byte size, character count, and language/territory. Code set information within XFN is specified by an OSF code set registry currently defined in DCE RFC 40.1. This registry is being extended to include language/territory registration.

Names that are in the portable representation maximise the benefits of portability and world-wide interoperability. Names that use characters other than the portable representation might improve local usability at the expense of portability and interoperability.

Specially defined XFN characters and names have the same encoding as they would in ISO 646 unless qualified otherwise.

Chapter 3

Interface Overview

This chapter presents an overview of the XFN client interfaces.

The XFN client interfaces consist primarily of two basic interfaces — the basic context interface and the basic attribute interface — and one extended attribute interface.

The base context interface provides the operations for naming, such as binding a name to a reference, looking up the reference bound to a name, unbinding a name.

The base attribute interface provides operations to examine and modify attributes associated with named objects.

The XFN client interface also contains an extended attribute interface consisting of operations to do searching and creation of objects in the namespace with attributes.

In addition, the XFN interface contains:

- operations on the status object and status codes used in the context and attribute operations
- a number of abstract data types defined to represent objects passed to and returned from the context and attribute operations, such as composite names, references and attributes. The XFN client interface defines these types and operations on these types.
- a standard model and operations for parsing compound names, whose syntax is specific to a naming system. These are of primary interest to service implementors.
- operations for manipulating objects that are used to specify the criteria of extended search operations.

An overview of these are provided in this chapter; detailed descriptions of them are given in Chapter 6.

3.1 Naming Conventions of the C Interface

The XFN interface is presented in ISO standard C, which is equivalent to ANSI standard C (see reference ISO C).

The symbols defined by the interface are prefixed by ‘‘fn_’’ or ‘‘FN_’’, for ‘‘Federated Naming’’.

The ‘‘FN_’’ prefix is used for data types and predefined constants.

The ‘‘fn_’’ prefix is used for function names.

Predefined constants appear in all upper case.

Names of functions in the base context interface have the prefix ‘‘fn_ctx_’’.

Names of functions in the attribute interfaces have the prefix ‘‘fn_attr_’’.

3.2 The Base Context Interface

This section describes the operations in the base context interface. The interfaces of the parameters and return values of these operations are described in Section 3.6 on page 42.

3.2.1 Names in Context Operations

In most of the operations of the base context interface, the caller supplies a context and a composite name argument. The supplied composite name is always interpreted relative to the supplied context.

The operation may eventually be effected on a different context called the operation's *target context*. Each operation has an initial resolution phase that conveys the operation to its target context, following which the operation is applied. The effect (but not necessarily the implementation) is that of:

1. doing a lookup on that portion of the name that represents the target context

   and then:

2. invoking the operation on the target context.

The contexts involved only in the resolution phase are called *intermediate contexts*.

Normal resolution of names in context operations always follows XFN links.

3.2.2 Requirements for Supporting the Context Operations

The lookup operation, *fn_ctx_lookup*( ), must be supported by all contexts. Contexts may indicate that they do not support other operations by returning a [FN_E_OPERATION_NOT_SUPPORTED] status code (see Section 3.5 on page 37).

XFN contexts are required to support the resolution phase of every operation in the XFN base context and attribute interfaces when involved in the operation as intermediate contexts. That is, each intermediate context must participate in the process of conveying the operation to the target context, even if it does not support that operation itself. For example, though not all XFN contexts need allow binding and listing names, they must support the resolution phase of these operations.

A name is passed to an XFN context implementation in a structural form as an ordered sequence of components. When resolving a name, the context implementation is responsible for:

1. determining which set of leading components it must resolve
2. resolving that portion to a reference
3. returning a status object containing this reference and the portion of the name unresolved.

More discussion on this topic appears in Section 4.3 on page 63.

3.2.3 Status Objects

In each context operation, the caller may supply an **FN_status_t** parameter, which the called function will set as described in Section 3.5 on page 37. The caller may supply a NULL pointer for this parameter, in which case, no status information is returned.

This holds true for each operation in the base context interface and will not be restated in the individual operation descriptions.

3.2.4 Context Operations

This section describes each context operation, its syntax (in C) and semantics.

3.2.4.1 Construct Handle to Initial Context

Interface:

```
FN_ctx_t *fn_ctx_handle_from_initial(
    unsigned int authoritative,
    FN_status_t *status);
```

Description:

This operation returns a handle to the caller's Initial Context. On successful return, the handle points to a context which meets the specification of the XFN Initial Context described in Section 5.3 on page 73.

authoritative specifies whether the handle to the Initial Context returned should be authoritative with respect to information the context obtains from the naming service. When the flag is non-zero, subsequent operations on this context handle will access the most authoritative information. When *authoritative* is zero, the handle to the Initial Context returned need not be authoritative. Authoritativeness is determined by specific naming services.

3.2.4.2 Lookup

Interface:

```
FN_ref_t *fn_ctx_lookup(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

Description:

This operation returns the reference bound to *name* relative to the context *ctx*.

3.2.4.3 List Names

Interface:

```
FN_namelist_t *fn_ctx_list_names(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);

FN_string_t *fn_namelist_next(
    FN_namelist_t *nl,
    FN_status_t *status);

void fn_namelist_destroy(
    FN_namelist_t *nl);
```

Description:

This set of operations is used to list the names bound in the target context named *name* relative to the context *ctx*. *name* must name a context. If the intent is to list the contents of *ctx*, *name* should be an empty composite name.

The call to *fn_ctx_list_names*( ) initiates the enumeration process for the target context. It returns a handle to an **FN_namelist_t** object that can be used to enumerate the names in the target context.

The operation *fn_namelist_next*( ) returns the next name in the enumeration identified by *nl* and updates *nl* to indicate the state of the enumeration. Successive calls to *fn_namelist_next*( ) using *nl* return successive names in the enumeration and further update the state of the enumeration. *fn_namelist_next*( ) returns a NULL pointer when the enumeration has been completed.

fn_namelist_destroy( ) is used to release resources used during the enumeration. This may be invoked before the enumeration has completed to terminate the enumeration.

The names enumerated using the list names operations are not ordered in any way. There is no guaranteed relation between the order in which names are added to a context and the order of names obtained by enumeration. The specification does not guarantee that any two series of enumerations will return the names in the same order.

When a name is added to or removed from a context, this may not necessarily invalidate the enumeration handle that the client holds for that context. If the enumeration handle remains valid, the update may or may not be visible to the client.

3.2.4.4 List Bindings

Interface:

```
FN_bindinglist_t *fn_ctx_list_bindings(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);

FN_string_t *fn_bindinglist_next(
    Fn_bindinglist_t *bl,
    FN_ref_t **ref,
    FN_status_t *status);

void fn_bindinglist_destroy(
    Fn_bindinglist_t *bl);
```

Description:

This set of operations is used to list the names and bindings in the target context named by *name* relative to the context *ctx*. *name* must name a context. If the intent is to list the contents of *ctx*, *name* should be an empty composite name.

The semantics of these operations are similar to those for listing names. In addition to a name string being returned, *fn_bindinglist_next*( ) also returns the reference of the binding for each member of the enumeration.

3.2.4.5 Bind

Interface:

```
int fn_ctx_bind(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    const    FN_ref_t *ref,
    unsigned int exclusive,
    FN_status_t *status);
```

Description:

This operation binds the supplied reference *ref* to the supplied composite name *name*, resolved relative to *ctx*. The binding is made in the target context — that named by all but the terminal atomic part of *name*. The operation binds the terminal atomic name to the supplied reference in the target context. The target context must already exist.

The value of *exclusive* determines what happens if the terminal atomic part of the name is already bound in the target context. If *exclusive* is non-zero and *name* is already bound, the operation fails. If *exclusive* is zero, the new binding replaces any existing binding.

The value of *ref* cannot be NULL. If the intent is to reserve a name using the *fn_ctx_bind*( ) operation, a reference containing no address should be bound. This reference may be naming service-specific or it may be the conventional NULL reference defined in Appendix G.

3.2.4.6 Unbind

Interface:

```
int fn_ctx_unbind(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

Description:

This operation removes the terminal atomic name in *name* from the target context — that named by all but the terminal atomic part of *name*.

This operation is successful even if the terminal atomic name was not bound in the target context, but fails if any of the intermediate names are not bound. *fn_ctx_unbind*( ) is idempotent.

3.2.4.7 Rename

Interface:

```
int fn_ctx_rename(
    FN_ctx_t *ctx,
    const FN_composite_name_t *oldname,
    const FN_composite_name_t *newname,
    unsigned int exclusive,
    FN_status_t *status);
```

Description:

This operation binds the reference currently bound to *oldname*, resolved relative to *ctx*, to the name *newname*, and unbinds *oldname*. *newname* is resolved relative to the target context — that named by all but the terminal atomic part of *oldname*.

If *exclusive* is zero, rename overwrites any old binding of *newname*. If *exclusive* is non-zero, the operation fails if *newname* is already bound.

The only restriction that XFN places on *newname* is that it be resolved relative to the target context. XFN does not specify further restrictions on *newname*. For example, in some implementations, *newname* might be restricted to be a name in the same naming system as the terminal component of *oldname*. In another implementation, *newname* might be restricted to be an atomic name.

3.2.4.8 Create Subcontext

Interface:

```
FN_ref_t *fn_ctx_create_subcontext(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

Description:

This operation creates a new XFN context of the same type as the target context — that named by all but the terminal atomic component of *name* — and binds it to the composite name *name* relative to the context *ctx*, and returns a reference to the newly created context.

As with the bind operation, the target context must already exist. The new context is created and bound in the target context using the terminal atomic name in *name*.

The operation fails if the terminal atomic name already exists in the target context.

The new subcontext is an XFN context and is created in the same naming system as the target context. XFN does not specify any further properties of the new subcontext. The target context and its naming system determine these.

3.2.4.9 Destroy Subcontext

Interface:

```
int fn_ctx_destroy_subcontext(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

Description:

This operation destroys the subcontext named by *name*, interpreted relative to *ctx*, and unbinds the name.

As in the unbind operation, the operation succeeds if the terminal atomic name is not bound in the target context — that named by all but the terminal atomic name in *name*.

Some aspects of this operation are not specified by XFN, but are determined by the target context and its naming system. For example, XFN does not specify what happens if the named subcontext is non-empty when the operation is invoked.

3.2.4.10 Lookup Link

Interface:

```
FN_ref_t *fn_ctx_lookup_link(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

Description:

This operation returns the XFN link bound to *name* if *name* is bound to an XFN link; otherwise, it returns the reference bound to *name*.

The normal *fn_ctx_lookup*() operation follows all links encountered, including any bound to the terminal atomic part of *name*. This operation differs from the normal lookup in that when the

terminal atomic part of *name* is an XFN link, this link is not followed, and the operation returns the link.

3.2.4.11 Construct Context Handle from Reference

Interface:

```
FN_ctx_t* fn_ctx_handle_from_ref(
    const FN_ref_t *ref,
    unsigned int authoritative,
    FN_status_t *status);
```

Description:

This operation creates a handle to an **FN_ctx_t** object using an **FN_ref_t** object for that context.

authoritative specifies whether the handle to the context returned should be authoritative with respect to information the context obtains from the naming service. When the flag is non-zero, subsequent operations on the context will access the most authoritative information. When *authoritative* is zero, the handle to the context returned need not be authoritative. Authoritativeness is determined by specific naming services.

3.2.4.12 Get Reference to Context

Interface:

```
FN_ref_t *fn_ctx_get_ref(
    const FN_ctx_t *ctx,
    FN_status_t *status);
```

Description:

This operation returns a reference to the supplied context object.

3.2.4.13 Get Syntax Attributes of Context

Interface:

```
FN_attrset_t *fn_ctx_get_syntax_attrs(
    FN_ctx_t *ctx,
    const   FN_composite_name_t *name,
    FN_status_t *status);
```

Description:

This operation returns the syntax attributes associated with the context named by *name* relative to the context *ctx*.

The syntax attributes are described in Section 3.8 on page 50.

This operation is different from other XFN attribute operations in that these syntax attributes could be obtained directly from the context. Attributes obtained through other XFN attribute operations may not necessarily be associated with the context; they may be associated with the reference of the context, rather than the context itself (see Section 3.3.2 on page 27).

3.2.4.14 Destroy Context Handle

Interface:

```
void fn_ctx_handle_destroy(FN_ctx_t *ctx);
```

Description:

This operation destroys the context handle *ctx* and allows the implementation to free resources associated with the context handle. This operation does not affect the state of the context itself.

3.2.4.15 Construct an Equivalent Name: Preliminary Specification

This section is assigned X/Open **Preliminary Specification** status (not **CAE Specification** status). For explanation of the difference between **Preliminary** and **CAE** specifications, see the description under **X/Open Technical Publications** in the **Preface**.

Interface:

```
FN_composite_name_t *fn_ctx_equivalent_name(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_string_t *leading_name,
    FN_status_t *status);
```

Description:

Given the name of an object *name* relative to the context *ctx*, this operation returns an equivalent name for that object, relative to the same context *ctx*, that has *leading_name* as its initial atomic name. Two names are said to be equivalent if they have prefixes that resolve to the same context, and the parts of the names immediately following the prefixes are identical.

If an equivalent name cannot be constructed, the value 0 is returned and the *status* argument is set appropriately.

3.3 The Base Attribute Interface

This section describes the operations in the base attribute interface. The interfaces of the parameters and return values of these operations are described in Section 3.6 on page 42.

3.3.1 XFN Attribute Model

XFN assumes the following model for attributes. A set of zero or more attributes is associated with a named object. Each attribute in the set has a unique attribute identifier, an attribute syntax and a set of zero or more distinct attribute values. Each attribute value has an opaque data type. The attribute identifier serves as a name for the attribute. The attribute syntax indicates how the value is encoded in the buffer.

The operations in the base attribute interface may be used to examine and modify the settings of attributes associated with existing named objects. These objects may be contexts or other types of objects.

The range of support for attribute operations may vary widely. Some naming systems may not support any attribute operations. Other naming systems may only support read operations, or operations on attributes whose identifiers are in some fixed set. A naming system may limit attributes to have a single value, or may require at least one value. Some naming systems may only associate attributes with context objects, while others may allow associating attributes with non-context objects.

The resolution phase of every attribute operation must be supported.

Typically, attributes of an object are manipulated through operations that operate on a single attribute, such as reading or updating a single attribute. In addition, it is expected that a client is able to read all attribute values of a single attribute in one call. However, sometimes there is a requirement to manipulate several attributes of a single object, or to obtain individual attribute values of a single attribute from the naming service. To address these requirements, XFN defines two kinds of attribute operations:

- single attribute operations
- multiple value and multiple attribute operations.

3.3.2 Relationship to Naming Operations

In XFN, an attribute operation using a composite name is not necessarily equivalent to an independent *fn_ctx_lookup*( ) operation followed by an attribute operation in which the caller supplies the resulting reference and an empty name. This is because there are a range of attribute models in which an attribute is associated with a name in a context, or an attribute is associated with the object named, or both. XFN accommodates all of these alternatives. Invoking an attribute operation using the target context and the terminal atomic name accesses either the attributes that are associated with the target name or target named object — this is dependent on the underlying attribute model. This document uses the term *attributes associated with a named object* to refer to all of these cases.

XFN provides no guarantee about the relationship between the attributes and the reference associated with a given name. Some naming systems may store the reference bound to a name in one or more attributes associated with a name. Attribute operations might affect the information used to construct a reference.

To avoid undefined results, applications that intend to manipulate references must use the operations in the context interface and not the attribute operations. Applications should avoid the use of specific knowledge about how an XFN context implementation over a particular naming system constructs references.

3.3.3 XFN Links

Operations in the base attribute interface that involve name resolution accept a *follow_link* parameter. The value of *follow_link* determines the behaviour of the operation when the terminal atomic part of the name being resolved is bound to an XFN link:

- If *follow_link* is non-zero, such a link is followed and the attribute associated with the final named object is examined or modified.
- If *follow_link* is zero, such a link is not followed.

Any XFN links encountered before the terminal atomic name are always followed.

XFN does not specify how *follow_links* affects the following of native naming system links.

3.3.4 Status Objects

In each attribute operation, the caller may supply an **FN_status_t** parameter, which the called function will set as described in Section 3.5 on page 37. The caller may supply a NULL pointer for this parameter, in which case, no status information is returned.

This holds true for each operation in the base attribute interface and will not be restated in the individual operation descriptions.

3.3.5 Single-Attribute Operations

3.3.5.1 Get Attribute

Interface:

```
FN_attribute_t *fn_attr_get(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    const    FN_identifier_t *attribute_id,
    unsigned int follow_link,
    FN_status_t *status);
```

Description:

This operation returns the identifier, syntax and values of a specified attribute for the object named *name* relative to *ctx*. If *name* is empty, the attribute associated with *ctx* is returned.

fn_attr_get_values( ) and its related functions (described below) are for getting individual values of an attribute and should be used if the combined size of all the values are expected to be too large to be returned in a single invocation of *fn_attr_get*( ).

3.3.5.2 Modify Attribute

Interface:

```
int fn_attr_modify(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    unsigned int mod_op,
    const    FN_attribute_t *attr,
    unsigned int follow_link,
    FN_status_t *status);
```

Description:

This operation modifies according to *mod_op* the attribute *attr* associated with the object named *name* relative to *ctx*. If *name* is empty, the attribute associated with *ctx* is modified.

The modification is made on the attribute identified by the attribute identifier of *attr*. The syntax and values of *attr* are use according to the modification operation. The modification operations are described in Table 3-1.

Operation Code	Meaning
FN_ATTR_OP_ADD	Add an attribute with given attribute identifier and set of values. If an attribute with this identifier already exists, replace the set of values with those in the given set. The set of values may be empty if the target naming system permits.
FN_ATTR_OP_ADD_EXCLUSIVE	Add an attribute with the given attribute identifier and set of values. The operation fails with [FN_E_ATTR_IN_USE] if an attribute with this identifier already exists. The set of values may be empty if the target naming system permits.
FN_ATTR_OP_REMOVE	Remove the attribute with the given attribute identifier and all of its values. The operation succeeds even if the attribute does not exist. The values of the attribute supplied with this operation are ignored.
FN_ATTR_OP_ADD_VALUES	Add the given values to those of the given attribute (resulting in the attribute having the union of its prior value set with the set given). Create the attribute if it does not exist already. The set of values may be empty if the target naming system permits.
FN_ATTR_OP_REMOVE_VALUES	Remove the given values from those of the given attribute (resulting in the attribute having the set difference of its prior value set and the set given). This succeeds even if some of the given values are not in the set of values that the attribute has. In naming systems that require an attribute to have at least one value, removing the last value will remove the attribute as well.

Table 3-1 XFN Attribute Modification Operations

3.3.6 Operations on Multiple Values

3.3.6.1 Get Attribute Values

This set of operations allows the caller to obtain attribute values associated with a single attribute individually.

Interface:

```
FN_valuelist_t *fn_attr_get_values(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    const     FN_identifier_t *attribute_id,
    unsigned int follow_link,
    FN_status_t *status);

FN_attrvalue_t *fn_valuelist_next(
    FN_valuelist_t *vl,
    FN_identifier_t **attr_syntax,
    FN_status_t *status);

void fn_valuelist_destroy(
    FN_valuelist_t *vl);
```

Description:

This set of operations is used to obtain the values of a single attribute, identified by *attribute_id*, associated with the object named *name*, resolved in the context *ctx*. If *name* is empty, the attribute values associated with *ctx* are obtained.

This interface should be used instead of *fn_attr_get*( ) if the combined size of all the values is expected to be too large to be returned by *fn_attr_get*( ).

The operation *fn_attr_get_values*( ) initiates the enumeration process. It returns a handle to an **FN_valuelist_t** object that can be used for subsequent *fn_valuelist_next*( ) calls to enumerate the values requested.

The operation *fn_valuelist_next*( ) returns the next attribute value in the enumeration and updates *vl* to indicate the state of the enumeration. It also returns the attribute's syntax identifier (this is the same for all values of a single attribute).

The operation *fn_valuelist_destroy*( ) frees the resources used during with the enumeration. This may be invoked before the enumeration has completed to terminate the enumeration.

3.3.7 Operations on Multiple Attributes

These operations allow the caller to specify an operation that operates on multiple attributes using one or more calls.

The failure semantics of these operations may vary widely across naming systems. In some systems the single function call may comprise multiple individual naming system operations, with no guarantees of atomicity.

3.3.7.1 Get Attribute Identifiers

Interface:

```
FN_attrset_t *fn_attr_get_ids(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    unsigned int follow_link,
    FN_status_t *status);
```

Description:

This operation returns a list of the attribute identifiers of all attributes associated with the object named by *name* relative to the context *ctx*. If *name* is empty, the attribute identifiers associated with *ctx* are returned.

3.3.7.2 Get Multiple Attributes

Interface:

```
FN_multigetlist_t *fn_attr_multi_get(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    const     FN_attrset_t *attr_ids,
    unsigned int follow_link,
    FN_status_t *status);

FN_attribute_t *fn_multigetlist_next(
    FN_multigetlist_t *ml,
    FN_status_t *status);

void fn_multigetlist_destroy(
    FN_multigetlist_t *ml);
```

Description:

This set of operations returns one or more attributes associated with the object named by *name* relative to the context *ctx*. If *name* is empty, the attributes associated with *ctx* are returned.

The attributes returned are those specified in *attr_ids*. If the value of *attr_ids* is 0, all attributes associated with the named object are returned. Any attribute values in *attr_ids* provided by the caller are ignored; only the attribute identifiers are relevant for this operation. Each attribute (identifier, syntax, values) is returned one at a time using an enumeration scheme similar to that for listing a context.

fn_attr_multi_get( ) initiates the enumeration process. It returns a handle to an **FN_multigetlist_t** object that can be used for subsequent *fn_multigetlist_next*( ) calls to enumerate the attributes requested.

The operation *fn_multigetlist_next*( ) returns the next attribute (identifier, syntax, and values) in the enumeration and updates *ml* to indicate the state of the enumeration. Successive calls to *fn_multigetlist_next*( ) using *ml* return successive attributes in the enumeration and further update the state of the enumeration.

The operation *fn_multigetlist_destroy*( ) frees the resources used during with the enumeration. This may be invoked before the enumeration has completed to terminate the enumeration.

Implementations are not required to return all attributes requested by *attr_ids*. Some may choose to return only the attributes found successfully; such implementations may not

necessarily return identifiers for attributes that could not be read.

3.3.7.3 Modify Multiple Attributes

Interface:

```
int fn_attr_multi_modify(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    const    FN_attrmodlist_t *mods,
    unsigned int follow_link,
    FN_attrmodlist_t **unexecuted_mods,
    FN_status_t *status);
```

Description:

This operation modifies the attributes associated with the object named *name* relative to *ctx*. If *name* is empty, the attributes associated with *ctx* are modified.

In the *mods* parameter, the caller specifies a sequence of modifications that are to be done in order on the attributes. Each modification in the sequence specifies a modification operation code (Table 3-1) and an attribute on which to operate.

If all the modifications were performed successfully, *unexecuted_mods* is a NULL pointer.

If an error is encountered while performing the list of modifications, *status* indicates the type of error and *unexecuted_mods* is set to a list of unexecuted modifications. The contents of *unexecuted_mods* do not share any state with *mods*; items in *unexecuted_mods* are copies of items in *mods* and appear in the same order in which they were originally supplied in *mods*. The first operation in *unexecuted_mods* is the first one that failed and the code in *status* applies to this modification operation in particular. If *status* indicates failure and a NULL pointer is returned in *unexecuted_mods*, that indicates no modifications were executed.

3.4 The Extended Attribute Interface

The XFN extended attribute interface consists of operations to do searching and creation of objects in the namespace with attributes.

3.4.1 The Attribute Search Interface: Preliminary Specification

This section (including all its sub-sections) is assigned X/Open **Preliminary Specification** status (not **CAE Specification** status). For explanation of the difference between **Preliminary** and **CAE** specifications, see the description under **X/Open Technical Publications** in the **Preface**.

There are two operations in the search interface: a basic search operation which performs associative lookup in a single context, and an extended search operation that allows the search criterion to be specified using an expression and allows the scope of the search to encompass a wider scope than just a single context.

This section gives an overview of these two search operations. The interfaces of the parameters and return values of these operations are described in Section 3.7 on page 44. The search filter expression used in the extended search is also described in detail in Section 3.7 on page 44.

3.4.1.1 Basic Search

Interface:

```
FN_searchlist_t *fn_attr_search(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrset_t *match_attrs,
    unsigned int return_ref,
    const FN_attrset_t *return_attr_ids,
    FN_status_t *status);

FN_string_t *fn_searchlist_next(
    FN_searchlist_t *sl,
    FN_ref_t **returned_ref,
    FN_attrset_t **returned_attrs,
    FN_status_t *status);

void fn_searchlist_destroy(
    FN_searchlist_t *sl);
```

Description:

This set of operations is used to enumerate names of objects bound in the target context named *name* relative to the context *ctx* with attributes whose values match all those specified by *match_attrs*. *return_ref* specifies whether to return the references of named objects in the search while *return_attr_ids* specifies the attributes to be returned in the search.

The call to *fn_attr_search*( ) initiates the search in the target context. It returns a handle to an **FN_searchlist_t** object that is used to enumerate the names of the objects whose attributes match *match_attrs*.

fn_searchlist_next( ) returns the next name in the enumeration identified by *sl*. The reference of the name, if requested, is returned in *returned_ref*. The attributes specified by *return_attr_ids* are returned in *returned_attrs*. Successive calls to *fn_searchlist_next*( ) using *sl* return successive names, and optionally, references and attributes, in the enumeration and further update the state of the enumeration.

fn_searchlist_destroy( ) releases resources used during the enumeration. It can be called at any time to terminate the enumeration.

3.4.1.2 Extended Search

Interface:

```
FN_ext_searchlist_t *fn_attr_ext_search(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_search_control_t *control,
    const FN_search_filter_t *filter,
    FN_status_t *status);

FN_composite_name_t *fn_ext_searchlist_next(
    FN_ext_searchlist_t *esl,
    FN_ref_t **returned_ref,
    FN_attrset_t **returned_attrs,
    FN_status_t *status);

void fn_ext_searchlist_destroy(
    FN_ext_searchlist_t *esl);
```

Description:

This set of operations is used to list names of objects whose attributes satisfy the filter expression *filter*. The *control* argument encapsulates the option settings for the search. These options are:

1. the scope of the search. This can be one of:
   - search the named object
   - search the context named by *name*.
   - search the entire subtree rooted at the context named by *name*.
   - search the context implementation-defined subtree rooted at the context named by *name*.
2. whether XFN links are followed during the search
3. a limit on the number of names returned
4. whether the reference associated with the named object is returned
5. which attributes associated with the named object are returned

The filter expression is evaluated against the attributes of the objects bound in the scope of the search. The filter evaluates to either TRUE or FALSE.

The call to *fn_attr_ext_search*( ) initiates the search and, if successful, returns a handle to an **FN_ext_searchlist_t** object, *esl*, that is used to enumerate the names of the objects that satisfy the filter.

fn_ext_searchlist_next( ) returns the next name, and optionally, its reference and attributes, in the enumeration identified by *esl*. The name returned is a composite name, to be resolved relative to the starting context for the search. The starting context is the context named *name* relative to *ctx* unless the scope of the search is only the named object. If the scope of the search is only the named object, the terminal atomic name is returned. Successive calls to *fn_ext_searchlist_next*( ) using *esl* return successive names, and optionally, references and attributes, in the enumeration and further update the state of the enumeration.

fn_ext_searchlist_destroy( ) releases resources used during the search and enumeration. It can be called at any time to terminate the enumeration.

3.4.2 Object Creation With Attributes

There are times when it is useful or necessary to associate attributes with an object at the time the object is created. The XFN extended attribute interface contains functions that provide these capabilities. The two functions in this interface, *fn_attr_bind*( ) and *fn_attr_create_subcontext*( ), are analogous to their counterparts in the base context interface, *fn_ctx_bind*( ) and *fn_ctx_create_subcontext*( ), respectively, except the versions in the extended attribute interface allow an extra parameter for specifying attributes to be associated with the new binding.

3.4.2.1 Bind with Attributes

Interface:

```
int fn_attr_bind(
    FN_ctx_t *ctx,
    const   FN_composite_name_t *name,
    const   FN_ref_t *ref,
    const   FN_attrset_t *attrs,
    unsigned int exclusive,
    FN_status_t *status);
```

Description:

This operation binds the supplied reference *ref* to the supplied composite name *name* relative to *ctx*, and associates the attributes specified in *attrs* with the named object. The binding is made in the target context — that context named by all but the terminal atomic part of *name*. The operation binds the terminal atomic name to the supplied reference in the target context. The target context must already exist.

The value of *exclusive* determines what happens if the terminal atomic part of the name is already bound in the target context. If *exclusive* is non-zero and *name* is already bound, the operation fails. If *exclusive* is zero, the new binding replaces any existing binding, and *attrs*, if not NULL, replaces any existing attributes associated with the named object.

3.4.2.2 Create Subcontext with Attributes

Interface:

```
FN_ref_t *fn_attr_create_subcontext(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrset_t *attrs,
    FN_status_t *status);
```

Description:

This operation creates a new XFN context of the same type as the target context — that named by all but the terminal atomic component of *name* — and binds it to the supplied composite name. In addition, attributes given in *attrs* are associated with the newly created context. The target context must already exist. The new context is created and bound in the target context using the terminal atomic name in *name*. The operation returns a reference to the newly created context.

3.5 Status Objects and Status Codes

The result status of operations in the context interface and the attribute interfaces is encapsulated in an **FN_status_t** object. The caller may supply a NULL pointer for this parameter, in which case, no status information is returned. If the caller supplies an **FN_status_t** object to the operation, upon return from the operation, this object will contain information about how the operation completed: whether an error occurred in performing the operation, the nature of the error, and information that helps locate where the error occurred. In the case that the error occurred while resolving an XFN link, the status object contains additional information about that error.

The context status object consists of several items of information:

primary status code	An **unsigned int** code describing the disposition of the operation.
resolved name	In the case of a failure during the resolution phase of the operation, this is the leading portion of the name that was resolved successfully. Resolution may have been successful beyond this point, but the error might not be pinpointed further.
resolved reference	The reference to which resolution was successful (in other words, the reference to which the resolved name is bound).
remaining name	The remaining unresolved portion of the name.
diagnostic message	This contains any diagnostic message returned by the context implementation. This message provides the context implementation a way of notifying the end-user or administrator of any implementation-specific information related to the returned error status. The diagnostic message could then be used by the end-user or administrator to take appropriate out-of-band action to rectify the problem.
link status code	In the case that an error occurred while resolving an XFN link, the primary status code has the value [FN_E_LINK_ERROR] and the link status code describes the error that occurred while resolving the XFN link.
resolved link name	In the case of a link error, this contains the resolved portion of the name in the XFN link.
resolved link reference	In the case of a link error, this contains the reference to which the resolved link name is bound.
remaining link name	In the case of a link error, this contains the remaining unresolved portion of the name in the XFN link.
link diagnostic message	In the case of a link error, this contains any diagnostic message related to the resolution of the link.

Both the primary status code and the link status code are values of type **unsigned int** that are drawn from the same set of meaningful values. XFN reserves the values 0 through 127 for standard meanings. Currently, values and interpretations for the following codes are determined by XFN.

Table 3-2 XFN Status Codes

Code	Meaning
FN_SUCCESS	The operation succeeded.
FN_E_ATTR_IN_USE	When an attribute is being modified using the operation FN_ATTR_OP_ADD_EXCLUSIVE and an attribute with the same identifier already exists, the operation fails with FN_E_ATTR_IN_USE.
FN_E_ATTR_NO_PERMISSION	The caller did not have permission to perform the attempted attribute operation.
FN_E_ATTR_VALUE_REQUIRED	The operation attempted to create an attribute without a value, and the specific naming system does not allow this.
FN_E_AUTHENTICATION_FAILURE	The identity of the client principal could not be verified.
FN_E_COMMUNICATION_FAILURE	An error occurred in communicating with one of the contexts involved in the operation.
FN_E_CONFIGURATION_ERROR	A problem was detected that indicated an error in the installation of the XFN implementation.
FN_E_CONTINUE	The operation should be continued using the remaining name and the resolved reference returned in the status.
FN_E_CTX_NO_PERMISSION	The client did not have permission to perform the operation.
FN_E_CTX_NOT_EMPTY	(Applies only to *fn_ctx_destroy_subcontext*().) The naming system required that the context be empty before its destruction, and it was not empty.
FN_E_CTX_UNAVAILABLE	Service could not be obtained from one of the contexts involved in the operation. This may be because the naming system is busy, or is not providing service. In some implementations this may not be distinguished from a communication failure.

Code	Meaning
FN_E_ILLEGAL_NAME	The name supplied to the operation was not a well-formed XFN composite name, or one of the component names was not well-formed according to the syntax of the naming system(s) involved in its resolution.
FN_E_INCOMPATIBLE_CODE_SETS	The operation involved character strings of incompatible code sets, or the supplied code set is not supported by the implementation.
FN_E_INCOMPATIBLE_LOCALES	The operation involved character strings of incompatible language or territory locale information, or the specified locale is not supported by the implementation.
FN_E_INSUFFICIENT_RESOURCES	Either the client or one of the involved contexts could not obtain sufficient resources (for example, memory, file descriptors, communication ports, stable media space, and so on.) to complete the operation successfully.
FN_E_INVALID_ATTR_IDENTIFIER	The attribute identifier was not in a format acceptable to the naming system, or its contents were not valid for the format specified for the identifier.
FN_E_INVALID_ATTR_VALUE	One of the values supplied was not in the appropriate form for the given attribute.
FN_E_INVALID_ENUM_HANDLE	The enumeration handle supplied was invalid, either because it was from another enumeration, or because an update operation occurred during the enumeration, or because of some other reason.
FN_E_INVALID_SYNTAX_ATTRS	The syntax attributes supplied are invalid or insufficient to fully specify the syntax.
FN_E_LINK_ERROR	There was an error encountered resolving an XFN link encountered during resolution of the supplied name.

Code	Meaning
FN_E_LINK_LOOP_LIMIT	A non-terminating loop (cycle) in the resolution can arise due to XFN links encountered during the resolution of a composite name. This code indicates either the definite detection of such a cycle, or that resolution exceeded an implementation-defined limit on the number of XFN links allowed for a single operation invoked by the caller.
FN_E_MALFORMED_LINK	A malformed link reference was encountered.
FN_E_MALFORMED_REFERENCE	A context object could not be constructed from the supplied reference, because the reference was not properly formed.
FN_E_NAME_IN_USE	(Only for operations that bind names.) The supplied name was already in use.
FN_E_NAME_NOT_FOUND	Resolution of the supplied composite name proceeded to a context in which the next atomic component of the name was not bound.
FN_E_NO_EQUIVALENT_NAME	No equivalent name can be constructed, either because there is no meaningful equivalence between *name* and *leading_name*, or the system does not support constructing the requested equivalent name, for implementation-specific reasons.
FN_E_NO_SUCH_ATTRIBUTE	The object did not have an attribute with the given identifier.
FN_E_NO_SUPPORTED_ADDRESS	A context object could not be constructed from a particular reference. The reference contained no address type over which the context interface was supported.
FN_E_NOT_A_CONTEXT	Either one of the intermediate atomic names did not name a context, and resolution could not proceed beyond this point, or the operation required that the caller supply the name of a context, and the name did not resolve to a reference for a context.
FN_E_OPERATION_NOT_SUPPORTED	The operation attempted is not supported.

Code	Meaning
FN_E_PARTIAL_RESULT	The operation attempted is returning a partial result.
FN_E_SEARCH_INVALID_FILTER	The filter expression had a syntax error or some other problem.
FN_E_SEARCH_INVALID_OP	An operator in the filter expression is not supported or, if the operator is an extended operator, the number of types of arguments supplied does not match the signature of the operation.
FN_E_SEARCH_INVALID_OPTION	A supplied search control option could not be supported.
FN_E_SYNTAX_NOT_SUPPORTED	The syntax type specified is not supported.
FN_E_TOO_MANY_ATTR_VALUES	The operation attempted to associate more values with an attribute than the naming system supported.
FN_E_UNSPECIFIED_ERROR	An error occurred that could not be classified by any of the other error codes.

3.6 Parameters Used in the Interface

This section gives an overview of the types of parameters that are passed and returned by operations in the base context and attribute interfaces. Detailed descriptions of these are in Chapter 6.

Manipulation of these objects using their corresponding interfaces (described in detail in Chapter 6) does not affect their representation in the underlying naming system. Changes to objects in the underlying naming system can only be effected through the use of the interfaces described in Section 3.2 on page 20, Section 3.3 on page 27 and Section 3.4 on page 33.

3.6.1 Composite Names

A composite name is represented by an object of type **FN_composite_name_t**. Abstractly, a composite name is a sequence of components, where each component is a string (of type **FN_string_t**) intended to contain a name from a single naming system. (See Section 4.1 on page 55 for a description of composite name syntax and structure.) Operations are provided to iterate over this sequence, modify it, and compare two composite names.

3.6.2 References and Addresses

An XFN reference is represented by the type **FN_ref_t**. An object of this type contains a reference type and a list of addresses. The ordering in this list at the time of binding might not be preserved when the reference is returned upon lookup.

The reference type is represented by an object of type **FN_identifier_t**. The reference type is intended to identify the class of object referenced. XFN does not dictate the precise use of this.

Each address is represented by an object of type **FN_ref_addr_t**. An address consists of an opaque data buffer and a type field, again of type **FN_identifier_t**. The address type is intended to identify the mechanism that should be used to reach the object using that address. Multiple addresses in a single reference are intended to identify multiple communication endpoints for the same conceptual object. Multiple addresses may arise for various reasons, such as the object offering interfaces over more than one communication mechanism.

The client process must interpret the contents of the opaque buffers based on the type of the address and the type of the reference. However, this interpretation is intended to occur below the application layer. Most applications developers should not have to manipulate the contents of either address or reference objects themselves. These interfaces would generally be used within service libraries.

3.6.3 Identifiers

Identifiers are used to identify reference types and address types in the reference, and to identify attributes and their syntax in the attribute operations.

The type **FN_identifier_t** is used to represent an identifier. It consists of an **unsigned int**, which determines the format of identifier, and the actual identifier, which is expressed as a sequence of octets.

XFN defines a small number of standard forms for identifiers.

Identifier Format	Description
FN_ID_STRING	The identifier is an ASCII string (ISO 646).
FN_ID_DCE_UUID	The identifier is an OSF DCE UUID in string representation. (See the X/Open DCE RPC.)
FN_ID_ISO_OID_STRING	The identifier is an ISO OID in ASN.1 dot-separated integer list string format. (See the ISO ASN.1.)

Table 3-3 XFN Identifier Formats

3.6.4 Strings

The **FN_string_t** type is used to represent character strings in the XFN interface. It provides insulation from specific string representations.

The **FN_string_t** supports multiple code sets. An XFN implementation is required to support ISO 646; all other code sets are optional.

FN_string_t contains operations for string comparison, substring searches and string manipulation.

3.6.5 Attributes and Attribute Values

An attribute has an attribute identifier, a syntax and a set of distinct values. An attribute is represented by the type **FN_attribute_t**. The attribute identifier and its syntax are specified using an **FN_identifier_t**. Each value is a sequence of octets, represented by the type **FN_attrvalue_t**.

There are operations to allow the construction, destruction and manipulation of an attribute and its value set.

3.6.6 Attribute Sets

An attribute set is composed of attribute objects with distinct identifiers. Attribute sets are represented by the type **FN_attrset_t**.

There are operations to allow the construction, destruction and update of an attribute set.

3.6.7 Attribute Modification Lists

An attribute modification list allows for multiple modification operations to be made on the attributes associated with a single named object. An attribute modification list is represented by the type **FN_attrmodlist_t**. An attribute modification specifier consists of an operation specifier and an attribute object. The attribute's identifier indicates the attribute that is to be operated upon. The attribute's values are used in a manner depending on the operation. The operation specifier is one of the values described in Table 3-1 on page 29. The operations are to be performed in the order in which they appear in the list.

3.7 Parameters Used in Extended Search: Preliminary Specification

This complete section (including all its sub-sections) is assigned X/Open **Preliminary Specification** status (not **CAE Specification** status). For explanation of the difference between **Preliminary** and **CAE** specifications, see the description under **X/Open Technical Publications** in the **Preface**.

There are two types of objects used to specify the scope and details of an extended search operation:

- the search control options (**FN_search_control_t**)
- the search filter expression (**FN_search_filter_t**).

3.7.1 Search Control

The **FN_search_control_t** object encapsulates the different options that the application can specify in controlling the scope and the return values of the extended search operation, *fn_attr_ext_search*( ). These options are:

scope of search
This determines which contexts and objects will be searched.

Scope	Meaning
FN_SEARCH_NAMED_OBJECT	Search just the given named object.
FN_SEARCH_ONE_CONTEXT	Search just the given context.
FN_SEARCH_SUBTREE	Search given context and all its subcontexts.
FN_SEARCH_CONSTRAINED_SUBTREE	Search given context and its subcontexts as constrained by the context-specific policy in place at the named context.

Default:
{FN_SEARCH_ONE_CONTEXT}.

follow links during search
This determines whether links encountered during the search will be followed. Note that the initial resolution phase of the operation (the resolution up to the target context) always follow links. This option controls the following of links after reaching the target context.

Default:
Do not follow links.

maximum names returned
This specifies the maximum number of names to be returned before terminating the search. A value of 0 indicates that the search is terminated only when all the context and objects specified by the scope have been searched.

Default:
Return all named objects found.

return reference
This determines whether the reference of the object is returned.

Default:
Do not return the reference.

return attributes
This determines which attributes associated with the named object, if any, are returned.

Default:
Do not return any attributes.

3.7.2 Search Filter

The *fn_attr_ext_search*( ) operation allows the search for named objects whose attributes satisfy a given filter expression. The filter is expressed in terms of logical expressions involving attribute identifiers and their values of named objects examined during the search. The filter is created from an expression string and a list of arguments that replace substitution tokens within the expression string.

3.7.2.1 BNF of Filter Expression

```
<FilterExpr> ::= [ <Expr> ]

<Expr> ::=       <Expr> "or" <Expr>
                 | <Expr> "and" <Expr>
                 | "not" <Expr>
                 | "(" <Expr> ")"
                 | <Attribute> [ <Rel_Op> <Value> ]
                 | <Ext>

<Rel_Op> ::=     "==" | "!=" | "<" | "<=" | ">" | ">=" | "~="

<Attribute> ::=  "%a"

<Value> ::=      <Integer>
                 | "%v"
                 |<Wildcarded_string>

<Wildcarded_string> ::= "*"
                 | <String>
                 | {<String> "*"}+ [<String>]
                 | {"*" <String>}+ ["*"]

<String> ::=     "'" { <Char> } * "'"
                 | "%s"

<Char> ::=       <PCS> // See BNF in Section 4.1.2 for PCS definition
                 | Characters in the repertoire of a string
                 | representation

<Identifier> ::= "%i"

<Ext> ::=        <Ext_Op> "(" [Arg_List] ")"

<Ext_Op> ::=     <String> | <Identifier>

<Arg_List> ::=   <Arg> | <Arg> "," <Arg_List>

<Arg> ::=        <Value> | <Attribute> | <Identifier>
```

3.7.2.2 Specification of Filter Expression

The arguments to *fn_search_filter_create*( ) are a return status, an expression string, and a list of arguments. The string contains the filter expression with substitution tokens for the attributes, attribute values, strings and identifiers that are part of the expression. The remaining list of arguments contains the attributes and values in the order of appearance of their corresponding substitution tokens in the expression. The arguments are of types **FN_attribute_t***, **FN_attrvalue_t***, **FN_string_t*** or **FN_identifier_t***. Except when attributes appear as arguments in specially-defined extended operations, any attribute values in an **FN_attribute_t** type of argument are ignored; only the attribute identifier and attribute syntax are relevant. The argument type expected by each substitution token are listed in Table 3-4.

Token	Argument Type
%a	**FN_attribute_t***
%v	**FN_attrvalue_t***
%s	**FN_string_t***
%i	**FN_identifier_t***

Table 3-4 Substitute Tokens in Search Filter Expressions

3.7.2.3 Precedence

The following precedence relations hold in the absence of parentheses, in the order of lowest to highest:

- or
- and
- not
- relational operators.

These boolean and relational operators are left associative.

3.7.2.4 Relational Operators

Table 3-5 on page 47 contains descriptions of the relational operators.

Comparisons and ordering are specific to the syntax or rules of the supplied attribute.

Locale (code set, language or territory) mismatches that occur during string comparisons and ordering operations are resolved in an implementation-dependent way. Relational operations that have ordering semantics may be used for strings of locales in which ordering is meaningful, but is not of general use in internationalized environments.

An attribute that occurs in the absence of any relational operator tests for the presence of the attribute.

Operator	Meaning
==	the sub-expression is TRUE if at least one value of the specified attribute is equal to the supplied value.
!=	the sub-expression is TRUE if no values of the specified attribute equal the supplied value.
>=	the sub-expression is TRUE if at least one value of the attribute is greater than or equal to the supplied value.
>	the sub-expression is TRUE if at least one value of the attribute is greater then the supplied value.
<=	the sub-expression is TRUE if at least one value of the attribute is less than or equal to the supplied value.
<	the sub-expression is TRUE if at least one value of the attribute is less than the supplied value.
~=	the sub-expression is TRUE if at least one value of the specified attribute matches the supplied value according to some context-specific approximate matching criterion. This criterion must subsume strict equality.

Table 3-5 Relational Operators in Search Filter Expressions

3.7.2.5 Wildcarded Strings

A wildcarded string consists of a sequence of alternating wildcard specifiers and strings. The sequence can start with either a wildcard specifier or a string, and end with either a wildcard specifier or a string.

The wildcard specifier is denoted by the asterisk character (‘*’) and means 0 or more occurrences of any character.

Wildcarded strings can be used to specify substring matches. Table 3-6 on page 48 contains examples of wildcarded strings and their meaning.

Wildcarded String	Meaning
*	any string
‘tom’	the string *tom*
’harv’*	any string starting with *harv*
*’ing’	any string ending with *ing*
‘a’*’b’	any string starting with *a* and ending with *b*
‘a*b’	the string *a*b*
‘jo’*’ph’*’ne’*’er’	any string starting with *jo*, and containing the substring *ph*, and which contains the substring *ne* in the portion of the string following *ph*, and which ends with *er*
%s*	any string starting with the supplied string
‘bix’*%s	any string starting with *bix* and ending with the supplied string

Table 3-6 Examples of Wildcarded Strings

3.7.2.6 Extended Operations

In addition to the relational operators, extended operators can be specified. All extended operators return either TRUE or FALSE. A filter expression can contain both relational and extended operations.

Extended operators are specified using an identifier (**FN_identifier_t**) or a string. If the operator is specified using a string, the string is used to construct an identifier of format {FN_ID_STRING}. Identifiers of extended operators and signatures of the corresponding extended operations, as well as their suggested semantics, are registered with X/Open (see Appendix G).

The following three extended operations are currently defined:

’name’(<Wildcarded String>)
: The identifier for this operation is **name** ({FN_ID_STRING}). The argument to this operation is a wildcarded string. The operation returns TRUE if the name of the object matches the supplied wildcarded string.

’reftype’(%i)
: The identifier for this operation is **reftype** ({FN_ID_STRING}). The argument to this operation is an identifier. The operation returns TRUE if the reference type of the object is equal to the supplied identifier.

’addrtype’(%i)
: The identifier for this operation is **addrtype** ({FN_ID_STRING}). The argument to this operation is an identifier. The operation returns TRUE if any of the address types in the reference of the object is equal to the supplied identifier.

Support and exact semantics of extended operations are context-specific. If a context does not support an extended operation, or if the filter expression supplies the extended operation with either an incorrect number or type of arguments, the error [FN_E_SEARCH_INVALID_OP] is returned.[1]

Table 3-7 contains examples of filter expressions that contain extended operations.

Expression	Meaning
'name'('bill'*)	evaluates to TRUE if the name of the object starts with *bill*
%i(%a, %v)	evaluates to result of applying the specified operation to the supplied arguments.
(%a == %v) and 'name'('joe'*)	evaluates to TRUE if the specified attribute has the given value and if the name of the object starts with *joe*.

Table 3-7 Examples of Extended Operations in Search Filter Expressions

1. [FN_E_OPERATION_NOT_SUPPORTED] is returned when *fn_attr_ext_search*( ) is not supported.

3.8 Parsing Compound Names

Most applications treat names as opaque data and therefore, the majority of clients of the XFN interface will not need to parse compound names from specific naming systems. Some applications, however, such as browsers, need such capabilities. For these applications, XFN provides support in the form of syntax attributes, the XFN Standard Syntax Model, and the **FN_compound_name_t** object.

3.8.1 Syntax Attributes

Each context has an associated set of syntax-related attributes. The attribute *fn_syntax_type* (FN_ID_STRING format) identifies the naming syntax supported by the context. The value **standard** (ASCII attribute syntax) in the *fn_syntax_type* attribute specifies that the context supports the XFN standard syntax model that is by default supported by the **FN_compound_name_t** object.

Implementations may choose to support other syntax types in addition to, or in place of, the XFN standard syntax model, in which case, the value of the *fn_syntax_type* attribute would be set to an implementation-specific string, and different or additional syntax attributes will be in the set.

Syntax attributes of a context may be generated automatically by a context, in response to *fn_ctx_get_syntax_attrs*( ), or they may be created and updated using the attribute operations described in Section 3.3 on page 27. This is implementation-dependent.

3.8.2 XFN Standard Syntax Model

Each naming system in an XFN federation has a naming convention. XFN defines a standard model of expressing compound name syntax that covers a large number of specific name syntaxes and is expressed in terms of syntax properties of the naming convention. The model uses the attributes in Table 3-8 on page 53 to describe properties of the syntax. Unless otherwise qualified, the syntax attributes described in Table 3-8 have attribute identifiers that use the FN_ID_STRING format. This does not specify or restrict the use of other formats for identifiers of additional syntax attributes supported by specific implementations.

The XFN standard syntax attributes are interpreted according to the following rules:

1. In a string without quotes or escapes, any instance of the separator string delimits two atomic names.
2. A separator, quotation or escape string is escaped if preceded immediately (on the left) by the escape string.
3. A non-escaped begin-quote which precedes a component must be matched by a non-escaped end-quote at the end of the component. Quotes embedded in non-quoted names are treated as simple characters and do not need to be matched. An unmatched quotation fails with the status code [FN_E_ILLEGAL_NAME].
4. If there are multiple values for begin-quote and end-quote, a specific begin-quote value must be matched with its corresponding end-quote value.
5. When the separator appears between a (non-escaped) begin quote and the end quote, it is ignored.
6. When the separator is escaped, it is ignored. An escaped begin-quote or end-quote string is not treated as a quotation mark. An escaped escape string is not treated as an escape string.

7. A non-escaped escape string appearing within quotes is interpreted as an escape string. This can be used to embed an end-quote within a quoted string.
8. An escape string which precedes a character other than an escape string, a begin-quote or an end-quote is consumed (in other words, escaping a non-meta character returns the non-meta character itself).

After constructing a compound name from a string, the resulting component atoms have one level of escape strings and quotations interpreted and consumed.

Locale (code set, language, or territory) mismatches that occur during the construction of the compound name's string form are resolved in an implementation-dependent way. When an implementation discovers that a compound name has components with incompatible code sets, it returns the error code [FN_E_INCOMPATIBLE_CODE_SETS]. When an implementation discovers that a compound name has components with incompatible language or territory locale information, it returns the error code [FN_E_INCOMPATIBLE_LOCALES].

3.8.2.1 Compound Names

The **FN_compound_name_t** type is used to represent a compound name.

The **FN_compound_name_t** object has associated operations for applications to process compound names that conform to the XFN standard syntax model of expressing compound name syntax. Operations are provided to iterate over the list of atomic components of the name, modify the list, and compare two compound names.

An **FN_compound_name_t** object is constructed using the operation *fn_compound_name_from_attrset*( ), with arguments consisting of a string name and an attribute set that contains the attribute *fn_syntax_type* with the value **standard**.

Attribute Identifier	Attribute Value
fn_syntax_type	Its value is the ASCII string **standard** if the context supports the XFN standard syntax model. Its value is an implementation-specific value if another syntax model is supported.
fn_std_syntax_direction	Its value is an ASCII string, one of **left_to_right**, **right_to_left** or **flat**. This determines whether the order of components in a compound name string goes from left to right, right to left, or whether the namespace is flat (in other words, not hierarchical — all names are atomic).
fn_std_syntax_separator	Its value is the separator string for this name syntax. This attribute is required unless the *fn_std_syntax_direction* is **flat**.
fn_std_syntax_escape	If present, its value is the escape string for this name syntax.
fn_std_syntax_case_insensitive	If this attribute is present, it indicates that names that differ only in case are considered identical. If this attribute is absent, it indicates that case is significant. If a value is present, it is ignored.
fn_std_syntax_begin_quote1	If present, its value is one of the begin-quote strings for this syntax. If *fn_std_syntax_end_quote1* is absent but *fn_std_syntax_begin_quote1* is present, the quote-string specified in *fn_std_syntax_begin_quote1* is used as both the begin and end quote-strings. If *fn_std_syntax_end_quote1* is present but *fn_std_syntax_begin_quote1* is absent, the quote-string specified in *fn_std_syntax_end_quote1* is used as both the begin and end quote-strings.
fn_std_syntax_end_quote1	If present, its value is the end-quote string that matches the begin-quote string specified in *fn_std_syntax_begin_quote1*. If *fn_std_syntax_end_quote1* is absent but *fn_std_syntax_begin_quote1* is present, the quote-string specified in *fn_std_syntax_begin_quote1* is used as both the begin and end quote-strings. If *fn_std_syntax_end_quote1* is present but *fn_std_syntax_begin_quote1* is absent, the quote-string specified in *fn_std_syntax_end_quote1* is used as both the begin and end quote-strings.
fn_std_syntax_begin_quote2	If present, its value is one of the begin-quote strings for this syntax. If *fn_std_syntax_end_quote2* is absent but *fn_std_syntax_begin_quote2* is present, the quote-string specified in *fn_std_syntax_begin_quote2* is used as both the begin and end quote-strings. If *fn_std_syntax_end_quote2* is present but *fn_std_syntax_begin_quote2* is absent, the quote-string specified in *fn_std_syntax_end_quote2* is used as both the begin and end quote-strings.

Attribute Identifier	Attribute Value
fn_std_syntax_end_quote2	If present, its value is the end-quote string that matches the begin-quote string specified in *fn_std_syntax_begin_quote2*. If *fn_std_syntax_end_quote2* is absent but *fn_std_syntax_begin_quote2* is present, the quote-string specified in *fn_std_syntax_begin_quote2* is used as both the begin and end quote-strings. If *fn_std_syntax_end_quote2* is present but *fn_std_syntax_begin_quote2* is absent, the quote-string specified in *fn_std_syntax_end_quote2* is used as both the begin and end quote-strings.
fn_std_syntax_ava_separator	If present, its value is the attribute value assertion separator string for this syntax.
fn_std_syntax_typeval_separator	If present, its value is the attribute type-value separator string for this syntax.

fn_std_syntax_locales	If this attribute is not present, or if the value is empty, the only locale supported by the context is the ‘‘C’’ locale. The ‘‘C’’ locale’s repertoire of characters includes those characters defined by ISO 646. Interoperability as well as portability can be guaranteed within the ‘‘C’’ locale by limiting the use to the repertiore of characters which are defined in ISO 646. This is the repertoire of characters defined for the Portable Character Set (PCS) in Section 4.1.2 on page 56.
	
	If present, the attribute’s value identifies the locales of string representations that can be supported by the context. The value consists of an array of structures. Each element in the array contains:
	`unsigned long code_set,`
	`unsigned long lang_terr`
	Arguments `code_set` and `lang_terr` together identify a locale. The values for the code sets are defined in the OSF code set registry currently defined in DCE RFC 40.1 (OSF Character and Code Set Registry). This registry is being extended to include language/territory registrations.

Table 3-8 XFN Standard Syntax Model Attributes

Chapter 4

XFN Composite Names

This chapter describes the composite name string syntax and the resolution techniques for composite names.

4.1 Composite Name String Syntax

An *XFN composite name* consists of an ordered list of zero or more components. Each component is a string name from the namespace of a single naming system and uses the naming syntax of that naming system. A component may be an atomic or a compound name from that namespace. XFN does not specify any syntax for regular expressions at the composite name level. However, an individual naming system may allow a component to contain expressions (for example, wildcard characters).

XFN defines an abstract data type, **FN_composite_name_t**, for representing the structural form of a composite name. XFN also defines a standard string form for composite names. This form is the concatenation of the components of a composite name from left to right with the *XFN component separator* character ('/') separating each component.

The XFN client interface includes operations to convert a composite name from its string form to its structural form, and vice versa. This section describes the syntax of XFN composite names and the rules for converting the string form of XFN composite names to and from its structural form.

4.1.1 Encoding of XFN Composite Name Strings

Special characters used in the XFN composite name syntax, such as the component separator or escape characters, have the same encoding as they would in ISO 646.

The minimum requirement for all XFN implementations is to support the portable representation of ISO 646 (same encoding as ASCII) for communication of name strings. All other representations are optional. See Section 2.5 on page 17.

All characters of the string form of an XFN composite name use a single encoding. This does not preclude component names of a composite name in its structural form from having different encodings. Locale (code set, language, or territory) mismatches that occur during the process of converting a composite name structure to its string form are resolved in an implementation-dependent way. When an implementation discovers that a composite name has components with incompatible locales, it returns an appropriate error:

[FN_E_INCOMPATIBLE_CODE_SETS]

or

[FN_E_INCOMPATIBLE_LOCALES].

4.1.2 Backus-Naur Form (BNF) of XFN Composite Names

This section defines the standard string form of XFN composite names in BNF. Note that all the characters of the string representation of one name must uniformly use the same encoding and locale information.

The notations used are as follows:

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Non-terminal element
""	Literal expression
*	The preceding syntactic unit can appear 0 or more times.
+	The preceding syntactic unit can appear 1 or more times.
{}	The enclosed syntactic units are grouped as a single syntactic unit (can be nested).

The XFN composite name syntax in BNF is as follows.

```
NULL ::=            // Empty set

<PCS> ::=           // Portable Character Set
                        // The set consists of the glyphs:
                        //   !"#$%&'()*+,-./0123456789:;<=>?
                        //  @ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
                        //  `abcdefghijklmnopqrstuvwxyz{|}~

<CharSet> ::=       <PCS>
                    | Characters from the repertoire of a string representation

<EscapeChar> ::=    \

<ComponentSep> ::=  /

<Quote1> ::=        "

<Quote2> ::=        ´

<MetaChar> ::=      <EscapeChar> | <ComponentSep>

<SimpleChar> ::=    // any character from <CharSet> with <ComponentSep>, <Quote1>,
                    // and <Quote2> excluded. An <EscapeChar> <MetaChar>, or
                    // <EscapeChar> <Quote1>, or <EscapeChar> <Quote2> is
                    // substituted by the corresponding unescaped character and
                    // is equivalent to a <SimpleChar>.

<Component> ::=     <SimpleChar>*
                    | <SimpleChar>+ {<Quote1> | <Quote2> | <SimpleChar>}*
                    | <Quote1> <CharSet>* {<EscapeChar><Quote1>}* <CharSet>*
                    <Quote1>
                        // <CharSet> must not contain unescaped <Quote1>
                        // (note that <Quote2> can appear unescaped)
                    | <Quote2> <CharSet>* {<EscapeChar><Quote2>}* <CharSet>*
                    <Quote2>
                        // <CharSet> must not contain unescaped <Quote2>
                        // (note that <Quote1> can appear unescaped)
```

<CompositeName> ::= NULL
| <Component> {<ComponentSep> <Component>}*

4.1.3 Decomposing a Composite Name String

This section defines the rules for converting the string representation of a composite name into the internal composite name object (**FN_composite_name_t**). This defines the semantics of the function *fn_composite_name_from_string*( ).

A composite name is decomposed into an ordered set of components (<Component>). Each component represents:

- a compound name

  or:

- a single atomic name of a compound name if the compound name's syntax uses the XFN component separator (''/'') as a separator for its atomic parts and the compound name is not quoted.

The following are the rules for parsing a composite name:

1. Any <ComponentSep> character that is neither escaped nor enclosed in quoted strings is considered to be a component separator.
2. Any string enclosed by component separators is a component (<Component>).
3. A composite name is parsed and decomposed into components from left to right:
   - The first component is the string preceding the first occurrence of a component separator.
   - Empty components are processed as follows:
     — A leading component separator (the composite name begins with a component separator) means a leading null component.
     — A trailing component separator (the composite name ends with a component separator) means a trailing null component.
     — Two consecutive component separators mean a null component.
   - The name string that immediately follows the last component separator of the composite name is the terminal component.
4. A component string is evaluated from left to right and converted into its standard form according to the following rules:
   - A component string is considered to be quoted if it is enclosed in a pair of matching unescaped quote characters (either a <Quote1> or a <Quote2> pair). The quoted string must represent the full component; that is, a begin quote must immediately be preceded by a component separator or no character, and the end quote must immediately be followed by a component separator or no character.
   - If a component does not contain a valid begin quote (a <Quote1> or <Quote2> immediately preceded by either a component separator or no character), any occurrence of <Quote1> or <Quote2> within that component is not treated as a quote but is treated just as any other <SimpleChar>.
   - An unmatched begin quote (missing or misplaced end quote) fails with an [FN_E_ILLEGAL_NAME] status.

- Quotes are considered to be escaped in quoted strings if a matching quote character is preceded immediately by the unescaped <EscapeChar>.
- Quoted components are resolved by eliminating the quote characters from the component name and substituting possibly escaped quotes by simple quote characters. <MetaChar>s and the not matching quote characters enclosed in quoted strings are treated just as any other <SimpleChar>.
- Any of the defined meta characters (<ComponentSep> and <EscapeChar>) or quote characters (<Quote1> and <Quote2>) is considered to be escaped in an unquoted component name string if preceded immediately by the unescaped <EscapeChar> (for instance, the sequence <EscapeChar> <EscapeChar> <ComponentSep> denotes an escaped <EscapeChar> but an unescaped <ComponentSep>).
- Any occurrence of escaped <MetaChar>, or escaped <Quote1> or escaped <Quote2> in unquoted components is substituted by the corresponding unescaped character and is treated as a <SimpleChar>.
- No substitution is done for <EscapeChar> <SimpleChar>. <EscapeChar> <SimpleChar> simply maps to <EscapeChar> <SimpleChar>.

Table 4-1 contains some examples of how the string form of a composite name are decomposed into components in accordance with the BNF syntax of composite names.

String Form	Components in FN_composite_name_t
a	a
a/b/c	a, b, c
a/	a, ""
/a	"", a
a//	a, "", ""
a//b	a, "", b
""	""
/	"", ""
//	"", "", ""
"a/b/c"/d	a/b/c, d
"a.b.c"/d	a.b.c, d
a.b.c/d	a.b.c, d
a"b/c	a"b, c
a\"b	a"b
a'b/c	a'b, c
"a/b/c	illegal name
\"a/b/c	"a, b, c
a\b\c/d	a\b\c, d
a\b\/c	a\b/c
"a\"b"/c	a"b, c
'"a/b/c"'	"a/b/c"
'a\/b'/c	a\/b, c
a\\b/c	a\b, c
a/\"b	a, "b

Table 4-1 Examples of String & Structural Forms of XFN Composite Names

4.1.4 Composing a Composite Name String

This section describes the rules used for converting the composite name object (**FN_composite_name_t**), representing an ordered set of components, into the composite name string representation. This defines the semantics of the operation *fn_string_from_composite_name*( ).

1. The components are added to the composite name string in left to right order (in other words, rightmost is the tail).
2. Successive components are separated by the component separator (<ComponentSep>).
3. Empty components are handled in the following way:
   - A leading empty component is represented by a leading <ComponentSep>.
   - A trailing empty component is represented by a trailing <ComponentSep>>
   - An empty component occurring within a composite name is represented by two consecutive <ComponentSep>s.
4. A composite name denoting a single non-empty component does not contain any unescaped component separator.
5. Any occurrence of <ComponentSep> in a component is escaped by inserting <EscapeChar> immediately preceding <ComponentSep>.
6. If the first character of a component is either <Quote1> or <Quote2>, it will be escaped by inserting <EscapeChar> immediately preceding the quote.
7. Any occurrence of <EscapeChar> before <ComponentSep> in a component is escaped by inserting <EscapeChar> immediately preceding the <EscapeChar>.
8. Any occurrence of <EscapeChar> as the first character of a component with <Quote1> or <Quote2> as the second character in a component is escaped by inserting <EscapeChar> immediately preceding the <EscapeChar>. Subsequent <EscapeChar> occurring before any matching quote character is also escaped by inserting <EscapeChar> immediately preceding the <EscapeChar>.

4.2 Composite Names and Naming System Boundaries

The correspondence between component separators and naming system boundaries may not be one to one if a composite name contains names from naming systems that use the same character as the XFN component separator to separate their atomic names. A component of a composite name may represent an atomic name from a hierarchical naming system that uses the XFN component separator, or a compound name. A single component cannot straddle more than one naming system.

4.2.1 Strong Separation

An XFN context that treats the XFN component separator as a naming system boundary supports *strong separation*.

Support for strong separation is a property of a context. A context that supports strong separation expects to receive the name that it is going to resolve entirely in one component of the composite name structure. When a composite name is supplied to such a context, it consumes the leading component of the name; any remaining components are left to be resolved by subordinate naming systems.

An XFN context with a name syntax that is either flat or hierarchical, but does not use the XFN component separator as its atomic separator, supports strong separation. An XFN context with a name syntax that is hierarchical and uses the XFN component separator as its atomic separator supports strong separation if it requires its atomic separator to be quoted or escaped whenever it appears in compound names within composite names.

For example, assume **A/B**, and **A=B/C=D/E=F** are compound names from naming systems that use the XFN component separator as their atomic component separator and compose atomic components left to right. Assume also that **M** is an atomic name from a naming system with a hierarchical left-to-right name syntax that uses the XFN component separator as its atomic component separator. Assume also that **B.A** is a compound name from a hierarchical naming system with a dot-separated, right-to-left name syntax. Finally, assume that **V** is a name from another naming system. The following are then examples of composite names intended for a context that supports strong separation.

```
A\/B/V

"A=B/C=D/E=F"/V

M/V

B.A/V
```

An XFN context that supports strong separation can be federated with arbitrary subordinate naming systems, with no restriction on the name syntax of subordinate naming systems. Naming systems federated in this way need not be changed as new naming systems (regardless of their name syntax) are added to the federation.

4.2.2 Weak Separation

An XFN context that does not always treat the XFN component separator as a naming system boundary supports *weak separation*. This arises when the component naming system associated with the context uses the same character as the XFN component separator as its atomic component separator, *and* the context allows its atomic separator to appear unescaped and unquoted in its compound names when they occur in composite names. This means that an XFN component separator may not necessarily signify a naming system boundary. XFN component separators that appear *within* an atomic component must be escaped or quoted.

Support for weak separation is a property of a context. A context that supports weak separation expects to receive its atomic names in separate components of the composite name structure. When a composite name is supplied to a context that supports weak separation, the context consumes the leading components of the name (and treats them as atomic components); any remaining components are resolved by subordinate naming systems. The number of components consumed is determined either syntactically or dynamically as described below.

For example, assume **A/B** and **A=B/C=D/E=F** are compound names from naming systems that use the XFN component separator as their atomic component separator and compose atomic components left to right. Also, assume that **M** is an atomic name from a naming system with a hierarchical left-to-right name syntax that uses the XFN component separator as its atomic component separator. Finally, assume that **V** is a name from another naming system. The following are then examples of composite names intended for a context that supports weak separation:

```
A/B/V

A=B/C=D/E=F/V

M/V
```

CDS names and X.500 names are examples of names that use the XFN component separator as their atomic name separator. See Section B.2 on page 230 and Section B.3 on page 240 for details of how XFN context implementations for these naming systems should deal with this issue.

4.2.2.1 Conditions for Supporting Weak Separation

XFN contexts of naming systems that use the same character as the XFN component separator to separate their atomic names may support weak separation if and only if their atomic component ordering is left to right, and *at least one* of the following conditions apply:

- The naming system is a terminal naming system; that is, its component name(s) always appear at the end of a composite name.
- The naming system is non-terminal and the context can do syntax-specific discovery of the boundary between its naming system and subordinate naming systems.
- The naming system is non-terminal and the context can return the unresolved remaining components.

Naming systems that use the same character as the XFN component separator as their atomic separator, and which do not meet the above conditions must provide context implementations that support strong separation. This means that occurrences of atomic separators must be quoted or escaped when they appear in compound names within composite names.

It may not always be possible for an XFN context that supports weak separation using only syntax-specific discovery of its naming system boundary to be federated with arbitrary subordinate naming systems. This is because if the subordinate naming system has a naming

syntax that is indistinguishable from that of the superior naming system, the superior naming system would not be able to identify the naming system boundary.

4.2.3 Strong and Weak Separation Support in Contexts

An XFN context may support either strong separation or weak separation. An XFN context is not required to support both forms of separation, though some may choose to do so. A context that supports strong separation can coexist with one that supports weak separation in the same federation.

A context that supports strong separation should expect to receive its component name in one component of the composite name structure. A context that supports weak separation should expect to receive its atomic names in separate components of the composite name structure. A context that supports both strong and weak separation must be prepared to receive its component name in either one component or multiple components, and must be able to disambiguate between a component that identifies a compound name and a component that identifies an atomic name.

4.3 Composite Name Resolution Techniques

A composite name consists of names from multiple naming systems. Composite name resolution combines resolution in each component naming system and resolution across federated naming system boundaries.

A *next naming system pointer* is the XFN reference of an XFN context in which composite name components from subordinate naming systems are to be resolved.

This section describes two implementation techniques for composite name resolution across a naming system boundary. One technique uses an explicitly named next naming system pointer — *junction* — to resolve across a naming system boundary. The other uses an implicit next naming system pointer to resolve across a naming system boundary.

A *junction* is an atomic name that is bound to a next naming system pointer. It is a terminal name in the superior naming system. There is no limit on the number of junctions bound in a single context, except that imposed by the context. A context may reserve certain names for use as junctions, or have other policies for selecting names for use as junction. The conventions used for identifying junctions and their references are context-specific.

When a context does not want to use part of its namespace for junctions, it uses *implicit* next naming system pointers for federating subordinate naming systems. An implicit next naming system pointer is named using the XFN component separator. For example, the name **B.A/** names the implicit next naming system pointer of **B.A**. Each context can have one implicit next naming system pointer.

Naming systems that implement either technique may coexist in a federation. A naming system that supports composite name resolution using junctions can be federated with one that supports implicit next naming system pointers, and vice versa.

4.3.1 Resolution Using Implicit Next Naming System Pointers

4.3.1.1 Strong Separation and Implicit Next Naming System Pointers

An XFN context that supports strong separation and resolves composite names using an implicit next naming system pointer consumes the first component of the composite name supplied to it. Any remaining components are resolved in the context pointed to by the implicit next naming system pointer of the first component.

For example, the composite name **A**:**B/E.D** (Figure 4-1) has two components, **A**:**B** (a left-to-right colon-separated name) and **E.D** (a right-to-left dot-separated name). Resolution in the first naming system starts with **A**, then **B**, and then the implicit next naming system pointer associated with **B**, which takes resolution into a new naming system, where resolution continues on **D** and then **E**.

4.3.1.2 Weak Separation and Implicit Next Naming System Pointers

An XFN context that supports weak separation and implicit next naming system pointers in its implementation needs to distinguish the use of the XFN component separator character as an XFN component separator or an atomic separator. This means that such a context needs to know when to exit the current (native) naming system and follow the implicit next naming system pointer. This can be achieved using a static, syntactic policy or a dynamic, resolution policy.

With the syntactic policy, a context syntactically discovers where the boundary between its naming system and the subordinate naming system lies. This may impose certain restrictions on the syntax of subordinate naming systems. Subordinate naming systems must not permit as valid top level names names that are syntactically indistinguishable from names allowed in the

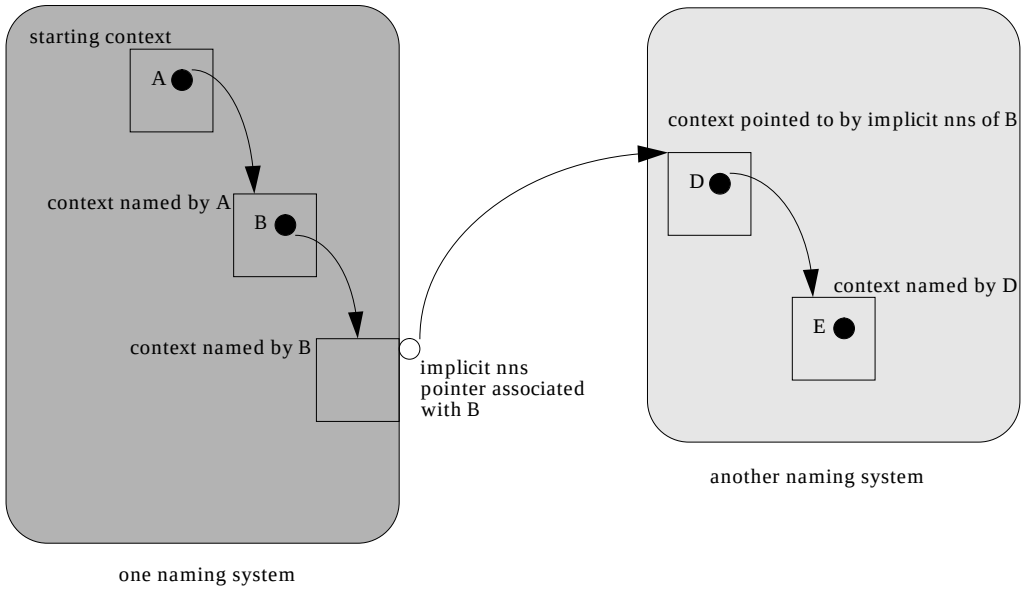

Figure 4-1 Resolution Using Implicit Next Naming System Pointer

superior naming system. For example, assume the superior naming system has a name syntax whose distinguishing feature is that each atomic part must have an equal character ('='). The superior naming system might impose as a policy that subordinate naming systems must not have top level names that have an equal character in them. Resolution in the superior naming system continues until all leading components of the supplied composite name fitting the syntactic rule are consumed. Any remaining components are resolved in the context of the implicit next naming system pointer of the last component fitting the syntactic rule.

If a context is not able to syntactically differentiate between atomic components and composite name components, or does not want to impose any syntactic restrictions, it may be able to determine the naming system boundary at runtime during resolution. The policy is to continue resolution in the current naming system until resolution fails, at which point, the implicit next naming system pointer associated with the last context at which resolution succeeded is used to continue the resolution. A conflict arises if the same atomic name is bound both in the last context *and* the context pointed to by the last context's implicit next naming system pointer. In this case, the binding in the last context takes precedence. Note that this way of supporting weak separation requires the context to have the capability of returning remaining unresolved parts of a given name. Figure 4-2 illustrates this conflict and shows how resolution would proceed for a context that supports weak separation and the implicit next naming system pointer when there is a conflict. The name **A/B/D** refers to the object **D** in the first naming system, not the one reached using the implicit next naming system pointer of **B**. This ambiguity is introduced by the fact that the context supports weak separation. With strong separation, both can be named precisely; with weak separation, the component separator can mean continue resolution either in the current naming system or in the subordinate naming system.

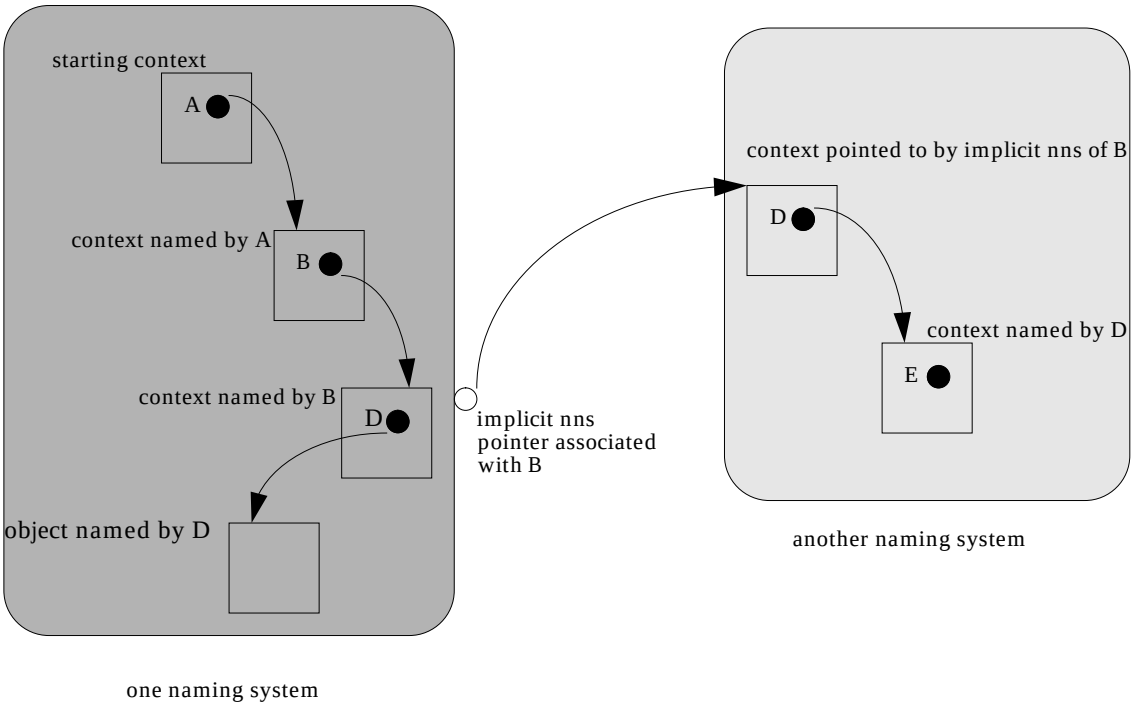

Figure 4-2 Conflict with Weak Separation and Implicit Next Naming

Consider another example in which **A/B** and **A=B/C=D** are compound names from naming systems with hierarchical left-to-right name syntax that use the XFN component separator as their atomic component separator. Assume also that **F** is an atomic name from another naming system. The following are then examples of names intended for contexts that support weak separation and implicit next naming system pointers:

```
A=B/C=D/F

A/B/F
```

The first name is intended for a context that determines how many leading components to consume based on a syntactic rule of checking for equal characters ('=') in the component name. The second name is intended for a context that determines when to follow the implicit next naming system pointer dynamically during resolution. The context attempts to resolve **A/B/F** and returns **F** as the remaining component. **F** is then resolved in the context pointed to by the implicit next naming system pointer of **B**.

4.3.1.3 Context Requirements

The implicit next naming system pointer is named using the XFN component separator.

As with any other XFN contexts, the context in which the implicit next naming system pointer is bound needs only to support the *fn_ctx_lookup*( ) operation and the resolution phase of all other operations in the base context and attribute interfaces when the composite name argument is the XFN component separator (see Section 3.2.2 on page 20).

An enumeration of the context in which the implicit next naming system pointer is bound does not include the implicit next naming system pointer.

The creation of the binding for the implicit next naming system pointer is implementation-dependent. There is no requirement that the binding for the next naming system pointer be stored in the same way that other bindings in the same context are stored. Some contexts may support the binding to be set using the *fn_ctx_bind*( ) operation. Another context may require that the binding be set using operations native to the naming system. Some other context may use the attribute operations to store and manipulate such bindings. While yet another context may generate the reference during resolution (for example, this may be the case if the implicit next naming system pointer is hard wired to a specific value based on the individual naming system's policy).

For contexts that support weak separation and implicit next naming system pointers, update operations to the context pointed to by implicit next naming system pointers, such as *fn_ctx_bind*( ) and *fn_ctx_unbind*( ), should be done in separate steps in order to have the desired effect:

1. Resolve the name of the target context explicitly using the *fn_ctx_lookup*( ) operation, supplying it with the name of the implicit next naming system pointer's context (by use of the trailing XFN component separator).
2. Get a handle to the target context by passing the reference returned from step 1 to the *fn_ctx_handle_from_ref*( ) operation.
3. Perform the intended operation (*fn_ctx_bind*( ) or some other operation) on the target context using the atomic name of the target object.

Trying to perform the operation in one step (by using the full composite name of the object) may not necessarily have the intended effect because for contexts that support weak separation and implicit next naming system pointers, an XFN component separator occurring within a composite name does not necessarily denote a naming system boundary.

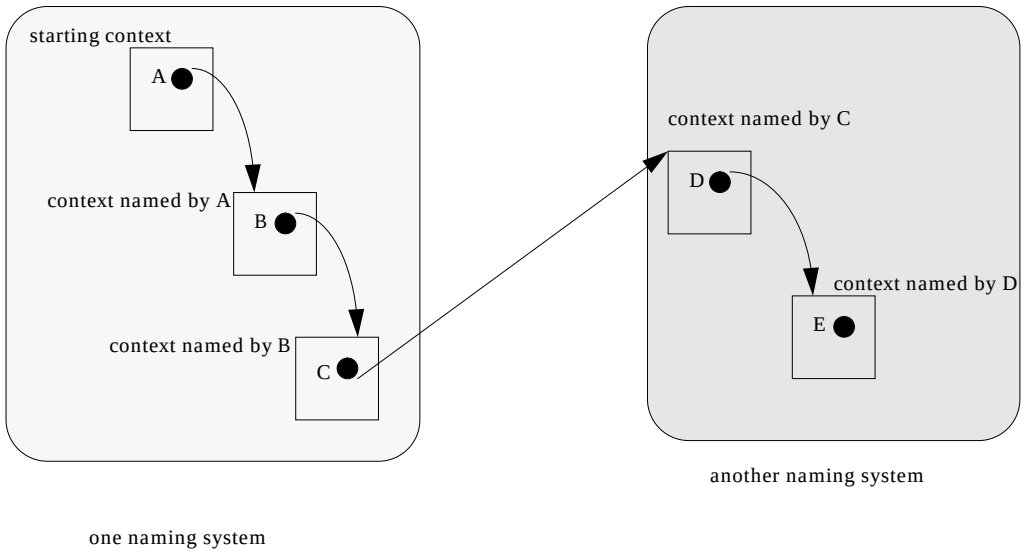

Figure 4-3 Example of Resolution Using Junction

Using the example in Figure 4-2, *fn_ctx_unbind*( ) on the name **A/B/D** results in the removal of **D** from the naming system on the left, while doing a lookup first on **A/B/**, followed by an *fn_ctx_unbind*( ) of **D** in the resulting context results in the removal of **D** from the naming system on the right.

4.3.2 Resolution Using Junctions

4.3.2.1 Strong Separation and Junctions

A context that supports strong separation and junctions consumes the first component of the composite name supplied to it. The last atomic name of the first component must be a junction. Any remaining components are resolved in the context named by the junction.

For example, the composite name **A:B:C/E.D** (Figure 4-3) has two components, **A:B:C** (a left-to-right colon-separated name) and **E.D** (a right-to-left dot-separated name). **C** is a junction. Resolution in the first naming system starts with **A**, then **B**, then **C**, which resolves to a context in a new naming system, where resolution continues on **D** and then **E**.

4.3.2.2 Weak Separation and Junctions

An XFN context that supports weak separation and uses junctions resolves a composite name by consuming leading components until a junction is reached, at which point, resolution of any remaining components is continued in the context bound to the junction. Determination of whether a component is a junction can be done statically using a syntactic policy or dynamically during resolution. For the latter, the context must be able to return the remaining unresolved components.

For example, assume that **A/B/J** and **A=B/C=D/E=J** are compound names from naming systems with hierarchical left-to-right name syntax that use the XFN component separator as their atomic component separator. Assume also that **C** is a name from another naming system. The following are then examples of names intended for contexts that support weak separation and junctions:

```
A/B/J/C

A=B/C=D/E=J/C
```

Given the first name, the starting context resolves the components, **A**, **B**, and **J** and discovers that **J** is a junction. The remaining component **C** is resolved in the context named by the **J** junction. In the second example, the starting context might syntactically determine that the first three components are to be resolved in its naming system. If the component **E=J** is a junction, the last component, **C** is resolved in the context named by **E=J**.

4.3.2.3 Context Requirements for Supporting Junctions

As with any other XFN contexts, the context in which the junction is bound needs only to support the *fn_ctx_lookup*( ) operation and the resolution phase of all other operations in the base context and attribute interfaces for the junction (see Section 3.2.2 on page 20).

The creation of the binding for a junction is implementation-dependent. Some contexts may require that the binding be manipulated using the context operations *fn_ctx_bind*( ) and *fn_ctx_unbind*( ). Others may require that the binding be set using operations native to the naming system.

4.3.3 Summary

The following table provides examples that summarises the four combinations of using implicit next naming system pointer and junctions techniques for contexts supporting strong and weak separation.

These examples assume that **B.A** (dot-separated, right-to-left name syntax) and **A/B** (slash-separated, left-to-right name syntax) are compound names. **J** is a junction.

	Strong Separation	**Weak Separation**
T{Implicit Next Naming System Pointer T}	B.A/X	A/B/X
Junction	J.B.A/X	A/B/J/X

4.4 Composite Name Resolution Involving Links

A link affects name resolution in the following way. Suppose *lname* is a link bound to the atomic name *aname* in the context *ctx*. If at some point, resolution of a composite name *cname* reaches the context *ctx* and the next atomic name is *aname*, resolution of *aname* results in the resolution of the link name *lname*. This is termed *following the link*. If the first composite name component of the link *lname* is the atomic name ".", the remaining components of *lname* are resolved relative to *ctx*; otherwise, *lname* is resolved from the Initial Context. The resolution of any remaining portion of the name *cname* proceeds from the reference that results by resolving *lname*.

The link name may itself cause resolution to resolve through other links. This gives rise to the possibility of a cycle of links whose resolution could not terminate normally. As a simple means to avoid such non-terminating resolutions, implementations may define limits on the number of XFN links that may be resolved in any single operation invoked by the caller.

Chapter 5

XFN Policies

Computing environments for enterprises are becoming world-wide in scope and encompass a wide range of services. Applications will be increasingly expected to reflect to the user the scope and range by enabling access anywhere in this environment. The goals of XFN are to promote the portability of applications and global interoperability. As a first step towards that goal, XFN specifies a minimal set of essential naming policies for naming objects across enterprises.

An XFN implementation is not required to support all the policies defined in this chapter, but for the XFN policies that are supported, their behaviour must comply with the behaviour specified herein. An XFN implementation is free to add policies that are not specified by, and which do not conflict with, XFN policies.

5.1 Terminology

Global naming service

A global naming service is a naming service that has world-wide scope. Internet DNS and X.500 are examples of global naming services. The types of entities named at this global level are typically countries, states, provinces, cities, companies, universities, institutions, and government departments and ministries. These entities are referred to as *enterprises*. Specific global naming services might name other types of entities. For example, in addition to enterprises, DNS is also used to name hosts and electronic mailboxes. X.500 is also used to name users and organizational units within enterprises.

Enterprise-level naming service

Enterprise-level naming services are used to name entities within an enterprise. Within an enterprise, there are naming services that provide contexts for naming common entities such as organizational units, physical sites, human users and computers. Enterprise-level naming services are below the global naming services. Global naming services provide contexts in which the root contexts of enterprise-level naming systems can be bound.

Application-level naming service

Application-level naming services are incorporated in applications offering services such as file service, mail service, print service, and so on. Application-level naming services are below enterprise-level naming services. The enterprise-level naming services provide contexts in which contexts of application-level naming systems can be bound.

5.2 Policy Overview

Global

XFN policy specifies that DNS and X.500 are global naming services that are used to name enterprises.

XFN policy does not preclude the use of other global naming services to name enterprises. Furthermore, XFN policy does not restrict the use of DNS or X.500 to provide naming service for entities other than enterprises.

Enterprise

XFN recommends optional policies for the enterprise namespace. These are described in Appendix D.

Applications

Naming within applications is left to individual applications or groups of related applications and not specified by XFN.

Initial Context

Each XFN client has an *Initial Context* that provides a starting point for resolution of composite names. XFN policy specifies *names and bindings* present in the Initial Context of an XFN client.

The bindings for the global context are specified in this chapter. Optional bindings for the enterprise-related contexts are specified in Appendix D.

Not all of the specified names are required to appear in all XFN Initial Contexts, but when they do appear, they must have the specified bindings. Implementations are free to have other names in the Initial Context but any such extensions are not part of XFN policies.

5.3 Naming Enterprises Using Global Naming Services

XFN policy specifies that the Internet Domain Naming Service (DNS) and X.500 provide global-level naming service for naming enterprises. XFN policy does not restrict the type of information that these global naming services can support in addition to naming enterprises.

Enterprise-level naming services are below the global naming services. The boundary of the namespace of the global naming system and that of the enterprise systems varies depending on many factors such as performance, granularity, update requirements, and flexibility of the global system. The boundary may differ from configuration to configuration and is not defined by XFN policy. Specifically, XFN policy does not preclude installations from using DNS and X.500 to offer enterprise-level or application-level naming services. For example, in one configuration, an X.500 system might support the organizational unit and user namespaces, with lower level namespaces supported by a local network naming system. In another configuration, an enterprise might choose to use DNS to support naming other enterprises and use ONC/NIS+ or DCE/CDS for all naming within the enterprise.

5.3.1 Bindings in the Initial Context for the Global Context

Namespace Identifier	Binding
‘‘`...`’’ and ‘‘`/...`’’	The global context for naming DNS or X.500 names.
`_dns`	The root of the DNS namespace.
`_x500`	The root of the X.500 namespace.

Table 5-1 Global Bindings in the Initial Context

Table 5-1 lists the bindings in the Initial Context for global names.

XFN policy specifies that the atomic name ‘‘`...`’’ appears in the Initial Context of every XFN client if the client has access to a global naming service. The atomic name ‘‘`...`’’ is bound to a context from which global names can be resolved.

Global names can be either fully-qualified Internet Domain Names or X.500 distinguished names; support for other types of global names is not specified by XFN. Internet domain names are to appear in the syntax specified by Internet RFC 1035. X.500 names are to appear in the syntax determined by the X/Open DCE Directory.

For example, `.../Wiz.COM` specifies a name to be resolved by DNS, whereas, `.../c=us/o=Wiz` specifies a name to be resolved by X.500.

XFN policy also specifies that the names ‘‘`...`’’ and ‘‘`/...`’’ are equivalent when resolved in the Initial Context.[2] An Initial Context implementation must support the resolution of both atomic names, ‘‘`...`’’ and ‘‘`/...`’’, to the same context, or neither of the two atomic names.

For example, the names `/.../c=us/o=Wiz` and `.../c=us/o=Wiz` resolved in the Initial Context must refer to the same object.

XFN policy specifies that the atomic names `_dns` and `_x500` name the root context in the DNS and X.500 namespaces, respectively, when supplied to the Initial Context. These names are

2. The motivation for supporting ‘‘`/...`’’ in the Initial Context is for compatibility with DCE names.

useful for browsing these global namespaces and for explicitly specifying the target global naming service.

For example, `_dns/Wiz.COM` specifies a name to be resolved by DNS while `_x500/c=us/o=Wiz` specifies a name to be resolved by X.500.

The atomic names ‘‘`...`’’, ‘‘`/...`’’, ‘‘`_dns`’’ and ‘‘`_x500`’’ are encoded using ISO 646 (same encoding as ASCII). An XFN implementation might support additional encodings for these names, or additional names that convey the same meaning as these. Such support is not part of XFN policies and may not necessarily be supported by other XFN implementations. This has implications on portability and interoperability of applications and systems that use them.

5.3.2 Support for Other Global Naming Services

XFN policy does not preclude the use of other global naming services for naming enterprises. However, XFN policy does not guarantee that arbitrary global naming services can be used without changes to the current policy — all global names must appear after ‘‘`...`’’ in the Initial Context and syntax-based heuristics are used to select the target global naming service. The rate and number that global naming services will emerge make this technique feasible.

Chapter 6

Reference Manual Pages

This chapter describes the XFN context and attribute interfaces in detail. Also described are the objects used in operations in the XFN context and attribute interfaces. There is one reference page per object type containing a description of the object and operations used to manipulate the object.

The interfaces described in this chapter follow the naming conventions for symbols described in Section 3.1 on page 19.

Reference pages appear in alphabetical order.

NAME

FN_attribute_t — an XFN attribute

SYNOPSIS

```
#include <xfn/xfn.h>

FN_attribute_t *fn_attribute_create(
    const FN_identifier_t *attribute_id,
    const FN_identifier_t *attribute_syntax);

void fn_attribute_destroy(FN_attribute_t *attr);

FN_attribute_t *fn_attribute_copy(const FN_attribute_t *attr);

FN_attribute_t *fn_attribute_assign(
    FN_attribute_t *dst,
    const FN_attribute_t *src);

const FN_identifier_t *fn_attribute_identifier(
    const FN_attribute_t *attr);

const FN_identifier_t *fn_attribute_syntax(const FN_attribute_t *attr);

unsigned int fn_attribute_valuecount(const FN_attribute_t *attr);

const FN_attrvalue_t *fn_attribute_first(
    const FN_attribute_t *attr,
    void* *iter_pos);

const FN_attrvalue_t *fn_attribute_next(
    const FN_attribute_t *attr,
    void* *iter_pos);

int fn_attribute_add(
    FN_attribute_t *attr,
    const FN_attrvalue_t *attribute_value,
    unsigned int exclusive);

int FN_attribute_remove(
    FN_attribute_t *attr,
    const FN_attrvalue_t *attribute_value);
```

DESCRIPTION

An attribute has an attribute identifier, a syntax, and a set of distinct values. Each value is a sequence of octets. The operations associated with objects of type **FN_attribute_t** allow the construction, destruction, and manipulation of an attribute and its value set.

The attribute identifier and its syntax are specified using an **FN_identifier_t**. *fn_attribute_create*( ) creates a new attribute object with the given identifier and syntax, and an empty set of values. *fn_attribute_destroy*( ) releases the storage associated with *attr*; if *attr* is NULL, no action is taken. *fn_attribute_copy*( ) returns a copy of the object pointed to by *attr*. *fn_attribute_assign*( ) makes a copy of the attribute object pointed to by *src* and assigns it to *dst*, releasing any old contents of *dst*. A pointer to the same object as *dst* is returned.

fn_attribute_identifier( ) returns the attribute identifier of *attr*. *fn_attribute_syntax*( ) returns the attribute syntax of *attr*. *fn_attribute_valuecount*( ) returns the number of attribute values in *attr*.

fn_attribute_first( ) and *fn_attribute_next*( ) are used to enumerate the values of an attribute. Enumeration of the values of an attribute may return the values in any order. *fn_attribute_first*( ) returns an attribute value from *attr* and sets the iteration marker *iter_pos*. Subsequent calls to *fn_attribute_next*( ) return the next attribute value identified by *iter_pos* and advance *iter_pos*. Adding or removing values from an attribute invalidates any iteration markers that the caller holds.

fn_attribute_add( ) adds a new value *attribute_value* to *attr*. The operation succeeds (but no change is made) if *attribute_value* is already in *attr* and *exclusive* is zero; the operation fails if *attribute_value* is already in *attr* and *exclusive* is non-zero. *fn_attribute_remove*( ) removes *attribute_value* from *attr*. The operation succeeds even if *attribute_value* is not amongst *attr*'s values.

RETURN VALUE

fn_attribute_first( ) returns 0 if the attribute contains no values. *fn_attribute_next*( ) returns 0 if there are no more values to be returned in the attribute (as identified by the iteration marker) or if the iteration marker is invalid.

fn_attribute_add( ) and *fn_attribute_remove*( ) return 1 if the operation succeeds, 0 if it fails.

APPLICATION USAGE

Manipulation of attributes using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to attributes in the underlying naming system can only be effected through the use of the interfaces described in the reference manual page for XFN_attribute_operations.

SEE ALSO

FN_attrvalue_t, **FN_attrset_t**, **FN_identifier_t**, *fn_attr_get*( ), *fn_attr_modify*( ), XFN_attribute_operations, **<xfn/xfn.h>**.

NAME

FN_attrmodlist_t — a list of attribute modifications

SYNOPSIS

```
#include <xfn/xfn.h>

FN_attrmodlist_t *fn_attrmodlist_create(void);

void fn_attrmodlist_destroy(FN_attrmodlist_t *modlist);

FN_attrmodlist_t *fn_attrmodlist_copy(const FN_attrmodlist_t *modlist);

FN_attrmodlist_t *fn_attrmodlist_assign(
    FN_attrmodlist_t *dst,
    const FN_attrmodlist_t *src);

unsigned int fn_attrmodlist_count(const FN_attrmodlist_t *modlist);

const FN_attribute_t *fn_attrmodlist_first(
    const FN_attrmodlist_t *modlist,
    void* *iter_pos,
    unsigned int *first_mod_op);

const FN_attribute_t *fn_attrmodlist_next(
    const FN_attrmodlist_t *modlist,
    void* *iter_pos,
    unsigned int *mod_op);

int fn_attrmodlist_add(
    FN_attrmodlist_t *modlist,
    unsigned int mod_op,
    const FN_attribute_t *attr);
```

DESCRIPTION

An attribute modification list allows for multiple modification operations to be made on the attributes associated with a single named object. It is used in the *fn_attr_multi_modify*( ) operation.

An attribute modification list is a list of attribute modification specifiers. An attribute modification specifier consists of an attribute object and an operation specifier. The attribute's identifier indicates the attribute that is to be operated upon. The attribute's values are used in a manner depending on the operation. The operation specifier is an **unsigned int** that must have one of the values: FN_ATTR_OP_ADD, FN_ATTR_OP_ADD_EXCLUSIVE, FN_ATTR_OP_REMOVE, FN_ATTR_OP_ADD_VALUES, or FN_ATTR_OP_REMOVE_VALUES. (See *fn_attr_modify*( ) for detailed descriptions of these specifiers.) The operations are to be performed in the order in which they appear in the modification list.

fn_attrmodlist_create( ) creates an empty attribute modification list. *fn_attrmodlist_destroy*( ) releases the storage associated with *modlist*; if *modlist* is NULL, no action is taken. *fn_attrmodlist_copy*( ) returns a copy of the attribute modification list *modlist*. *fn_attrmodlist_assign*( ) makes a copy of *src* and assigns it to *dst*, releasing any old contents of *dst*. It returns a pointer to the same object as *dst*.

fn_attrmodlist_count( ) returns the number attribute modification items in the attribute modification list.

The iterators *fn_attrmodlist_first*( ) and *fn_attrmodlist_next*( ) return a handle to the attribute part of the modification and return the operation specifier part through an **unsigned int *** parameter. *fn_attrmodlist_first*( ) returns the attribute of the first modification item from *modlist* and sets *mod_op* to be the code of the modification operation of that item; *iter_pos* is set after the first modification item. *fn_attrmodlist_next*( ) returns the attribute of the next modification item from *modlist* after *iter_pos* and advances *iter_pos*; *mod_op* is set to the code of the modification operation of that item. The order of the items returned during an enumeration is the same as the order by which the items were added to the modification list.

fn_attrmodlist_add( ) adds a new item consisting of the given modification operation code *mod_op* and attribute *attr* to the end of the modification list *modlist*. *attr*'s identifier indicates the attribute that is to be operated upon. *attr*'s values are used in a manner depending on the operation.

RETURN VALUE

fn_attrmodlist_first( ) returns 0 if the modification list is empty. *fn_attrmodlist_next*( ) returns 0 if there are no more items on the modification list to be enumerated or if the iteration marker is invalid.

fn_attrmodlist_add( ) returns 1 if the operation succeeds, 0 if the operation fails.

APPLICATION USAGE

Manipulation of attributes using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to attributes in the underlying naming system can only be effected through the use of the interfaces described in the manual page for XFN_attribute_operations.

SEE ALSO

FN_attribute_t, **FN_attrset_t**, **FN_identifier_t**, *fn_attr_multi_modify*( ), *fn_attr_modify*( ), XFN_attribute_operations, **<xfn/xfn.h>**.

NAME

FN_attrset_t — a set of XFN attributes

SYNOPSIS

```
#include <xfn/xfn.h>

FN_attrset_t *fn_attrset_create(void);

void fn_attrset_destroy(FN_attrset_t *aset);

FN_attrset_t *fn_attrset_copy(const FN_attrset_t *aset);

FN_attrset_t *fn_attrset_assign(
    FN_attrset_t *dst,
    const FN_attrset_t *src);

const FN_attribute_t *fn_attrset_get(
    const FN_attrset_t *aset,
    const FN_identifier_t *attr_id);

unsigned int fn_attrset_count(const FN_attrset_t *aset);

const FN_attribute_t *fn_attrset_first(
    const FN_attrset_t *aset,
    void* *iter_pos);

const FN_attribute_t *fn_attrset_next(
    const FN_attrset_t *aset,
    void* *iter_pos);

int fn_attrset_add(
    FN_attrset_t *aset,
    const FN_attribute_t *attr,
    unsigned int exclusive);

int fn_attrset_remove(
    FN_attrset_t *aset,
    const FN_identifier_t *attr_id);
```

DESCRIPTION

An attribute set is a set of attribute objects with distinct identifiers. The *fn_attr_multi_get*( ) operation takes an attribute set as parameter and returns an attribute set. The *fn_attr_get_ids*( ) operation returns an attribute set containing the identifiers of the attributes.

Attribute sets are represented by the type **FN_attrset_t**. The following operations are defined for manipulating attribute sets.

fn_attrset_create( ) creates an empty attribute set. *fn_attrset_destroy*( ) releases the storage associated with the attribute set *aset*; if *aset* is NULL, no action is taken. *fn_attrset_copy*( ) returns a copy of the attribute set *aset*. *fn_attrset_assign*( ) makes a copy of the attribute set *src* and assigns it to *dst*, releasing any old contents of *dst*. A pointer to the same object as *dst* is returned.

fn_attrset_get( ) returns the attribute with the given identifier *attr_id* from *aset*. *fn_attrset_count*( ) returns the number attributes found in the attribute set *aset*.

fn_attrset_first( ) and *fn_attrset_next*( ) are functions that can be used to return an enumeration of all the attributes in an attribute set. The attributes are not ordered in any way. There is no guaranteed relation between the order in which items are added to an attribute set and the order of the enumeration. The specification does guarantee that any two enumeration will return the members in the same order, provided that no *fn_attrset_add*( ) or *fn_attrset_remove*( ) operation was performed on the object in between or during the two enumerations. *fn_attrset_first*( ) returns the first attribute from the set and sets *iter_pos* after the first attribute. *fn_attrset_next*( ) returns the attribute following *iter_pos* and advances *iter_pos*.

fn_attrset_add( ) adds the attribute *attr* to the attribute set *aset*, replacing the attribute's values if the identifier of *attr* is not distinct in *aset* and *exclusive* is zero. If *exclusive* is non-zero and the identifier of *attr* is not distinct in *aset*, the operation fails. *fn_attrset_remove*( ) removes the attribute with the identifier *attr_id* from *aset*. The operation succeeds even if no such attribute occurs in *aset*.

RETURN VALUE

fn_attrset_first( ) returns 0 if the attribute set is empty. *fn_attrset_next*( ) returns 0 if there are no more attributes in the set.

fn_attrset_add( ) and *fn_attrset_remove*( ) return 1 if the operation succeeds, and 0 if the operation fails.

APPLICATION USAGE

Manipulation of attributes using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to attributes in the underlying naming system can only be effected through the use of the interfaces described in the manual page for XFN_attribute_operations.

SEE ALSO

FN_attribute_t, **FN_attrvalue_t**, **FN_identifier_t**, *fn_attr_multi_get*( ), *fn_attr_get_ids*( ), XFN_attribute_operations, **<xfn/xfn.h>**.

NAME

FN_attrvalue_t — an XFN attribute value

SYNOPSIS

```
#include <xfn/xfn.h>
```

DESCRIPTION

The type **FN_attrvalue_t** is used to represent the contents of a single attribute value, within an attribute of type **FN_attribute_t**.

The representation of this structure is defined by XFN as follows:

```
typedef struct {
    size_t length;
    void *contents;
} FN_attrvalue_t;
```

SEE ALSO

FN_attribute_t, *fn_attr_get_values*( ), **<xfn/xfn.h>**.

NAME

FN_composite_name_t — a sequence of component names spanning multiple naming systems

SYNOPSIS

```
#include <xfn/xfn.h>

FN_composite_name_t *fn_composite_name_create(void);

void fn_composite_name_destroy(FN_composite_name_t *name);

FN_composite_name_t *fn_composite_name_from_str(
    const unsigned char *cstr);

FN_composite_name_t *fn_composite_name_from_string(
    const FN_string_t *str);

FN_string_t *fn_string_from_composite_name(
    const FN_composite_name_t *name,
    unsigned int *status);

FN_composite_name_t *fn_composite_name_copy(
    const FN_composite_name_t *name);

FN_composite_name_t *fn_composite_name_assign(
    FN_composite_name_t *dst,
    const    FN_composite_name_t *src);

int fn_composite_name_is_empty(const FN_composite_name_t *name);

unsigned int fn_composite_name_count(const FN_composite_name_t *name);

const FN_string_t *fn_composite_name_first(
    const     FN_composite_name_t *name,
    void* *iter_pos);

const FN_string_t *fn_composite_name_next(
    const FN_composite_name_t *name,
    void* *iter_pos);

const FN_string_t *fn_composite_name_prev(
    const FN_composite_name_t *name,
    void* *iter_pos);

const FN_string_t *fn_composite_name_last(
    const     FN_composite_name_t *name,
    void* *iter_pos);

FN_composite_name_t *fn_composite_name_prefix(
    const     FN_composite_name_t *name,
    const     void* iter_pos);

FN_composite_name_t *fn_composite_name_suffix(
    const     FN_composite_name_t *name,
```

```
    const     void* iter_pos);

int fn_composite_name_is_equal(
    const     FN_composite_name_t *name,
    const     FN_composite_name_t *name2,
    unsigned int *status);

int fn_composite_name_is_prefix(
    const     FN_composite_name_t *name,
    const     FN_composite_name_t *prefix,
    void* *iter_pos,
    unsigned int *status);

int fn_composite_name_is_suffix(
    const    FN_composite_name_t *name,
    const     FN_composite_name_t *suffix,
    void* *iter_pos,
    unsigned int *status);

int fn_composite_name_prepend_comp(
    FN_composite_name_t *name,
    const     FN_string_t *newcomp);

int fn_composite_name_append_comp(
    FN_composite_name_t *name,
    const     FN_string_t *newcomp);

int fn_composite_name_insert_comp(
    FN_composite_name_t *name,
    void* *iter_pos,
    const     FN_string_t *newcomp);

int fn_composite_name_delete_comp(
    FN_composite_name_t *name,
    void* *iter_pos);

int fn_composite_name_prepend_name(
    FN_composite_name_t *name,
    const     FN_composite_name_t *newcomps);

int fn_composite_name_append_name(
    FN_composite_name_t *name,
    const     FN_composite_name_t *newcomps);

int fn_composite_name_insert_name(
    FN_composite_name_t *name,
    void* *iter_pos,
    const     FN_composite_name_t *newcomps);
```

DESCRIPTION

A composite name is represented by an object of type **FN_composite_name_t**. Each component is a string name, of type **FN_string_t**, from the namespace of a single naming system. It may be an atomic name or a compound name in that namespace.

fn_composite_name_create( ) creates an **FN_composite_name_t** object with zero components. Components may be subsequently added to the composite name using the modify operations described below. *fn_composite_name_destroy*( ) releases any storage associated with the given **FN_composite_name_t** handle. If the argument to *fn_composite_name_destroy*( ) is NULL, no action is taken.

fn_composite_name_from_str( ) creates an **FN_composite_name_t** from the given null-terminated string based on the code set of the current locale setting, using the XFN composite name syntax. *fn_composite_name_from_string*( ) creates an **FN_composite_name_t** from the string *str* using the XFN composite name syntax. *fn_string_from_composite_name*( ) returns the standard string form of the given composite name, by concatenating the components of the composite name in a left to right order, each separated by the XFN component separator.

fn_composite_name_copy( ) returns a copy of the given composite name object. *fn_composite_name_assign*( ) makes a copy of the composite name object pointed to by *src* and assigns it to *dst*, releasing any old contents of *dst*. A pointer to the same object as *dst* is returned.

fn_composite_name_is_empty( ) returns 1 if the given composite name is an empty composite name (that is, consists of a single, empty component name); otherwise, it returns 0. *fn_composite_name_count*( ) returns the number of components in the given composite name.

The iteration scheme is based on the exchange of an opaque **void *** argument, *iter_pos*, that serves to record the position of the iteration in the sequence. Conceptually, *iter_pos* records a position between two successive components (or at one of the extreme ends of the sequence).

The function *fn_composite_name_first*( ) returns a handle to the **FN_string_t** that is the first component in the name, and sets *iter_pos* to indicate the position immediately following the first component. It returns 0 if the name has no components. Thereafter, successive calls of the *fn_composite_name_next*( ) function return pointers to the component following the iteration marker, and advance the iteration marker. If the iteration marker is at the end of the sequence, *fn_composite_name_next*( ) returns 0. Similarly, *fn_composite_name_prev*( ) returns the component preceding the iteration pointer and moves the marker back one component. If the marker is already at the beginning of the sequence, *fn_composite_name_prev*( ) returns 0. The function *fn_composite_name_last*( ) returns a pointer to the last component of the name and sets the iteration marker immediately preceding this component (so that subsequent calls to *fn_composite_name_prev*( ) can be used to step through leading components of the name).

The *fn_composite_name_suffix*( ) function returns a composite name consisting of a copy of those components following the supplied iteration marker. The method *fn_composite_name_prefix*( ) returns a composite name consisting of those components that precede the iteration marker. Using these functions with an iteration marker that was not initialized using *fn_composite_name_first*( ), *fn_composite_name_last*( ), *fn_composite_name_is_prefix*( ), or *fn_composite_name_is_suffix*( ) yields undefined and generally undesirable behavior.

The functions *fn_composite_name_is_equal*( ), *fn_composite_name_is_prefix*( ), *fn_composite_name_is_suffix*( ), test for equality between composite names or between parts of composite names. For these functions, equality is defined as exact string equality, not name equivalence. A name's syntactic property, such as case-insensitivity, is not taken into account by these functions.

The function *fn_composite_name_is_prefix*( ) tests if one composite name is a prefix of another. If so, it returns 1 and sets the iteration marker immediately following the prefix. (So for example, a subsequent call to *fn_composite_name_suffix*( ) will return the remainder of the name.) Otherwise it returns 0 and value of the iteration marker is undefined. The function *fn_composite_name_is_suffix*( ) is similar. It tests if a one composite name is a suffix of another. If so it returns 1 and sets the iteration marker immediately preceding the suffix.

fn_composite_name_prepend_comp( ) and *fn_composite_name_append_comp*( ) prepends and appends a single component to the given composite name, respectively. These operations invalidate any iteration marker the client holds for that object. *fn_composite_name_insert_comp*( ) inserts a single component before *iter_pos* to the given composite name and sets *iter_pos* to be immediately after the component just inserted. *fn_composite_name_delete_comp*( ) deletes the component located before *iter_pos* from the given composite name and sets *iter_pos* back one component.

The functions *fn_composite_name_prepend_name*( ), *fn_composite_name_append_name*( ) and *fn_composite_name_insert_name*( ) perform the same update functions as their *_comp* counterparts, respectively, except that multiple components are being added, rather than single components. *fn_composite_name_insert_name*( ) sets *iter_pos* to be immediately after the name just added.

RETURN VALUE

The functions *fn_composite_name_is_empty*( ), *fn_composite_name_is_equal*( ), *fn_composite_name_is_suffix*( ) and *fn_composite_name_is_prefix*( ) return 1 if the test indicated is TRUE; 0 otherwise.

The update functions *fn_composite_name_prepend_comp*( ), *fn_composite_name_append_comp*( ), *fn_composite_name_insert_comp*( ), *fn_composite_name_delete_comp*( ) and their *_name* counterparts return 1 if the update was successful; 0 otherwise.

If a function is expected to return a pointer to an object, a null pointer (0) is returned if the function fails.

ERRORS

Locale (code set, language, or territory) mismatches that occur during the composition of the string form or during comparisons of composite names are resolved in an implementation-dependent way. *fn_string_from_composite_name*( ), *fn_composite_name_is_equal*( ), *fn_composite_name_is_suffix*( ) and *fn_composite_name_is_prefix*( ) set *status* to the appropriate error code ([FN_E_INCOMPATIBLE_CODE_SETS] or [FN_E_INCOMPATIBLE_LOCALES]) for composite names whose components have code sets or locales that are determined by the implementation to be incompatible. If the *status* argument to these functions is NULL, no status is set upon return of the function.

SEE ALSO

FN_string_t, **<xfn/xfn.h>**.

NAME

FN_compound_name_t — an XFN compound name

SYNOPSIS

```
#include <xfn/xfn.h>

FN_compound_name_t *fn_compound_name_from_syntax_attrs(
    const     FN_attrset_t *aset,
    const     FN_string_t *name,
    FN_status_t *status);

FN_attrset_t *fn_compound_name_get_syntax_attrs(
    const FN_compound_name_t *name);

void fn_compound_name_destroy(FN_compound_name_t *name);

FN_string_t *fn_string_from_compound_name(
    const FN_compound_name_t *name);

FN_compound_name_t *fn_compound_name_copy(
    const FN_compound_name_t *name);

FN_compound_name_t *fn_compound_name_assign(
    FN_compound_name_t *dst,
    const    FN_compound_name_t *src);

unsigned int fn_compound_name_count(const FN_compound_name_t *name);

const FN_string_t *fn_compound_name_first(
    const     FN_compound_name_t *name,
    void* *iter_pos);

const FN_string_t *fn_compound_name_next(
    const     FN_compound_name_t *name,
    void* *iter_pos);

const FN_string_t *fn_compound_name_prev(
    const     FN_compound_name_t *name,
    void* *iter_pos);

const FN_string_t *fn_compound_name_last(
    const     FN_compound_name_t *name,
    void* *iter_pos);

FN_compound_name_t *fn_compound_name_prefix(
    const    FN_compound_name_t *name,
    const     void* iter_pos);

FN_compound_name_t *fn_compound_name_suffix(
    const     FN_compound_name_t *name,
    const     void* iter_pos);

int fn_compound_name_is_empty(const FN_compound_name_t *name);
```

```
int fn_compound_name_is_equal(
    const FN_compound_name_t *name1,
    const FN_compound_name_t *name2,
    unsigned int *status);

int fn_compound_name_is_prefix(
    const     FN_compound_name_t *name,
    const     FN_compound_name_t *pre,
    void* *iter_pos,
    unsigned int *status);

int fn_compound_name_is_suffix(
    const     FN_compound_name_t *name,
    const     FN_compound_name_t *suffix,
    void* *iter_pos,
    unsigned int *status);

int fn_compound_name_prepend_comp(
    FN_compound_name_t *name,
    const FN_string_t *atomic_comp,
    unsigned int *status);

int fn_compound_name_append_comp(
    FN_compound_name_t *name,
    const FN_string_t *atomic_comp,
    unsigned int *status);

int fn_compound_name_insert_comp(
    FN_compound_name_t *name,
    void* *iter_pos,
    const     FN_string_t *atomic_comp,
    unsigned int *status);

int fn_compound_name_delete_comp(
    FN_compound_name_t *name,
    void* *iter_pos);

int fn_compound_name_delete_all(FN_compound_name_t *name);
```

DESCRIPTION

Most applications treat names as opaque data and hence, the majority of clients of the XFN interface will not need to parse names. Some applications, however, such as browsers, need to parse names. For these applications, XFN provides support in the form of the **FN_compound_name_t** object.

Each naming system in an XFN federation potentially has its own naming conventions. The **FN_compound_name_t** object has associated operations for applications to process compound names that conform to the XFN model of expressing compound name syntax. The XFN syntax model for compound names covers a large number of specific name syntaxes and is expressed in terms of syntax properties of the naming convention. See XFN_compound_syntax.

An **FN_compound_name_t** object is constructed by the operation *fn_compound_name_from_syntax_attrs*( ), using a string name and an attribute set containing the

fn_syntax_type (FN_ID_STRING syntax) attribute identifying the namespace syntax of the string name. The value **standard** (FN_ID_STRING syntax) in the *fn_syntax_type* specifies a syntax model that is by default supported by the **FN_compound_name_t** object. An implementation may support other syntax types instead of the XFN standard syntax model, in which case, the value of the *fn_syntax_type* attribute would be set to an implementation specific string. *fn_compound_name_get_syntax_attrs*( ) returns an attribute set containing the syntax attributes that describes the given compound name. *fn_compound_name_destroy*( ) releases the storage associated with the given compound name. If the argument to *fn_compound_name_destroy*( ) is NULL, no action is taken.

fn_string_from_compound_name( ) returns the string form of the given compound name. *fn_compound_name_copy*( ) returns a copy of the given compound name. *fn_compound_name_assign*( ) makes a copy of the compound name *src* and assigns it to *dst*, releasing any old contents of *dst*. A pointer to the object pointed to by *dst* is returned. *fn_compound_name_count*( ) returns the number of atomic components in the given compound name.

The function *fn_compound_name_first*( ) returns a handle to the **FN_string_t** that is the first atomic component in the compound name, and sets *iter_pos* to indicate the position immediately following the first component. It returns 0 if the name has no components. Thereafter, successive calls of the *fn_compound_name_next*( ) function return pointers to the component following the iteration marker, and advance the iteration marker. If the iteration marker is at the end of the sequence, *fn_compound_name_next*( ) returns 0. Similarly, *fn_compound_name_prev*( ) returns the component preceding the iteration pointer and moves the marker back one component. If the marker is already at the beginning of the sequence, *fn_compound_name_prev*( ) returns 0. The function *fn_compound_name_last*( ) returns a pointer to the last component of the name and sets the iteration marker immediately preceding this component (so that subsequent calls to *fn_compound_name_prev*( ) can be used to step through trailing components of the name).

The *fn_compound_name_suffix*( ) function returns a compound name consisting of a copy of those components following the supplied iteration marker. The function *fn_compound_name_prefix*( ) returns a compound name consisting of those components that precede the iteration marker. Using these functions with an iteration marker that was not initialized using *fn_compound_name_first*( ), *fn_compound_name_last*( ), *fn_compound_name_is_prefix*( ), or *fn_compound_name_is_suffix*( ) yields undefined and generally undesirable behavior.

The function *fn_compound_name_is_equal*( ), *fn_compound_name_is_prefix*( ), *fn_compound_name_is_suffix*( ), tests for equality between compound names or between parts of compound names. For these functions, equality is defined as name equivalence. A name's syntactic property, such as case-insensitivity, is taken into account by these functions.

The function *fn_compound_name_is_prefix*( ) tests if one compound name is a prefix of another. If so, it returns 1 and sets the iteration marker immediately following the prefix. (So for example, a subsequent call to *fn_compound_name_suffix*( ) will return the remainder of the name.) Otherwise it returns 0 and value of the iteration marker is undefined. The function *fn_compound_name_is_suffix*( ) is similar. It tests if a one compound name is a suffix of another. If so it returns 1 and sets the iteration marker immediately preceding the suffix.

fn_compound_name_prepend_comp( ) and *fn_compound_name_append_comp*( ) prepends and appends a single atomic component to the given compound name, respectively. These operations invalidates any iteration marker the client holds for that object. *fn_compound_name_insert_comp*( ) inserts an atomic component before *iter_pos* to the given compound name and sets *iter_pos* to be immediately after the component just inserted. *fn_compound_name_delete_comp*( ) deletes the atomic component located before *iter_pos* from the given compound name and sets *iter_pos* back one component. *fn_compound_name_delete_all*( )

deletes all the atomic components from *name*.

RETURN VALUE

The functions *fn_compound_name_is_empty*( ), *fn_compound_name_is_equal*( ), *fn_compound_name_is_suffix*( ) and *fn_compound_name_is_prefix*( ) return 1 if the test indicated is TRUE; 0 otherwise.

The update functions *fn_compound_name_prepend_comp*( ), *fn_compound_name_append_comp*( ), *fn_compound_name_insert_comp*( ), *fn_compound_name_delete_comp*( ) and *fn_compound_name_delete_all*( ) return 1 if the update was successful; 0 otherwise.

If a function is expected to return a pointer to an object, a null pointer (0) is returned if the function fails.

ERRORS

When the function *fn_compound_name_from_syntax_attrs*( ) fails, it returns in *status* a status code. The possible status codes are:

[FN_E_ILLEGAL_NAME]
The name supplied to the operation was not a well- formed XFN compound name, or one of the component names was not well-formed according to the syntax of the naming system(s) involved in its resolution.

[FN_E_INCOMPATIBLE_CODE_SETS]
The code set of the given string is incompatible with that supported by the compound name.

[FN_E_INCOMPATIBLE_LOCALES]
Language or territory locale information of the given string is incompatible with that supported by the compound name.

[FN_E_INVALID_SYNTAX_ATTRS]
The syntax attributes supplied are invalid or insufficient to fully specify the syntax.

[FN_E_SYNTAX_NOT_SUPPORTED]
The syntax type specified is not supported.

The following functions:
fn_compound_name_from_syntax_attrs( ),
fn_compound_name_is_equal( ),
fn_compound_name_is_suffix( ),
fn_compound_name_is_prefix( ),
fn_compound_name_prepend_comp( ),
fn_compound_name_append_comp( )
fn_compound_name_insert_comp( )
returns in *status* an appropriate error code ([FN_E_INCOMPATIBLE_CODE_SETS] or [FN_E_INCOMPATIBLE_LOCALES]) when the code set or locale of the given string is incompatible with that of the compound name. If the *status* argument to these functions is NULL, no status is set upon return of the function.

SEE ALSO

FN_attribute_t, **FN_attrset_t**, **FN_composite_name_t**, **FN_status_t**, **FN_string_t**, *fn_ctx_get_syntax_attrs*( ), XFN_compound_syntax, **<xfn/xfn.h>**.

NAME

FN_ctx_t — an XFN context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ctx_t*
fn_ctx_handle_from_initial(
        unsigned int authoritative,
        FN_status_t* status);

FN_ctx_t*
fn_ctx_handle_from_ref(
        const FN_ref_t* ref,
        unsigned int authoritative,
        FN_status_t* status);

FN_ref_t *fn_ctx_get_ref(
    const    FN_ctx_t *ctx,
    FN_status_t *status);

void fn_ctx_handle_destroy(FN_ctx_t *ctx);

FN_ref_t *fn_ctx_lookup(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);

FN_namelist_t *fn_ctx_list_names(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);

FN_string_t *fn_namelist_next(
    FN_namelist_t *nl,
    FN_status_t *status);

void fn_namelist_destroy(
    FN_namelist_t *nl);

FN_bindinglist_t *fn_ctx_list_bindings(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);

FN_string_t *fn_bindinglist_next(
    FN_bindinglist_t *iter,
    FN_ref_t * *ref,
    FN_status_t *status);

void fn_bindinglist_destroy(
    FN_bindinglist_t *iter_pos);
```

```
int fn_ctx_bind(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    const    FN_ref_t *ref,
    unsigned int exclusive,
    FN_status_t *status);

int fn_ctx_unbind(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);

int fn_ctx_rename(
    FN_ctx_t *ctx,
    const FN_composite_name_t *oldname,
    const FN_composite_name_t *newname,
    unsigned int exclusive,
    FN_status_t *status);

FN_ref_t *fn_ctx_create_subcontext(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);

int fn_ctx_destroy_subcontext(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);

FN_ref_t *fn_ctx_lookup_link(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);

FN_attrset_t *fn_ctx_get_syntax_attrs(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);

FN_composite_name_t *fn_ctx_equivalent_name(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_string_t *leading_name,
    FN_status_t *status);
```

DESCRIPTION

An XFN context consists of a set of name to reference bindings. An XFN context is represented by the type **FN_ctx_t** in the client interface. The operations for manipulating an **FN_ctx_t** object are described in detail in separate reference manual pages. The following contains a brief summary of these operations.

fn_ctx_handle_from_initial( ) returns a pointer to an Initial Context that provides a starting point for resolution of composite names. *fn_ctx_handle_from_ref*( ) returns a handle to an **FN_ctx_t**

object using the given reference *ref*. *fn_ctx_get_ref*( ) returns the reference of the context *ctx*. *fn_ctx_handle_destroy*( ) releases the resources associated with the **FN_ctx_t** object *ctx*; it does not affect the state of the context itself. If the argument to *fn_ctx_handle_destroy*( ) is NULL, no action is taken.

fn_ctx_lookup( ) returns the reference bound to *name* resolved relative to *ctx*. *fn_ctx_list_names*( ) is used to enumerate the atomic names bound in the context named by *name* resolved relative to *ctx*. *fn_ctx_list_bindings*( ) is used to enumerate the atomic names and their references in the context named by *name* resolved relative to *ctx*.

fn_ctx_bind( ) binds the composite name *name* to a reference *ref* resolved relative to *ctx*. *fn_ctx_unbind*( ) unbinds *name* resolved relative to *ctx*. *fn_ctx_rename*( ) binds *newname* to the reference bound to *oldname* and unbinds *oldname*. *oldname* is resolved relative to *ctx*; *newname* is resolved relative to the target context.

fn_ctx_create_subcontext( ) creates a new context with the given composite name *name* resolved relative to *ctx*. *fn_ctx_destroy_subcontext*( ) destroys the context named by *name* resolved relative to *ctx*.

Normal resolution always follow links. *fn_ctx_lookup_link*( ) lookups up *name* relative to *ctx*, following links except for the last atomic part of *name*.

fn_ctx_get_syntax_attrs( ) returns an attribute set containing attributes that describes a context's syntax. *name* must name a context.

fn_ctx_equivalent_name( ) returns an equivalent name for an object named *name* relative to the context *ctx*. The equivalent name returned has *leading_name* as its initial atomic name and is to resolved relative to the same context, *ctx*.

ERRORS

In each context operation, the caller supplies an **FN_status_t** object as a parameter. The called function sets this status object as described in the reference manual pages for **FN_status_t** and XFN_status_code. If the caller supplies a NULL pointer for the status object, no status information is returned.

APPLICATION USAGE

In most of the operations of the base context interface, the caller supplies a context and a composite name. The supplied name is always interpreted relative to the supplied context.

The operation may eventually be effected on a different context called the operation's *target context*. Each operation has an initial resolution phase that conveys the operation to its target context, and the operation is then applied. The effect (but not necessarily the implementation) is that of

1. doing a lookup on that portion of the name that represents the target context, and then
2. invoking the operation on the target context.

The contexts involved only in the resolution phase are called *intermediate contexts.*

Normal resolution of names in context operations always follows XFN links.

SEE ALSO

FN_attrset_t, **FN_composite_name_t**, **FN_ref_t**, **FN_status_t**, *fn_ctx_create_subcontext*( ), *fn_ctx_bind*( ), *fn_ctx_destroy_subcontext*( ), *fn_ctx_equivalent_name*( ), *fn_ctx_handle_destroy*( ), *fn_ctx_handle_from_initial*( ), *fn_ctx_handle_from_ref*( ), *fn_ctx_get_ref*( ), *fn_ctx_get_syntax_attrs*( ), *fn_ctx_list_names*( ), *fn_ctx_list_bindings*( ), *fn_ctx_lookup*( ), *fn_ctx_lookup_link*( ), *fn_ctx_rename*( ), *fn_ctx_unbind*( ), XFN_links, XFN_status_codes, **<xfn/xfn.h>**.

NAME

FN_identifier_t — an XFN identifier

SYNOPSIS

```
#include <xfn/xfn.h>
```

DESCRIPTION

Identifiers are used to identify reference types and address types in an XFN reference, and to identify attributes and their syntax in the attribute operations.

An XFN identifier consists of an **unsigned int**, which determines the format of identifier, and the actual identifier, which is expressed as a sequence of octets.

The representation of this structure is defined by XFN as follows:

```
typedef struct {
    unsigned int format;
    size_t length;
    void *contents;
} FN_identifier_t;
```

XFN defines a small number of standard forms for identifiers.

Identifier Format	Description
FN_ID_STRING	The identifier is an ASCII string (ISO 646).
FN_ID_DCE_UUID	The identifier is an OSF DCE UUID in string representation. (See the X/Open DCE RPC.)
FN_ID_ISO_OID_STRING	The identifier is an ISO OID in ASN.1 dot-separated integer list string format. (See the ISO ASN.1.)

SEE ALSO

FN_ref_t, **FN_ref_addr_t**, **FN_attribute_t**, **<xfn/xfn.h>**.

NAME

FN_ref_t — an XFN reference

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ref_t *fn_ref_create(const FN_identifier_t *ref_type);

void fn_ref_destroy(FN_ref_t *ref);

FN_ref_t *fn_ref_copy(const FN_ref_t *ref);

FN_ref_t *fn_ref_assign(FN_ref_t *dst, const FN_ref_t *src);

const FN_identifier_t *fn_ref_type(const FN_ref_t *ref);

unsigned int fn_ref_addrcount(const FN_ref_t *ref);

const FN_ref_addr_t *fn_ref_first(const FN_ref_t *ref, void* *iter_pos);

const FN_ref_addr_t *fn_ref_next(const FN_ref_t *ref, void* *iter_pos);

int fn_ref_prepend_addr(FN_ref_t *ref, const FN_ref_addr_t *addr);

int fn_ref_append_addr(FN_ref_t *ref, const FN_ref_addr_t *addr);

int fn_ref_insert_addr(
    FN_ref_t *ref,
    void* *iter_pos,
    const FN_ref_addr_t *addr);

int fn_ref_delete_addr(FN_ref_t *ref, void* *iter_pos);

int fn_ref_delete_all(FN_ref_t *ref);

FN_ref_t *fn_ref_create_link(const FN_composite_name_t *link_name);

int fn_ref_is_link(const FN_ref_t *ref);

FN_composite_name_t *fn_ref_link_name(const FN_ref_t *link_ref);

FN_string_t *fn_ref_description(
    const FN_ref_t *ref,
    unsigned int detail,
    unsigned int *more_detail);
```

DESCRIPTION

An XFN reference is represented by the type **FN_ref_t**. An object of this type contains a reference type and a list of addresses. The ordering in this list at the time of binding might not be preserved when the reference is returned upon lookup.

The reference type is represented by an object of type **FN_identifier_t**. The reference type is intended to identify the class of object referenced. XFN does not dictate the precise use of this.

Each address is represented by an object of type **FN_ref_addr_t**.

fn_ref_create( ) creates a reference with no address, using *ref_type* as its reference type. Addresses can be later added to the reference using the functions described below. *fn_ref_destroy*( ) releases the storage associated with *ref*; if *ref* is NULL, no action is taken. *fn_ref_copy*( ) creates a copy of *ref* and returns it. *fn_ref_assign*( ) creates a copy of *src* and assigns it to *dst*, releasing any old contents of *dst*. A pointer to the same object as *dst* is returned.

fn_ref_type( ) returns the reference type of *ref*.

fn_ref_addrcount( ) returns the number of addresses in the reference *ref*.

fn_ref_first( ) returns the first address in *ref* and sets *iter_pos* to be after the address. It returns 0 if there is no address in the list. *fn_ref_next*( ) returns the address following *iter_pos* in *ref* and sets *iter_pos* to be after the address. If the iteration marker *iter_pos* is at the end of the sequence, *fn_ref_next*( ) returns 0.

fn_ref_prepend_addr( ) adds *addr* to the front of the list of addresses in *ref*. *fn_ref_append_addr*( ) adds *addr* to the end of the list of addresses in *ref*. *fn_ref_insert_addr*( ) adds *addr* to *ref* before *iter_pos* and sets *iter_pos* to be immediately after the new reference added. *fn_ref_delete_addr*( ) deletes the address located before *iter_pos* in the list of addresses in *ref* and sets *iter_pos* back one address. *fn_ref_delete_all*( ) deletes all addresses in *ref*.

fn_ref_create_link( ) creates a reference using the given composite name *link_name* as an address. The reference type of this reference is defined in Appendix G on page 299. *fn_ref_is_link*( ) tests if *ref* is a link. It returns 1 if it is; 0 if it is not. *fn_ref_link_name*( ) returns the composite name stored in a link reference. It returns 0 if *link_ref* is not a link.

fn_ref_description( ) returns a string description of the given reference. It takes as argument an integer, *detail*, and a pointer to an integer *more_detail*. *detail* specifies the level of detail for which the description should be generated; the higher the number, the more detail is to be provided. If *more_detail* is zero, it is ignored. If *more_detail* is non-zero, it is set by the description operation to indicate the next level of detail available, beyond that specified by *detail*. If no higher level of detail is available, *more_detail* is set to *detail*.

RETURN VALUE

The operations *fn_ref_prepend_addr*( ), *fn_ref_append_addr*( ), *fn_ref_insert_addr*( ), *fn_ref_delete_addr*( ) and *fn_ref_delete_all*( ) return 1 if the operation succeeds, 0 if the operation fails.

APPLICATION USAGE

The reference type is intended to identify the class of object referenced. XFN does not dictate the precise use of this.

Multiple addresses in a single reference are intended to identify multiple communication endpoints for the same conceptual object. Multiple addresses may arise for various reasons, such as the object offering interfaces over more than one communication mechanism.

The client must interpret the contents of a reference based on the type of the addresses and the type of the reference. However, this interpretation is intended to occur below the application layer. Most applications developers should not have to manipulate the contents of either address or reference objects themselves. These interfaces would generally be used within service libraries.

Manipulation of references using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to references in the underlying naming system can only be effected through the use of the interfaces described in the reference manual page for **FN_ctx_t**.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_identifier_t**, **FN_string_t**, **FN_ref_addr_t**, **FN_string_t**, *fn_ctx_lookup*( ), *fn_ctx_lookup_link*( ), XFN_links, **<xfn/xfn.h>**.

NAME

FN_ref_addr_t — an address in an XFN reference

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ref_addr_t *fn_ref_addr_create(
    const FN_identifier_t *type,
    size_t length,
    const void *data);

void fn_ref_addr_destroy(FN_ref_addr_t *addr);

FN_ref_addr_t *fn_ref_addr_copy(const FN_ref_addr_t *addr);

FN_ref_addr_t *fn_ref_addr_assign(
    FN_ref_addr_t *dst,
    const FN_ref_addr_t *src);

const FN_identifier_t *fn_ref_addr_type(const FN_ref_addr_t *addr);

size_t fn_ref_addr_length(const FN_ref_addr_t *addr);

const void* fn_ref_addr_data(const FN_ref_addr_t *addr);

FN_string_t *fn_ref_addr_description(
    const FN_ref_addr_t *addr,
    unsigned int detail,
    unsigned int *more_detail);
```

DESCRIPTION

An XFN reference is represented by the type **FN_ref_t**. An object of this type contains a reference type and a list of addresses. Each address in the list is represented by an object of type **FN_ref_addr_t**. An address consists of an opaque data buffer and a type field, of type **FN_identifier_t**.

fn_ref_addr_create( ) creates and returns an address with the given type and data. *length* indicates the size of the data. *fn_ref_addr_destroy*( ) releases the storage associated with the given address. If the argument to *fn_ref_addr_destroy*( ) is NULL, no action is taken. *fn_ref_addr_copy*( ) returns a copy of the given address object. *fn_ref_addr_assign*( ) makes a copy of the address pointed to by *src* and assigns it to *dst*, releasing any old contents of *dst*. A pointer to the same object as *dst* is returned.

fn_ref_addr_type( ) returns the type of the given address. *fn_ref_addr_length*( ) returns the size of the address in bytes. *fn_ref_addr_data*( ) returns the contents of the address.

fn_ref_addr_description( ) returns the implementation-defined textual description of the address. It takes as arguments a number, *detail*, and a pointer to a number *more_detail*. *detail* specifies the level of detail for which the description should be generated; the higher the number, the more detail is to be provided. If *more_detail* is zero, it is ignored. If *more_detail* is non-zero, it is set by the description operation to indicate the next level of detail available, beyond that specified by *detail*. If no higher level of detail is available, *more_detail* is set to *detail*.

APPLICATION USAGE

The address type of an **FN_ref_addr_t** object is intended to identify the mechanism that should be used to reach the object using that address. The client must interpret the contents of the opaque data buffer of the address based on the type of the address, and on the type of the reference that the address is in. However, this interpretation is intended to occur below the application layer. Most applications developers should not have to manipulate the contents of either address or reference objects themselves. These interfaces would generally be used within service libraries.

Multiple addresses in a single reference are intended to identify multiple communication endpoints for the same conceptual object. Multiple addresses may arise for various reasons, such as the object offering interfaces over more than one communication mechanism.

Manipulation of addresses using the operations described in this manual page does not affect their representation in the underlying naming system. Changes to addresses in the underlying naming system can only be effected through the use of the interfaces described in the reference manual page for **FN_ctx_t**.

SEE ALSO

FN_ctx_t, **FN_identifier_t**, **FN_ref_t**, **FN_string_t**, **<xfn/xfn.h>**.

NAME

FN_search_control_t — options for attribute search

SYNOPSIS

```
#include <xfn/xfn.h>

FN_search_control_t *fn_search_control_create(
    unsigned int scope,
    unsigned int follow_links,
    unsigned int max_names,
    unsigned int return_ref,
    const FN_attrset_t *return_attr_ids,
    unsigned int *status);

void fn_search_control_destroy(
    FN_search_control_t *scontrol);

FN_search_control_t *fn_search_control_copy(
    const FN_search_control_t *scontrol);

FN_search_control_t *fn_search_control_assign(
    FN_search_control_t *dst,
    const FN_search_control_t *src);

unsigned int fn_search_control_scope(
    const FN_search_control_t *scontrol);

unsigned int fn_search_control_follow_links(
    const FN_search_control_t *scontrol);

unsigned int fn_search_control_max_names(
    const FN_search_control_t *scontrol);

unsigned int fn_search_control_return_ref(
    const FN_search_control_t *scontrol);

const FN_attrset_t *fn_search_control_return_attr_ids(
    const FN_search_control_t *scontrol);
```

DESCRIPTION

The **FN_search_control_t** object is used to specify options for the attribute search operation *fn_attr_ext_search*( ).

fn_search_control_create( ) creates an **FN_search_control_t** object using information in *scope*, *follow_links*, *max_names*, *return_ref* and *return_attr_ids* to set the search options. If the operation succeeds, *fn_search_control_create*( ) returns a pointer to an **FN_search_control_t** object; otherwise it returns a NULL pointer.

The scope of the search is either the named object, the named context, the named context and its subcontexts, or the named context and a context implementation defined set of subcontexts. The values for scope are:

FN_SEARCH_NAMED_OBJECT
: Search just the given named object.

FN_SEARCH_ONE_CONTEXT
Search just the given context.

FN_SEARCH_SUBTREE
Search given context and all its subcontexts.

FN_SEARCH_CONSTRAINED_SUBTREE
Search given context and its subcontexts as constrained by the context-specific policy in place at the named context.

follow_links further defines the scope and nature of the search. If *follow_links* is non-zero, the search follows XFN links. If *follow_links* is 0, XFN links are not followed. The description of *fn_attr_ext_search*( ) gives more detail about how XFN links are treated.

max_names specifies the maximum number of names to return in an **FN_ext_searchlist_t** enumeration. The names of all objects whose attributes satisfy the filter are returned when *max_names* is 0.

If *return_ref* is non-zero, the reference bound to the named object is returned with the object's name by *fn_ext_searchlist_next*( ). If *return_ref* is 0, the reference is not returned .

Attribute identifiers and values associated with named objects that satisfy the filter may be returned by *fn_ext_searchlist_next*( ). The attributes returned are those listed in *return_attr_ids*. If the value of *return_attr_ids* is 0, all attributes are returned. If *return_attr_ids* is an empty **FN_attrset_t** object, no attributes are returned. Any attribute values in *return_attr_ids* are ignored; only the attribute identifiers are relevant for this operation.

fn_attr_ext_search( ) interprets a value of 0 for the search control argument as a default search control which has the following option settings:

- *scope* is {FN_SEARCH_ONE_CONTEXT}
- *follow_links* is 0 (do not follow links)
- *max_names* is 0 (return all named objects that match filter)
- *return_ref* is 0 (do not return the reference of the named object)
- *return_attr_ids* is an empty **FN_attrset_t** (do not return any attributes of the named object).

fn_search_control_destroy( ) releases the storage associated with *scontrol*; if *scontrol* is NULL, no action is taken.

fn_search_control_copy( ) returns a copy of the search control *scontrol*. *fn_search_control_assign*( ) makes a copy of the search control *src* and assigns it to *dst*, releasing the old contents of *dst*. A pointer to the same object as *dst* is returned.

fn_search_control_scope( ) returns the scope for the search. *fn_search_control_follow_links*( ) returns non-zero if links are followed; 0 if not. *fn_search_control_max_names*( ) returns the maximum number of names. *fn_search_control_return_ref*( ) returns non-zero if the reference is returned; 0 if not. *fn_search_control_return_attr_ids*( ) returns a pointer to the list of attributes; a NULL pointer indicates that all attributes and values are returned.

ERRORS

fn_search_control_create( ) returns a NULL pointer if the operation fails and sets *status* as follows:

[FN_E_SEARCH_INVALID_OPTION]
A supplied search option was invalid or inconsistent.

Other status codes are possible as described in the reference manual pages for XFN_status_codes.

SEE ALSO

fn_attr_ext_search( ), *fn_ext_searchlist_next*( ), **FN_attrset_t**, XFN_status_codes, **<xfn/xfn.h>**.

NAME

FN_search_filter_t — filter expression for attribute search

SYNOPSIS

```
#include <xfn/xfn.h>

FN_search_filter_t *fn_search_filter_create(
    unsigned int *status,
    const unsigned char *estr,
    ... );

void fn_search_filter_destroy(FN_search_filter_t *sfilter);

FN_search_filter_t *fn_search_filter_copy(
    const FN_search_filter_t *sfilter);

FN_search_filter_t *fn_search_filter_assign(
    FN_search_filter_t *dst,
    const FN_search_filter_t *src);

const unsigned char *fn_search_filter_expression(
    const FN_search_filter_t *sfilter);

const void **fn_search_filter_arguments(
    const FN_search_filter_t *sfilter,
    size_t *number_of_arguments);
```

DESCRIPTION

The **FN_search_filter_t** type is an expression that is evaluated against the attributes of named objects bound in the scope of the search operation *fn_attr_ext_search*( ). The filter evaluates to TRUE or FALSE. If the filter is empty, it evaluates to TRUE. Names of objects whose attribute values satisfy the filter expression are returned by the search operation.

If the identifier in any subexpression of the filter does not exist as an attribute of an object then the innermost logical expression containing that identifier is FALSE. A subexpression that is only an attribute tests for the presence of the attribute; the subexpression evaluates to TRUE if the attribute has been defined for the object and FALSE otherwise.

fn_search_filter_create( ) creates a search filter from the expression string *estr* and the remaining arguments. *fn_search_filter_destroy*( ) releases the storage associated with the search filter *sfilter*; if *sfilter* is NULL, no action is taken. *fn_search_filter_copy*( ) returns a copy of the search filter *sfilter*. *fn_search_filter_assign*( ) makes a copy of the search filter *src* and assigns it to *dst*, releasing old contents of *dst*. A pointer to the same object as *dst* is returned.

fn_search_filter_expression( ) returns the filter expression of *sfilter*. *fn_search_filter_arguments*( ) returns an array of pointers to arguments supplied to the filter constructor. *number_of_arguments* is set to the size of this array. The types of the arguments are determined by the substitution tokens in the expression in *sfilter*.

BNF of Filter Expression

```
<FilterExpr> ::= [ <Expr> ]

<Expr> ::= <Expr> "or" <Expr>
    | <Expr> "and" <Expr>
    | "not" <Expr>
```

```
    | "(" <Expr> ")"
    | <Attribute> [ <Rel_Op> <Value> ]
    | <Ext>

<Rel_Op> ::= "==" | "!=" | "<" | "<=" | ">" | ">=" | "~="

<Attribute> ::= "%a"

<Value> ::= <Integer>
    | "%v"
    |<Wildcarded_string>

<Wildcarded_string> ::= "*"
    | <String>
    | {<String> "*"}+ [<String>]
    | {"*" <String>}+ ["*"]

<String> ::= "'" { <Char> } * "'"
    | "%s"

<Char> ::= <PCS> // See BNF in Section 4.1.2 for PCS definition
    | Characters in the repertoire of a string representation

<Identifier> ::= "%i"

<Ext> ::= <Ext_Op> "(" [Arg_List] ")"

<Ext_Op> ::= <String> | <Identifier>

<Arg_List> ::= <Arg> | <Arg> "," <Arg_List>

<Arg> ::= <Value> | <Attribute> | <Identifier>
```

Specification of Filter Expression

The arguments to *fn_search_filter_create*( ) are a return status, an expression string, and a list of arguments. The string contains the filter expression with substitution tokens for the attributes, attribute values, strings and identifiers that are part of the expression. The remaining list of arguments contains the attributes and values in the order of appearance of their corresponding substitution tokens in the expression. The arguments are of types **FN_attribute_t***, **FN_attrvalue_t***, **FN_string_t*** or **FN_identifier_t***. Except when attributes appear as arguments in specially-defined extended operations, any attribute values in an **FN_attribute_t** type of argument are ignored; only the attribute identifier and attribute syntax are relevant. The argument type expected by each substitution token are listed in the following table.

Token	Argument Type
%a	**FN_attribute_t***
%v	**FN_attrvalue_t***
%s	**FN_string_t***
%i	**FN_identifier_t***

Precedence

The following precedence relations hold in the absence of parentheses, in the order of lowest to highest:

- or
- and

- not
- relational operators.

These boolean and relational operators are left associative.

Relational Operators

Comparisons and ordering are specific to the syntax or rules of the supplied attribute.

Locale (code set, language or territory) mismatches that occur during string comparisons and ordering operations are resolved in an implementation-dependent way. Relational operations that have ordering semantics may be used for strings of locales in which ordering is meaningful, but is not of general use in internationalized environments.

An attribute that occurs in the absence of any relational operator tests for the presence of the attribute.

Operator	Meaning
==	The sub-expression is TRUE if at least one value of the specified attribute is equal to the supplied value.
!=	The sub-expression is TRUE if no values of the specified attribute equal the supplied value.
>=	The sub-expression is TRUE if at least one value of the attribute is greater than or equal to the supplied value.
>	The sub-expression is TRUE if at least one value of the attribute is greater then the supplied value.
<=	The sub-expression is TRUE if at least one value of the attribute is less than or equal to the supplied value.
<	The sub-expression is TRUE if at least one value of the attribute is less than the supplied value.
~=	The sub-expression is TRUE if at least one value of the specified attribute matches the supplied value according to some context-specific approximate matching criterion. This criterion must subsume strict equality.

Wildcarded Strings

A wildcarded string consists of a sequence of alternating wildcard specifiers and strings. The sequence can start with either a wildcard specifier or a string, and end with either a wildcard specifier or a string.

The wildcard specifier is denoted by the asterisk character (‘*’) and means 0 or more occurrences of any character.

Wildcarded strings can be used to specify substring matches. The following are examples of wildcarded strings and what they mean:

Wildcarded String	Meaning
*	any string
'tom'	the string *tom*
'harv'*	any string starting with *harv*
*'ing'	any string ending with *ing*
'a'*'b'	any string starting with *a* and ending with *b*
'a*b'	the string *a*b*
'jo'*'ph'*'ne'*'er'	any string starting with *jo*, and containing the substring *ph*, and which contains the substring *ne* in the portion of the string following *ph*, and which ends with *er*
%s*	any string starting with the supplied string
'bix'*%s	any string starting with *bix* and ending with the supplied string

String matches involving strings of different locales (code set, language, or territory) are resolved in an implementation-dependent way.

Extended Operations

In addition to the relational operators, extended operators can be specified. All extended operators return either TRUE or FALSE. A filter expression can contain both relational and extended operations.

Extended operators are specified using an identifier (**FN_identifier_t**) or a string. If the operator is specified using a string, the string is used to construct an identifier of format {FN_ID_STRING}. Identifiers of extended operators and signatures of the corresponding extended operations, as well as their suggested semantics, are registered with X/Open (Appendix G).

The following three extended operations are currently defined:

'name'(<Wildcarded String>)
: The identifier for this operation is 'name' ({FN_ID_STRING}). The argument to this operation is a wildcarded string. The operation returns TRUE if the name of the object matches the supplied wildcarded string.

'reftype'(%i)
: The identifier for this operation is 'reftype' ({FN_ID_STRING}). The argument to this operation is an identifier. The operation returns TRUE if the reference type of the object is equal to the supplied identifier.

'addrtype'(%i)
: The identifier for this operation is 'addrtype' ({FN_ID_STRING}). The argument to this operation is an identifier. The operation returns TRUE if any of the address types in the reference of the object is equal to the supplied identifier.

Support and exact semantics of extended operations are context-specific. If a context does not support an extended operation, or if the filter expression supplies the extended operation with either an incorrect number or type of arguments, the error [FN_E_SEARCH_INVALID_OP] is returned.[3]

3. [FN_E_OPERATION_NOT_SUPPORTED] is returned when *fn_attr_ext_search*( ) is not supported.

The following are examples of filter expressions that contain extended operations:

Expression	Meaning
'name'('bill'*)	evaluates to TRUE if the name of the object starts with *bill*
%i(%a, %v)	evaluates to result of applying the specified operation to the supplied arguments.
(%a == %v) and 'name'('joe'*)	evaluates to TRUE if the specified attribute has the given value and if the name of the object starts with *joe.*

RETURN VALUE

fn_search_filter_create( ) returns a pointer to an **FN_search_filter_t** object if the operation succeeds; otherwise it returns a NULL pointer.

ERRORS

fn_search_filter_create( ) returns a NULL pointer if the operation fails and sets *status* in the following way:

[FN_E_SEARCH_INVALID_FILTER]
The filter expression had a syntax error or some other problem.

[FN_E_SEARCH_INVALID_OP]
An operator in the filter expression is not supported or, if the operator is an extended operator, the number of types of arguments supplied does not match the signature of the operation.

[FN_E_INVALID_ATTR_IDENTIFIER]
The left hand side of an operator expression was not an attribute.

[FN_E_INVALID_ATTR_VALUE]
The right hand side of an operator expression was not an integer, attribute value, or (wildcarded) string.

Other status codes are possible as described in the reference manual pages for XFN_status_codes.

EXAMPLES

The following illustrates how to create three different filters.

The first example shows how to construct a filter involving substitution tokens and literals in the same filter expression. This example creates a filter for named objects whose ''color'' attribute contains a string value of ''red'', ''blue'' or ''white''. The first two values are specified using substitution tokens, the last value ''white'' is specified as a literal in the expression.

```
unsigned int status;
extern FN_attribute_t *attr_color;
FN_string_t *red = fn_string_from_str((unsigned char *)"red");
FN_string_t *blue = fn_string_from_str((unsigned char *)"blue");
FN_search_filter_t *sfilter;

sfilter = fn_search_filter_create(
    &status,
    "(%a == %s) or (%a == %s) or (%a == 'white')",
    attr_color, red, attr_color, blue,
    attr_color);
```

The second example illustrates how to construct a filter involving a wildcarded string. This example creates a filter for searching for named objects whose "last_name" attribute has a value that begins with the character ‘‘m’’:

```
unsigned int status;
extern FN_attribute_t *attr_last_name;
FN_search_filter_t *sfilter;
sfilter = fn_search_filter_create(
    &status, "%a == 'm'*", attr_last_name);
```

The third example illustrates how to construct a filter involving extended operations. This example creates a filter for finding all named objects whose name ends with ‘‘ton’’.

```
unsigned int status;
FN_search_filter_t *sfilter;
sfilter= fn_search_filter_create(&status, "'name'(*'ton')");
```

SEE ALSO

fn_attr_ext_search( ), *fn_ext_searchlist_next*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

FN_status_t — an XFN status object

SYNOPSIS

```
#include <xfn/xfn.h>

FN_status_t *fn_status_create(void);

void fn_status_destroy(FN_status_t *stat);

FN_status_t *fn_status_copy(const FN_status_t *stat);

FN_status_t *fn_status_assign(
    FN_status_t *dst, const FN_status_t *src);

unsigned int fn_status_code(const FN_status_t *stat);

const FN_composite_name_t *fn_status_remaining_name(
    const FN_status_t *stat);

const FN_composite_name_t *fn_status_resolved_name(
    const FN_status_t *stat);

const FN_ref_t *fn_status_resolved_ref(
    const FN_status_t *stat);

const FN_string_t* fn_status_diagnostic_message(
    const FN_status_t *stat);

unsigned int fn_status_link_code(const FN_status_t *stat);

const FN_composite_name_t *fn_status_link_remaining_name(
    const FN_status_t *stat);

const FN_composite_name_t *fn_status_link_resolved_name(
    const FN_status_t *stat);

const FN_ref_t *fn_status_link_resolved_ref(const FN_status_t *stat);

const FN_string_t* fn_status_link_diagnostic_message(
    const FN_status_t *stat);

int fn_status_is_success(const FN_status_t *stat);

int fn_status_set_success(FN_status_t *stat);

int fn_status_set(
    FN_status_t *stat,
    unsigned int code,
    const FN_ref_t *resolved_ref,
    const FN_composite_name_t *resolved_name,
    const FN_composite_name_t *remaining_name);
```

```
int fn_status_set_code(FN_status_t *stat, unsigned int code);

int fn_status_set_remaining_name(
    FN_status_t *stat,
    const FN_composite_name_t *name);

int fn_status_set_resolved_name(
    FN_status_t *stat,
    const FN_composite_name_t *name);

int fn_status_set_resolved_ref(
    FN_status_t *stat,
    const FN_ref_t *ref);

int fn_status_set_diagnostic_message(
    FN_status_t *stat,
    const FN_string_t *msg);

int fn_status_set_link_code(FN_status_t *stat, unsigned int code);

int fn_status_set_link_remaining_name(
    FN_status_t *stat,
    const FN_composite_name_t *name);

int fn_status_set_link_resolved_name(
    FN_status_t *stat,
    const FN_composite_name_t *name);

int fn_status_set_link_resolved_ref(
    FN_status_t *stat,
    const FN_ref_t *ref);

int fn_status_set_link_diagnostic_message(
    FN_status_t *stat,
    const FN_string_t *msg);

int fn_status_append_resolved_name(
    FN_status_t *stat,
    const FN_composite_name_t *name);

int fn_status_append_remaining_name(
    FN_status_t *stat,
    const FN_composite_name_t *name);

int fn_status_advance_by_name(
    FN_status_t *stat,
    const FN_composite_name_t *prefix,
    const FN_ref_t *resolved_ref);

FN_string_t *fn_status_description(
    const FN_status_t *stat,
    unsigned int detail,
```

```
    unsigned int *more_detail);
```

DESCRIPTION

The result status of operations in the context interface and the attribute interfaces is encapsulated in an **FN_status_t** object. The caller may supply a NULL pointer for this parameter, in which case, no status information is returned. If the caller supplies an **FN_status_t** object to the operation, upon return from the operation, this object will contain information about how the operation completed: whether an error occurred in performing the operation, the nature of the error, and information that helps locate where the error occurred. In the case that the error occurred while resolving an XFN link, the status object contains additional information about that error.

The context status object consists of several items of information.

primary status code
: An **unsigned int** code describing the disposition of the operation.

resolved name
: In the case of a failure during the resolution phase of the operation, this is the leading portion of the name that was resolved successfully. Resolution may have been successful beyond this point, but the error might not be pinpointed further.

resolved reference
: The reference to which resolution was successful (in other words, the reference to which the resolved name is bound).

remaining name
: The remaining unresolved portion of the name.

diagnostic message
: This contains any diagnostic message returned by the context implementation. This message provides the context implementation a way of notifying the end-user or administrator of any implementation-specific information related to the returned error status. The diagnostic message could then be used by the end-user or administrator to take appropriate out-of-band action to rectify the problem.

link status code
: In the case that an error occurred while resolving an XFN link, the primary status code has the value [FN_E_LINK_ERROR] and the link status code describes the error that occurred while resolving the XFN link.

resolved link name
: In the case of a link error, this contains the resolved portion of the name in the XFN link.

resolved link reference
: In the case of a link error, this contains the reference to which the resolved link name is bound.

remaining link name
: In the case of a link error, this contains the remaining unresolved portion of the name in the XFN link.

link diagnostic message
: In the case of a link error, this contains any diagnostic message related to the resolution of the link.

Both the primary status code and the link status code are values of type **unsigned int** that are drawn from the same set of meaningful values. XFN reserves the values 0 through 127 for standard meanings. The values and interpretations for the codes are determined by XFN (see the

reference manual page for XFN_status_code).

fn_status_create( ) creates a status object with status [FN_SUCCESS]. *fn_status_destroy*( ) releases the storage associated with *stat*; if *stat* is NULL, no action is taken. *fn_status_copy*( ) returns a copy of the status object *stat*. *fn_status_assign*( ) makes a copy of the status object *src* and assigns it to *dst*, releasing any old contents of *dst*. A pointer to the same object as *dst* is returned.

fn_status_code( ) returns the status code. *fn_status_remaining_name*( ) returns the remaining part of name to be resolved. *fn_status_resolved_name*( ) returns the part of the composite name that has been resolved. *fn_status_resolved_ref*( ) returns the reference to which resolution was successful. *fn_status_diagnostic_message*( ) returns any diagnostic message set by the context implementation.

fn_status_link_code( ) returns the link status code. *fn_status_link_remaining_name*( ) returns the remaining part of the link name that has not been resolved. *fn_status_link_resolved_name*( ) returns the part of the link name that has been resolved. *fn_status_link_resolved_ref*( ) returns the reference to which resolution of the link was successful. *fn_status_link_diagnostic_message*( ) returns any diagnostic message set by the context implementation during resolution of the link.

fn_status_is_success( ) returns 1 if the status indicates success, 0 otherwise.

fn_status_set_success( ) sets the status code to [FN_SUCCESS] and clears all other parts of *stat*. *fn_status_set*( ) sets the non-link contents of the status object *stat*. *fn_status_set_code*( ) sets the primary status code field of the status object *stat*. *fn_status_set_remaining_name*( ) sets the remaining name part of the status object *stat* to *name*. *fn_status_set_resolved_name*( ) sets the resolved name part of the status object *stat* to *name*. *fn_status_set_resolved_ref*( ) sets the resolved reference part of the status object *stat* to *ref*. *fn_status_set_diagnostic_message*( ) sets the diagnostic message part of the status object to *msg*.

fn_status_set_link_code( ) sets the link status code field of the status object *stat* to indicate why resolution of the link failed. *fn_status_set_link_remaining_name*( ) sets the remaining link name part of the status object *stat* to *name*. *fn_status_set_link_resolved_name*( ) sets the resolved link name part of the status object *stat* to *name*. *fn_status_set_link_resolved_ref*( ) sets the resolved link reference part of the status object *stat* to *ref*. *fn_status_set_link_diagnostic_message*( ) sets the link diagnostic message part of the status object to *msg*.

fn_status_append_resolved_name( ) appends as additional components *name* to the resolved name part of the status object *stat*. *fn_status_append_remaining_name*( ) appends as additional components *name* to the remaining name part of the status object *stat*. *fn_status_advance_by_name*( ) removes *prefix* from the remaining name, and appends it to the resolved name. The resolved reference part is set to *resolved_ref*. This operation returns 1 on success, 0 if the *prefix* is not a prefix of the remaining name.

fn_status_description( ) returns a string description of the given status object. It takes as argument an integer, *detail*, and a pointer to an integer, *more_detail*. Integer *detail* specifies the level of detail for which the description should be generated; the high the number, the more detail is to be provided. If *more_detail* is zero, it is ignored. If *more_detail* is non-zero, it is set by the description operation to indicate the next level of detail available, beyond that specified by *detail*. If no higher level of detail is available, *more_detail* is set to *detail*.

RETURN VALUE

The *fn_status_set_**( ) operations return 1 if the operation succeeds, 0 if the operation fails.

SEE ALSO

FN_composite_name_t, **FN_ref_t**, **FN_string_t**, XFN_status_codes, **<xfn/xfn.h>**.

NAME

FN_string_t — a character string

SYNOPSIS

```
#include <xfn/xfn.h>

FN_string_t *fn_string_create(void);

void fn_string_destroy(FN_string_t *str);

FN_string_t *fn_string_from_str(const unsigned char *cstr);

FN_string_t *fn_string_from_str_n(
    const unsigned char *cstr,
    size_t n);

const unsigned char *fn_string_str(
    const FN_string_t *str,
    unsigned int *status);

FN_string_t *fn_string_from_contents(
    unsigned long code_set,
    unsigned long lang_terr,
    size_t charcount,
    size_t bytecount,
    const void *contents,
    unsigned int *status);

unsigned long  fn_string_code_set(const FN_string_t *str);

unsigned long  fn_string_lang_terr(const FN_string_t *str);

size_t fn_string_charcount(const FN_string_t *str);

size_t fn_string_bytecount(const FN_string_t *str);

const void *fn_string_contents(const FN_string_t *str);

FN_string_t *fn_string_copy(const FN_string_t *str);

FN_string_t *fn_string_assign(
    FN_string_t *dst, const FN_string_t *src);

FN_string_t *fn_string_from_strings(
    unsigned int *status,
    const FN_string_t *s1,
    const FN_string_t *s2, ...);

FN_string_t *fn_string_from_substring(
    const FN_string_t *str,
    int first,
    int last);
```

```
int fn_string_is_empty(const FN_string_t *str);

int fn_string_compare(
    const FN_string_t *str1,
    const FN_string_t *str2,
    unsigned int string_case,
    unsigned int *status);

int fn_string_compare_substring(
    const FN_string_t *str1,
    int first,
    int last,
    const FN_string_t *str2,
    unsigned int string_case,
    unsigned int *status);

int fn_string_next_substring(
    const FN_string_t *str,
    const FN_string_t *sub,
    int index,
    unsigned int string_case,
    unsigned int *status);

int fn_string_prev_substring(
    const FN_string_t *str,
    const FN_string_t *sub,
    int index,
    unsigned int string_case,
    unsigned int *status);
```

DESCRIPTION

The **FN_string_t** type is used to represent character strings in the XFN interface. It provides insulation from specific string representations.

The **FN_string_t** supports multiple locales. It provides creation functions for character strings of the current locale and a generic creation function for arbitrary locales. A locale is identified by its code set, language and territory. The degree of support for the functions that manipulate **FN_string_t** for arbitrary locales is implementation-dependent. An XFN implementation is required to support the ''C'' locale. The ''C'' locale's repertoire of characters is restricted to that defined by the portable representation of ISO 646 (same encoding as ASCII). Support for other locales is optional.

fn_string_destroy( ) releases the storage associated with the given string. If the argument to *fn_string_destroy*( ) is NULL, no action is taken.

fn_string_create( ) creates an empty string.

fn_string_from_str( ) creates an **FN_string_t** object from the given null terminated string based on the code set of the current locale setting. The number of characters in the string is determined by the code set of the current locale setting. *fn_string_from_str_n*( ) is like *fn_string_from_str*( ) except only *n* characters from the given string are used. *fn_string_str*( ) returns the contents of the given string *str* in the form of a null terminated string in the code set and current locale setting.

fn_string_from_contents( ) creates an **FN_string_t** object using the specified code set *code_set* language and territory *lang_terr* and data in the given buffer *contents*. *bytecount* specifies the number of bytes in *contents* and *charcount* specifies the number of characters represented by *contents*.

fn_string_code_set( ) returns the code set associated with the given string object. *fn_string_lang_terr*( ) returns the language and territory associated with the given string object. *fn_string_charcount*( ) returns the number of characters in the given string object. *fn_string_bytecount*( ) returns the number of bytes used to represent the given string object. *fn_string_contents*( ) returns a pointer to the contents of the given string object.

fn_string_copy( ) returns a copy of the given string object. *fn_string_assign*( ) makes a copy of the string object *src* and assigns it to *dst*, releasing any old contents of *dst*. A pointer to the same object as *dst* is returned. *fn_string_from_strings*( ) is a function that takes a variable number of arguments (minimum of 2), the last of which must be NULL (0); it returns a new string object composed of the left to right concatenation of the given strings, in the given order. The support for strings with different code sets and/or locales as arguments to a single invocation of *fn_string_from_strings*( ) is implementation-dependent. *fn_string_from_substring*( ) returns a new string object consisting of the characters located between *first* and *last* inclusive from *str*. Indexing begins with 0. If *last* is {FN_STRING_INDEX_LAST} or exceeds the length of the string, the index of the last character of the string is used.

fn_string_is_empty( ) returns whether *str* is an empty string.

Comparison of two strings must take into account code set and locale information. If strings are in the same code set and same locale, case sensitivity is applied according to the case sensitivity rules applicable for the code set and locale; case sensitivity may not necessarily be relevant for all string encodings. If *string_case* is non-zero, case is significant and equality for strings of the same code set is defined as equality between byte-wise encoded values of the strings. If *string_case* is zero, case is ignored and equality for strings of the same code set is defined using the definition of case-insensitive equality for the specific code set. Support for comparison between strings of different code sets, or lack thereof, is implementation-dependent.

fn_string_compare( ) compares strings *str1* and *str2* and returns 0 if they are equal, non-zero if they are not equal. If two strings are not equal, *fn_string_compare*( ) returns a positive value if the difference of *str2* precedes that of *str1* in terms of byte-wise encoded value (with case-sensitivity taken into account when *string_case* is non-zero), and a negative value if the difference of *str1* precedes that of *str2*, in terms of byte-wise encoded value (with case-sensitivity taken into account when *string_case* is non-zero). Such information (positive versus negative return value) may be used by applications that use strings of code sets in which ordering is meaningful; this information is not of general use in internationalized environments. *fn_string_compare_substring*( ) is similar to *fn_string_compare*( ) except *fn_string_compare_substring*( ) compares characters between *first* and *last* inclusive of *str2* with *str1*. Comparison of strings with incompatible code sets returns a negative or positive value (never 0) depending on the implementation.

fn_string_next_substring( ) returns the index of the next occurrence of *sub* at or after *index* in the string *str*. {FN_STRING_INDEX_NONE} is returned if *sub* does not occur. *fn_string_prev_substring*( ) returns the index of the previous occurrence of *sub* at or before *index* in the string *str*. {FN_STRING_INDEX_NONE} is returned if *sub* does not occur. In both of these functions, *string_case* specifies whether the search should take case-sensitivity into account.

ERRORS

fn_string_str( ) returns 0 and sets *status* to [FN_E_INCOMPATIBLE_CODE_SETS] if the given string's representation cannot be converted into the code set of the current locale setting. It is implementation-dependent which code sets can be converted into the code set of the current

locale.

Locale (code set, language, or territory) mismatches that occur during concatenation, searches, or comparisons are resolved in an implementation-dependent way. When an implementation discovers that arguments to substring searches and comparison operations have incompatible code sets, it sets status to [FN_E_INCOMPATIBLE_CODE_SETS]. When an implementation discovers that the arguments have incompatible language or territory locale information, it sets status to [FN_E_INCOMPATIBLE_LOCALES]. In such cases, *fn_string_from_strings*( ) returns 0. The returned value for comparison operations when there is such incompatibilities is either negative or positive (greater than 0); it is never 0.

Function *fn_string_from_contents*( ) returns 0 and sets status to an appropriate error code ([FN_E_INCOMPATIBLE_CODE_SETS] or [FN_E_INCOMPATIBLE_LOCALES]) if the code set or locale of the given string object is not supported by the XFN implementation.

The *status* argument to the following functions may be NULL, in which case no status is set upon return of the function:
fn_string_from_contents( ),
fn_string_str( ),
fn_string_from_strings( ),
fn_string_compare( ),
fn_string_compare_substring( ),
fn_string_next_substring( ),
fn_string_prev_substring( ).

SEE ALSO

<xfn/xfn.h>.

NAME

fn_attr_bind — bind a reference to a name and associate attributes with named object

SYNOPSIS

```
#include <xfn/xfn.h>

int fn_attr_bind(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    const    FN_ref_t *ref,
    const    FN_attrset_t *attrs,
    unsigned int exclusive,
    FN_status_t *status);
```

DESCRIPTION

This operation binds the supplied reference *ref* to the supplied composite name *name* relative to *ctx*, and associates the attributes specified in *attrs* with the named object. The binding is made in the target context — that context named by all but the terminal atomic part of *name*. The operation binds the terminal atomic name to the supplied reference in the target context. The target context must already exist.

The value of *exclusive* determines what happens if the terminal atomic part of the name is already bound in the target context. If *exclusive* is nonzero and *name* is already bound, the operation fails. If *exclusive* is zero, the new binding replaces any existing binding, and, if *attrs* is not NULL, *attrs* replaces any existing attributes associated with the named object. If *attrs* is NULL and *exclusive* is zero, any existing attributes associated with the named object are left unchanged.

RETURN VALUE

fn_attr_bind( ) returns 1 upon success, 0 upon failure.

ERRORS

fn_attr_bind( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes. Of special relevance for this operation is the following status code:

[FN_E_NAME_IN_USE]
The supplied name is already in use.

APPLICATION USAGE

The value of *ref* cannot be NULL. If the intent is to reserve a name using *fn_attr_bind*( ), a reference containing no address should be supplied. This reference may be name service-specific or it may be the conventional NULL reference defined in Appendix G.

If multiple sources are updating a reference or attributes associated with a named object, they must synchronize amongst each other when adding, modifying, or removing from the address list of a bound reference, or manipulating attributes associated with the named object.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_ref_t**, **FN_status_t**, *fn_ctx_lookup*( ), *fn_ctx_bind*( ), *fn_ctx_unbind*( ), XFN_attribute_operations, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_attr_create_subcontext — create a subcontext in a context and associate attributes with newly created context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ref_t *fn_attr_create_subcontext(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrset_t *attrs,
    FN_status_t *status);
```

DESCRIPTION

This operation creates a new XFN context of the same type as the target context — that named by all but the terminal atomic component of *name* — and binds it to the supplied composite name. In addition, attributes given in *attrs* are associated with the newly created context.

The target context must already exist. The new context is created and bound in the target context using the terminal atomic name in *name*. The operation returns a reference to the newly created context.

RETURN VALUE

fn_attr_create_subcontext( ) returns a reference to the newly created context; if the operation fails, it returns a NULL pointer (0).

ERRORS

fn_attr_create_subcontext( ) sets *status* as described in the reference manual pages for **FN_status_t** and **XFN_status_codes**. Of special relevance for this operation is the following status code:

[FN_E_NAME_IN_USE]
The terminal atomic name already exists in the target context.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_ref_t**, **FN_status_t**, *fn_attr_bind*( ), *fn_ctx_bind*( ), *fn_ctx_create_subcontext*( ), *fn_ctx_lookup*( ), *fn_ctx_destroy*( ), XFN_status_codes, XFN_attribute_operations, **<xfn/xfn.h>**.

NAME

fn_attr_ext_search, FN_ext_searchlist_t, fn_ext_searchlist_next, fn_ext_searchlist_destroy — search for names in the specified context(s) whose attributes satisfy the filter

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ext_searchlist_t *fn_attr_ext_search(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_search_control_t *control,
    const FN_search_filter_t *filter,
    FN_status_t *status);

FN_composite_name_t *fn_ext_searchlist_next(
    FN_ext_searchlist_t *esl,
    FN_ref_t **returned_ref,
    FN_attrset_t **returned_attrs,
    FN_status_t *status);

void fn_ext_searchlist_destroy(
    FN_ext_searchlist_t *esl);
```

DESCRIPTION

This set of operations is used to list names of objects whose attributes satisfy the filter expression. The references to which these names are bound and specified attributes and their values may also be returned.

control encapsulates the option settings for the search. These options are:

- the scope of the search
- whether XFN links are followed
- a limit on the number of names returned
- whether references and specific attributes associated with the named objects that satisfy the filter are returned.

The scope of the search is one of:

- the object named *name* relative to the context *ctx*
- the context named *name* relative to the context *ctx*
- the context named *name* relative to the context *ctx*, and its subcontexts

  or:

- the context named *name* relative to the context *ctx*, and a context implementation-defined set of subcontexts.

If the value of *control* is 0, default control option settings are used. The default settings are:

- scope is search named context
- links are not followed
- all names of objects that satisfy the filter are returned

- references and attributes are not returned.

The **FN_search_control_t** type is described in the reference manual page for **FN_search_control_t**.

The filter expression *filter* in *fn_attr_ext_search*( ) is evaluated against the attributes of the objects bound in the scope of the search. The filter evaluates to either TRUE or FALSE. The names and, optionally, the references and attributes of objects whose attributes satisfy the filter are enumerated. If the value of *filter* is 0, all names within the search scope are enumerated. The **FN_search_filter_t** type is described in the reference manual page for **FN_search_filter_t**.

The call to *fn_attr_ext_search*( ) initiates the search process. It returns a handle to an **FN_ext_searchlist_t** object that is used to enumerate the names of the objects that satisfy the filter.

The operation *fn_ext_searchlist_next*( ) returns the next name in the enumeration identified by *esl*; it also updates *esl* to indicate the state of the enumeration. If the reference to which the name is bound was requested, it is returned in *returned_ref*. Requested attributes associated with the name are returned in *returned_attrs*; each attribute consists of an attribute identifier, syntax and value(s). Successive calls to *fn_ext_searchlist_next*( ) using *esl* return successive names and, optionally their references and attributes, in the enumeration and further update the state of the enumeration.

The names that are returned are composite names, to be resolved relative to the starting context for the search. This starting context is the context named *name* relative to *ctx* unless the scope of the search is only the named object. If the scope of the search is only the named object, the terminal atomic name in *name* is returned.

fn_ext_searchlist_destroy( ) releases resources used during the enumeration. This may be invoked at any time to terminate the enumeration. If the argument to *fn_ext_searchlist_destroy*( ) is NULL, no action is taken.

RETURN VALUE

fn_attr_ext_search( ) returns a pointer to an **FN_ext_searchlist_t** object if the search is successfully initiated; it returns a NULL pointer if the search cannot be initiated or if no named object with attributes whose values satisfy the filter expression is found.

fn_ext_searchlist_next( ) returns a pointer to an **FN_composite_name_t** object that is the next name in the enumeration; it returns a NULL pointer if no more names can be returned. If *returned_attrs* is a NULL pointer, no attributes are returned; otherwise, *returned_attrs* contains the attributes associated with the named object, as specified in the control parameter to *fn_attr_ext_search*. If *returned_ref* is a NULL pointer, no reference is returned; otherwise, if *control* specified the return of the reference of the named object, that reference is returned in *returned_ref*.

In the case of a failure, these operations return in the *status* argument a code indicating the nature of the failure.

ERRORS

If successful, *fn_attr_ext_search*( ) returns a pointer to an **FN_ext_searchlist_t** object and sets *status* to [FN_SUCCESS].

fn_attr_ext_search( ) returns a NULL pointer when no more names can be returned. *status* is set in the following way:

[FN_SUCCESS]
A named object could not be found whose attributes satisfied the filter expression.

[FN_E_NOT_A_CONTEXT]
The object named for the start of the search was not a context and the search scope was the

given context or the given context and its subcontexts.

[FN_E_SEARCH_INVALID_FILTER]
The filter could not be evaluated TRUE or FALSE, or there was some other problem with the filter.

[FN_E_SEARCH_INVALID_OPTION]
A supplied search control option could not be supported.

[FN_E_SEARCH_INVALID_OP]
An operator in the filter expression is not supported or, if the operator is an extended operator, the number of types of arguments supplied does not match the signature of the operation.

[FN_E_ATTR_NO_PERMISSION]
The caller did not have permission to read one or more of the attributes specified in the filter.

[FN_E_INVALID_ATTR_VALUE]
A value type in the filter did not match the syntax of the attribute against which it was being evaluated.

Other status codes are possible as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

Each successful call to *fn_ext_searchlist_next*( ) returns a name and, optionally, its reference in *returned_ref* and requested attributes in *returned_attrs*. *status* is set in the following way:

[FN_SUCCESS]
All requested attributes were returned successfully with the name.

[FN_E_ATTR_NO_PERMISSION]
The caller did not have permission to read one or more of the requested attributes.

[FN_E_INVALID_ATTR_IDENTIFIER]
A requested attribute identifier was not in a format acceptable to the naming system, or its contents was not valid for the format specified.

[FN_E_NO_SUCH_ATTRIBUTE]
The named object did not have one of the requested attributes.

[FN_E_INSUFFICIENT_RESOURCES]
Insufficient resources are available to return all the requested attributes and their values.

The status codes [FN_E_ATTR_NO_PERMISSION], [FN_E_INVALID_ATTR_IDENTIFIER], [FN_E_NO_SUCH_ATTRIBUTE] and [FN_E_INSUFFICIENT_RESOURCES] indicate that some of the requested attributes may have been returned in *returned_attrs* but one or more of them could not be returned. Use *fn_attr_get*( ) or *fn_attr_multi_get*( ) to discover why these attributes could not be returned.

If *fn_ext_searchlist_next*( ) returns a name, it can be called again to get the next name in the enumeration.

fn_ext_searchlist_next( ) returns a NULL pointer if no more names can be returned. *status* is set in the following way:

[FN_SUCCESS]
The search has completed successfully.

[FN_E_PARTIAL_RESULT]
The enumeration is not yet complete but cannot be continued.

[FN_E_ATTR_NO_PERMISSION]
The caller did not have permission to read one or more of the attributes specified in the filter.

[FN_E_INVALID_ENUM_HANDLE]
The supplied enumeration handle was not valid. Possible reasons could be that the handle was from another enumeration, or the context being enumerated no longer accepts the handle (due to such events as handle expiration or updates to the context).

Other status codes are possible as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

EXAMPLE

The following code fragment illustrates how the *fn_attr_ext_search*( ) operation may be used. The code consists of three parts, preparing the arguments for the search, performing the search, and cleaning up.

The first part involves getting the name of the context to start the search and constructing the search filter that named objects in the context must satisfy. This is done in the declarations part of the code and by the routine *get_search_query*( ). See the reference manual page for **FN_search_filter_t** for the description of *sfilter* and the filter creation operation.

The next part involves doing the search and enumerating the results of the search. This is done by first getting a context handle to the Initial Context, and then passing that handle along with the name of the target context and search filter to *fn_attr_ext_search*( ). This particular call to *fn_attr_ext_search*( ) uses the default search control options (by passing in 0 as the *control* argument). This means that the search will be performed in the context named by *target_name* and that no reference or attributes will be returned. In addition, any XFN links encountered will not be followed and all named objects that satisfy the search filter will be returned (that is, no limit). If successful, *fn_attr_ext_search*( ) returns *esl*, a handle for enumerating the results of the search. The results of the search are enumerated using calls to *fn_ext_searchlist_next*( ), which returns the name of the object. (The arguments *returned_ref* and *returned_attrs* to *fn_ext_searchlist_next*( ) are 0 because the default search control used in *fn_attr_ext_search*( ) did not request them to be returned.)

The last part of the code involves cleaning up the resources used during the search and enumeration. The call to *fn_ext_searchlist_destroy*( ) releases resources reserved for this enumeration. The other calls release the context handle, name, filter and status objects created earlier.

```
FN_ctx_t *ctx;
FN_ext_searchlist_t *esl;
FN_composite_name_t *name;
FN_status_t *status = fn_status_create();
FN_composite_name_t *target_name = get_name_from_user_input();
FN_search_filter_t *sfilter = get_search_query();

ctx = fn_ctx_handle_from_initial(0, status);
/* error checking on ‘status’ */

if ((esl=fn_attr_ext_search(ctx, target_name,
    /* default controls */ 0, sfilter, status)) == 0) {
    /* report ‘status’, cleanup, and exit */
}

while (name=fn_ext_searchlist_next(esl, 0, 0, status)) {
```

```
        /* do something with 'name' */
        fn_composite_destroy(name);
    }

    /* check 'status' for reason for end of enumeration */

    /* Clean up */
    fn_ext_searchlist_destroy(esl);
    fn_search_filter_destroy(sfilter);
    fn_ctx_handle_destroy(ctx);
    fn_composite_name_destroy(target_name);
    fn_status_destroy(status);

    /*
     * Procedure for constructing the filter object for search:
     *     "age" attribute is greater than or equal to 17
     *     AND less than or equal to 25
     *     AND the "student" attribute is present.
     */

    FN_search_filter_t *
    get_search_query()
    {
        extern FN_attribute_t *attr_age;
        extern FN_attribute_t *attr_student;
        FN_search_filter_t *sfilter;
        unsigned int filter_status;

        sfilter = fn_search_filter_create(
            &filter_status,
            "(%a >= 17) and (%a <= 25) and %a",
            attr_age, attr_age, attr_student);

        /* error checking on 'filter_status' */

        return (sfilter);
    }
```

APPLICATION USAGE

The search performed by *fn_attr_ext_search*( ) is not ordered in any way, including the traversal of subcontexts. The names enumerated using *fn_ext_searchlist_next*( ) are not ordered in any way. Furthermore, there is no guarantee that any two series of enumerations with the same arguments to *fn_attr_ext_search*( ) will return the names in the same order.

XFN links encountered during the resolution of *name* are followed, regardless of the follow links control setting, and the search starts at the final named object or context.

If *control* specifies that the search should follow links, XFN link names encountered during the search are followed and the terminal named object is searched. If the terminal named object is bound to a context and the scope of the search includes subcontexts, that context and its subcontexts are also searched. For example, if *aname* is bound to an XFN link, *lname*, in a context within the scope of the search, and *aname* is returned by *fn_ext_searchlist_next*( ), this means that the object identified by *lname* satisfied the filter expression. *aname* is returned instead of *lname* because *aname* can always be named relative to the starting context for the search.

If *control* specifies that the search should not follow links, the attributes associated with the names of XFN links are searched. For example, if *aname* is bound to an XFN link, *lname*, in a context within the scope of the search, and *aname* is returned by *fn_ext_searchlist_next*( ), this means that the object identified by *aname* satisfied the filter expression.

When following XFN links, *fn_attr_ext_search*( ) may search contexts outside of *scope*. In addition, if the link name's terminal atomic name is bound in a context within *scope*, the operation may return the same object more than once.

XFN does not specify how *control* affects the following of native naming system links during the search.

SEE ALSO

FN_attrset_t, FN_composite_name_t, FN_ctx_t, FN_ref_t, FN_search_control_t, FN_search_filter_t, FN_status_t, *fn_attr_get*( ), *fn_attr_multi_get*( ), *fn_attr_search*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_attr_get — return specified attribute associated with name

SYNOPSIS

```
#include <xfn/xfn.h>

FN_attribute_t *fn_attr_get(
    FN_ctx_t       *ctx,
    const           FN_composite_name_t *name,
    const           FN_identifier_t *attribute_id,
    unsigned int    follow_link,
    FN_status_t    *status);
```

DESCRIPTION

This operation returns the identifier, syntax and values of a specified attribute for the object named *name* relative to *ctx*. If *name* is empty, the attribute associated with *ctx* is returned.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed and the attribute associated with the final named object is returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

RETURN VALUE

fn_attr_get( ) returns a pointer to an **FN_attribute_t** object if the operation succeeds; it returns a NULL pointer (0) if the operation fails.

ERRORS

fn_attr_get( ) sets *status* as described in the reference manual page for **FN_status_t** and XFN_status_codes.

APPLICATION USAGE

fn_attr_get_values( ) and its related operations are used for getting individual values of an attribute. They should be used if the combined size of all the values are expected to be too large to be returned in a single invocation of *fn_attr_get*( ).

SEE ALSO

FN_attribute_t, **FN_composite_name_t**, **FN_ctx_t**, **FN_identifier_t**, **FN_status_t**, *fn_attr_get_values*( ), XFN_attribute_operations, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_attr_get_ids — get a list of the identifiers of all attributes associated with named object

SYNOPSIS

```
#include <xfn/xfn.h>
FN_attrset_t *fn_attr_get_ids(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    unsigned int follow_link,
    FN_status_t *status);
```

DESCRIPTION

This operation returns a list of the attribute identifiers of all attributes associated with the object named by *name* relative to the context *ctx*. If *name* is empty, the attribute identifiers associated with *ctx* are returned.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed and the identifiers of the attributes associated with the final named object are returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

RETURN VALUE

This operation returns a pointer to an object of type **FN_attrset_t**; if the operation fails, a NULL pointer (0) is returned.

ERRORS

This operation sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

APPLICATION USAGE

The attributes in the returned set do not contain the syntax or values of the attributes, only their identifiers.

SEE ALSO

FN_attrset_t, **FN_attribute_t**, **FN_composite_name_t**, **FN_ctx_t**, **FN_status_t**, *fn_attr_get*( ), *fn_attr_multi_get*( ), XFN_attribute_operations, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_attr_get_values, FN_valuelist_t, fn_valuelist_next, fn_valuelist_destroy, — return values of an attribute

SYNOPSIS

```
#include <xfn/xfn.h>

FN_valuelist_t *fn_attr_get_values(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    const     FN_identifier_t *attribute_id,
    unsigned int follow_link,
    FN_status_t *status);

FN_attrvalue_t *fn_valuelist_next(
    FN_valuelist_t *vl,
    FN_identifier_t **attr_syntax,
    FN_status_t *status);

void fn_valuelist_destroy(
    FN_valuelist_t *vl);
```

DESCRIPTION

This set of operations is used to obtain the values of a single attribute, identified by *attribute_id*, associated with the object named *name*, resolved in the context *ctx*. If *name* is empty, the attribute values associated with *ctx* are obtained.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed and the values of the attribute associated with the final named object are returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

The operation *fn_attr_get_values*( ) initiates the enumeration process. It returns a handle to an **FN_valuelist_t** object that can be used to enumerate the values of the specified attribute.

The operation *fn_valuelist_next*( ) returns a new **FN_attrvalue_t** object containing the next value in the attribute and may be called multiple times until all values are retrieved. The syntax of the attribute is returned in *attr_syntax*.

The operation *fn_valuelist_destroy*( ) is used to release the resources used during the enumeration. This may be invoked before the enumeration has completed to terminate the enumeration. If the argument to *fn_valuelist_destroy*( ) is NULL, no action is taken.

These operations work in a similar fashion as the *fn_ctx_list_names*( ) operations.

RETURN VALUE

fn_attr_get_values( ) returns a pointer to an **FN_valuelist_t** object if the enumeration process is successfully initiated; it returns a NULL pointer if the process failed.

fn_valuelist_next( ) returns a NULL pointer if no more attribute value can be returned.

In the case of a failure, these operations set *status* to indicate the nature of the failure.

ERRORS

Each successful call to *fn_valuelist_next*( ) returns an attribute value. *status* is set to [FN_SUCCESS].

When *fn_valuelist_next*( ) returns a NULL pointer, it indicates that no more values can be returned. *status* is set in the following way:

[FN_SUCCESS]
The enumeration has completed successfully.

[FN_E_INVALID_ENUM_HANDLE]
The given enumeration handle is not valid. Possible reasons could be that the handle was from another enumeration, or the context being enumerated no longer accepts the handle (due to such events as handle expiration or updates to the context).

[FN_E_PARTIAL_RESULT]
The enumeration is not yet complete but cannot be continued.

In addition to these status codes, other status codes are also possible in calls to these operations. In such cases, *status* is set as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

APPLICATION USAGE

This interface should be used instead of *fn_attr_get*( ) if the combined size of all the values is expected to be too large to be returned by *fn_attr_get*( ).

There may be a relationship between the *ctx* argument supplied to *fn_attr_get_values*( ) and the **FN_valuelist_t** object it returns. For example, some implementations may store the context handle *ctx* within the **FN_valuelist_t** object for subsequent *fn_valuelist_next*( ) calls. In general, a *fn_ctx_handle_destroy*( ) should not be invoked on *ctx* until the enumeration has terminated.

SEE ALSO

FN_attribute_t, **FN_attrvalue_t**, **FN_composite_name_t**, **FN_ctx_t**, **FN_identifier_t**, **FN_status_t**, *fn_attr_get*( ), *fn_ctx_list_names*( ), XFN_attribute_operations, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_attr_modify — modify specified attribute associated with name

SYNOPSIS

```
#include <xfn/xfn.h>

int fn_attr_modify(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    unsigned int mod_op,
    const    FN_attribute_t *attr,
    unsigned int follow_link,
    FN_status_t *status);
```

DESCRIPTION

This operation modifies according to *mod_op* the attribute *attr* associated with the object named *name* relative to *ctx*. If *name* is empty, the attribute associated with *ctx* is modified.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed and the attribute associated with the final named object is modified; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

The modification is made on the attribute identified by the attribute identifier of *attr*. The syntax and values of *attr* are use according to the modification operation.

The modification operations are described in the following table.

Operation Code	Meaning
FN_ATTR_OP_ADD	Add an attribute with given attribute identifier and set of values. If an attribute with this identifier already exists, replace the set of values with those in the given set. The set of values may be empty if the target naming system permits.
FN_ATTR_OP_ADD_EXCLUSIVE	Add an attribute with the given attribute identifier and set of values. The operation fails with [FN_E_ATTR_IN_USE] if an attribute with this identifier already exists. The set of values may be empty if the target naming system permits.
FN_ATTR_OP_REMOVE	Remove the attribute with the given attribute identifier and all of its values. The operation succeeds even if the attribute does not exist. The values of the attribute supplied with this operation are ignored.
FN_ATTR_OP_ADD_VALUES	Add the given values to those of the given attribute (resulting in the attribute having the union of its prior value set with the set given). Create the attribute if it does not exist already. The set of values may be empty if the target naming system permits.
FN_ATTR_OP_REMOVE_VALUES	Remove the given values from those of the given attribute (resulting in the attribute having the set difference of its prior value set and the set given). This succeeds even if some of the given values are not in the set of values that the attribute has. In naming systems that require an attribute to have at least one value, removing the last value will remove the attribute as well.

RETURN VALUE

This operation returns 1 if the operation succeeds, 0 if the operation fails.

ERRORS

fn_attr_modify( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_attribute_t**, **FN_status_t**, *fn_attr_multi_modify*( ), XFN_attribute_operations, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_attr_multi_get, FN_multigetlist_t, fn_multigetlist_next, fn_multigetlist_destroy — return multiple attributes associated with named object

SYNOPSIS

```
#include <xfn/xfn.h>

FN_multigetlist_t *fn_attr_multi_get(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    const     FN_attrset_t *attr_ids,
    unsigned int follow_link,
    FN_status_t *status);

FN_attribute_t *fn_multigetlist_next(
    FN_multigetlist_t *ml,
    FN_status_t *status);

void fn_multigetlist_destroy(
    FN_multigetlist_t *ml);
```

DESCRIPTION

This set of operations returns one or more attributes associated with the object named by *name* relative to the context *ctx*. If *name* is empty, the attributes associated with *ctx* are returned.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed and attributes associated with the final named object are returned; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

The attributes returned are those specified in *attr_ids*. If the value of *attr_ids* is 0, all attributes associated with the named object are returned. Any attribute values in *attr_ids* provided by the caller are ignored; only the attribute identifiers are relevant for this operation. Each attribute (identifier, syntax, values) is returned one at a time using an enumeration scheme similar to that for listing a context.

fn_attr_multi_get( ) initiates the enumeration process. It returns a handle to an **FN_multigetlist_t** object that can be used for the enumeration.

The operation *fn_multigetlist_next*( ) returns a new **FN_attribute_t** object containing the next attribute (identifiers, syntaxes, and values) requested and updates *ml* to indicate the state of the enumeration.

The operation *fn_multigetlist_destroy*( ) releases the resources used during the enumeration. It may be invoked before the enumeration has completed to terminate the enumeration. If the argument to *fn_multigetlist_destroy*( ) is NULL, no action is taken.

RETURN VALUE

fn_attr_multi_get( ) returns a pointer to an **FN_multigetlist_t** object if the enumeration has been initiated successfully; a NULL pointer is returned if it failed.

fn_multigetlist_next( ) returns a pointer to an **FN_attribute_t** object if an attribute was returned, a NULL pointer (0) if no attribute was returned.

In the case of a failure, these operations set *status* to indicate the nature of the failure.

ERRORS

Each call to *fn_multigetlist_next*( ) sets status as follows:

[FN_SUCCESS]
If an attribute was returned, there are more attributes to be enumerated. If no attribute was returned, the enumeration has completed successfully.

[FN_E_ATTR_NO_PERMISSION]
The caller did not have permission to read this attribute.

[FN_E_INSUFFICIENT_RESOURCES]
Insufficient resources are available to return the attribute's values.

[FN_E_INVALID_ATTR_IDENTIFIER]
This attribute identifier was not in a format acceptable to the naming system, or its contents was not valid for the format specified for the identifier.

[FN_E_INVALID_ENUM_HANDLE]
(No attribute should be returned with this status code). The given enumeration handle is not valid. Possible reasons could be that the handle was from another enumeration, or the object being processed no longer accepts the handle (due to such events as handle expiration or updates to the object's attribute set).

[FN_E_NO_SUCH_ATTRIBUTE]
The object did not have an attribute with the given identifier.

[FN_E_PARTIAL_RESULT]
(No attribute should be returned with this status code). The enumeration is not yet complete but cannot be continued.

For
[FN_E_ATTR_NO_PERMISSION],
[FN_E_INVALID_ATTR_IDENTIFIER],
[FN_E_INSUFFICIENT_RESOURCES],
[FN_E_NO_SUCH_ATTRIBUTE],
the returned attribute contains only the attribute identifier (no value or syntax). For these four status codes and [FN_SUCCESS] (when an attribute was returned), *fn_multigetlist_next*( ) can be called again to return another attribute. All other status codes indicate that no more attributes can be returned by *fn_multigetlist_next*( ).

Other status codes, such as [FN_E_COMMUNICATION_FAILURE], are also possible, in which case, no attribute is returned. In such cases, *status* is set as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

EXAMPLES

The following code fragment illustrates how to obtain all attributes associated with a given name using the *fn_attr_multi_get*( ) operations.

```
/* list all attributes associated with given name */

extern FN_string_t *input_string;
FN_ctx_t  *ctx;
FN_composite_name_t *target_name = fn_composite_name_from_string(
    input_string);
FN_multigetlist_t *ml;
FN_status_t  *status = fn_status_create();
FN_attribute_t *attr;
int done = 0;
```

```
ctx = fn_ctx_handle_from_initial(0, status);
/* error checking on 'status' */

/* attr_ids == 0 indicates all attributes are to be returned  */
/* follow_link == 1 means if terminal atom is link, follow it */
if ((ml=fn_attr_multi_get(ctx, target_name, 0, 1, status)) == 0) {
    /* report 'status' and exit */
}

while ((attr=fn_multigetlist_next(ml, status)) && !done) {
    switch (fn_status_code(status)) {
    case FN_SUCCESS:
        /* do something with 'attr' */
        break;
    case FN_E_ATTR_NO_PERMISSION:
    case FN_E_INVALID_ATTR_IDENTIFIER:
    case FN_E_NO_SUCH_ATTRIBUTE:
        /* report error using identifier in 'attr' */
        break;
    default:
        /* other error handling */
        done = 1;
    }
    if (attr)
        fn_attribute_destroy(attr);
}

/* check 'status' for reason for end of enumeration */
/* and report if necessary                          */

/* clean up */
fn_multigetlist_destroy(ml);

}
```

APPLICATION USAGE

Implementations are not required to return all attributes requested by *attr_ids*. Some may choose to return only the attributes found successfully, followed by a status of [FN_E_PARTIAL_RESULT]; such implementations may not necessarily return attributes identifying those that could not be read. Implementations are not required to return the attributes in any order.

There may be a relationship between the *ctx* argument supplied to *fn_attr_multi_get*( ) and the **FN_multigetlist_t** object it returns. For example, some implementations may store the context handle *ctx* within the **FN_multigetlist_t** object for subsequent *fn_multigetlist_next*( ) calls. In general, a *fn_ctx_handle_destroy*( ) should not be invoked on *ctx* until the enumeration has terminated.

SEE ALSO

FN_attrset_t, **FN_attribute_t**, **FN_composite_name_t**, **FN_ctx_t**, **FN_identifier_t**, **FN_status_t**, *fn_attr_get*( ), *fn_ctx_list_names*( ), XFN_attribute_operations, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_attr_multi_modify — modify multiple attributes associated with named object

SYNOPSIS

```
#include <xfn/xfn.h>

int fn_attr_multi_modify(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    const    FN_attrmodlist_t *mods,
    unsigned int follow_link,
    FN_attrmodlist_t **unexecuted_mods,
    FN_status_t *status);
```

DESCRIPTION

This operation modifies the attributes associated with the object named *name* relative to *ctx*. If *name* is empty, the attributes associated with *ctx* are modified.

The value of *follow_link* determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed and the attributes associated with the final named object are modified; if *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

In the *mods* parameter, the caller specifies a sequence of modifications that are to be done in order on the attributes. Each modification in the sequence specifies a modification operation code (see reference manual page for *fn_attr_modify* ( )) and an attribute on which to operate.

The **FN_attrmodlist_t** type is described in the reference manual page for **FN_attrmodlist_t**.

RETURN VALUE

fn_attr_multi_modify ( ) returns 1 if all the modification operations were performed successfully. The function returns 0 if any error occurs. If the operation fails, *status* and *unexecuted_mods* are set as described below.

ERRORS

If an error is encountered while performing the list of modifications, *status* indicates the type of error and *unexecuted_mods* is set to a list of unexecuted modifications. The contents of *unexecuted_mods* do not share any state with *mods*; items in *unexecuted_mods* are copies of items in *mods* and appear in the same order in which they were originally supplied in *mods*. The first operation in *unexecuted_mods* is the first one that failed and the code in *status* applies to this modification operation in particular. If *status* indicates failure and a NULL pointer is returned in *unexecuted_mods*, that indicates no modifications were executed.

SEE ALSO

FN_attrmodlist_t, **FN_composite_name_t**, **FN_ctx_t**, **FN_status_t**, *fn_attr_modify* ( ), XFN_attribute_operations, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_attr_search, FN_searchlist_t, fn_searchlist_next, fn_searchlist_destroy — search for the atomic name of objects with the specified attributes in a single context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_searchlist_t *fn_attr_search(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrset_t *match_attrs,
    unsigned int return_ref,
    const FN_attrset_t *return_attr_ids,
    FN_status_t *status);

FN_string_t *fn_searchlist_next(
    FN_searchlist_t *sl,
    FN_ref_t **returned_ref,
    FN_attrset_t **returned_attrs,
    FN_status_t *status);

void fn_searchlist_destroy(
    FN_searchlist_t *sl);
```

DESCRIPTION

This set of operations is used to enumerate names of objects bound in the target context named *name* relative to the context *ctx* with attributes whose values match all those specified by *match_attrs*.

The attributes specified by *match_attrs* form a conjunctive AND expression against which the attributes of each named object in the target context are evaluated. For multi-valued attributes, the list order of values is ignored and attribute values not specified in *match_attrs* are ignored. If no value is specified for an attribute in *match_attrs*, the presence of the attribute is tested. If the value of *match_attrs* is 0, all names in the target context are enumerated.

If a non-zero value of *return_ref* is passed to *fn_attr_search*( ), the reference bound to the name is returned in the *returned_ref* argument to *fn_searchlist_next*( ).

Attribute identifiers and values associated with named objects that satisfy *match_attrs* may be returned by *fn_searchlist_next*( ). The attributes returned are those listed in the *return_attr_ids* argument to *fn_attr_search*( ). If the value of *return_attr_ids* is 0, all attributes are returned. If *return_attr_ids* is an empty **FN_attrset_t** object, no attributes are returned. Any attribute values in *return_attr_ids* are ignored; only the attribute identifiers are relevant for *return_attr_ids*.

The call to *fn_attr_search*( ) initiates the enumeration process. It returns a handle to an **FN_searchlist_t** object that is used to enumerate the names of the objects whose attributes match the attributes specified by *match_attrs*.

The operation *fn_searchlist_next*( ) returns the next name in the enumeration identified by the *sl*. The reference of the name is returned in *returned_ref* if *return_ref* was set in the call the *fn_attr_search*( ). The attributes specified by *return_attr_ids* are returned in *returned_attrs*. *fn_searchlist_next*( ) also updates *sl* to indicate the state of the enumeration. Successive calls to *fn_searchlist_next*( ) using *sl* return successive names, and optionally, references and attributes, in the enumeration and further update the state of the enumeration.

fn_searchlist_destroy( ) releases resources used during the enumeration. This can be invoked at any time to terminate the enumeration. If the argument to *fn_searchlist_destroy*( ) is NULL, no action is taken.

fn_attr_search( ) does not follow XFN links that are bound in the target context.

APPLICATION USAGE

The names enumerated using *fn_searchlist_next*( ) are not ordered in any way. Furthermore, there is no guarantee that any two series of enumerations on the same context with identical *match_attrs* will return the names in the same order.

RETURN VALUE

fn_attr_search( ) returns a pointer to an **FN_searchlist_t** object if the enumeration is successfully initiated; it returns a NULL pointer if the enumeration cannot be initiated or if no named object with attributes whose values match those specified in *match_attrs* is found.

fn_searchlist_next( ) returns a pointer to an **FN_string_t** object; it returns a NULL pointer if no more names can be returned in the enumeration. If *returned_ref* is a NULL pointer, or if the *return_ref* parameter to *fn_attr_search*( ) was zero, no reference is returned; otherwise, *returned_ref* contains the reference bound to the name. If *returned_attrs* is a NULL pointer, no attributes are returned; otherwise, *returned_attrs* contains the attributes associated with the named object, as specified by the *return_attr_ids* parameter to *fn_attr_search*( ).

In the case of a failure, these operations return in the *status* argument a code indicating the nature of the failure.

ERRORS

fn_attr_search( ) returns a NULL pointer if the enumeration could not be initiated. The status argument is set in the following way:

[FN_SUCCESS]
A named object could not be found whose attributes satisfied the implied filter of equality and conjunction.

[FN_E_ATTR_NO_PERMISSION]
The caller did not have permission to read one or more of the specified attributes.

[FN_E_INVALID_ATTR_VALUE]
A value type in the specified attributes did not match the syntax of the attribute against which it was being evaluated.

Other status codes are possible as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

Each successful call to *fn_searchlist_next*( ) returns a name and, optionally, the reference and requested attributes. *status* is set in the following way:

[FN_SUCCESS]
All requested attributes were returned successfully with the name.

[FN_E_ATTR_NO_PERMISSION]
The caller did not have permission to read one or more of the requested attributes.

[FN_E_INVALID_ATTR_IDENTIFIER]
A requested attribute identifier was not in a format acceptable to the naming system, or its contents was not valid for the format specified.

[FN_E_NO_SUCH_ATTRIBUTE]
The named object did not have one of the requested attributes.

[FN_E_INSUFFICIENT_RESOURCES]
Insufficient resources are available to return all the requested attributes and their values.

Errors
[FN_E_ATTR_NO_PERMISSION],
[FN_E_INVALID_ATTR_IDENTIFIER],
[FN_E_NO_SUCH_ATTRIBUTE],
[FN_E_INSUFFICIENT_RESOURCES]
indicate that some of the requested attributes may have been returned in *returned_attrs* but one or more of them could not be returned. Use *fn_attr_get*( ) or *fn_attr_multi_get*( ) to discover why these attributes could not be returned.

fn_searchlist_next( ) returns a NULL pointer if no more names can be returned. The status argument is set in the following way:

[FN_SUCCESS]
The search has completed successfully.

[FN_E_PARTIAL_RESULT]
The enumeration is not yet complete but cannot be continued.

[FN_E_ATTR_NO_PERMISSION]
The caller did not have permission to read one or more of the specified attributes.

[FN_E_INVALID_ENUM_HANDLE]
The supplied enumeration handle was not valid. Possible reasons could be that the handle was from another enumeration, or the context being enumerated no longer accepts the handle (due to such events as handle expiration or updates to the context).

Other status codes are possible as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

EXAMPLE

The following code fragment illustrates how the *fn_attr_search*( ) operation may be used. The code consists of three parts, preparing the arguments for the search, performing the search, and cleaning up. The first part involves getting the name of the context to start the search and constructing the set of attributes that named objects in the context must satisfy. This is done in the declarations part of the code and by the routine *get_search_query*( ).

The next part involves doing the search and enumerating the results of the search. This is done by first getting a context handle to the Initial Context, and then passing that handle along with the name of the target context and matching attributes to *fn_attr_search*( ). This particular call to *fn_attr_search*( ) is requesting that no reference be returned (by passing in 0 for *return_ref*), and that all attributes associated with the named object be returned (by passing in 0 as the *return_attr_ids* argument). If successful, *fn_attr_search*( ) returns *sl*, a handle for enumerating the results of the search. The results of the search are enumerated using calls to *fn_searchlist_next*( ), which returns the name of the object and the attributes associated with the named object in *returned_attrs*.

The last part of the code involves cleaning up the resources used during the search and enumeration. The call to *fn_searchlist_destroy*( ) releases resources reserved for this enumeration. The other calls release the context handle, name, attribute set and status objects created earlier.

```
/* Declarations */
FN_ctx_t *ctx;
FN_searchlist_t *sl;
FN_string_t *name;
FN_attrset_t *returned_attrs;
FN_status_t *status = fn_status_create();
FN_composite_name_t *target_name = get_name_from_user_input();
FN_attrset_t *match_attrs = get_search_query();

/* Get context handle to Initial Context */
ctx = fn_ctx_handle_from_initial(0, status);

/* error checking on ‘status’ */

/* Initiate search */
if ((sl=fn_attr_search(ctx, target_name, match_attrs,
    /* no reference */ 0, /* return all attrs */ 0, status)) == 0) {
    /* report ‘status’, cleanup, and exit */
}

/* Enumerate names and attributes requested */

while (name=fn_searchlist_next(sl, 0, &returned_attrs, status)) {
    /* do something with ‘name’ and ‘returned_attrs’*/
    fn_string_destroy(name);
    fn_attrset_destroy(returned_attrs);
}

/* check ‘status’ for reason for end of enumeration */

/* Clean up */
fn_searchlist_destroy(sl); /* Free resources of ‘sl’ */
fn_status_destroy(status);
fn_attrset_destroy(match_attrs);
fn_ctx_handle_destroy(ctx);
fn_composite_name_destroy(target_name);

/*
 * Procedure for constructing attribute set containing
 * attributes to be matched:
 *     "zip_code" attribute value is "02158"
 *     AND "employed" attribute is present.
*/

FN_attrset_t *
get_search_query()
{
/* Zip code and employed attribute identifier, syntax */
    extern FN_attribute_t    *attr_zip_code;
    extern FN_attribute_t    *attr_employed;

    FN_attribute_t *zip_code = fn_attribute_copy(attr_zip_code);
```

```
        FN_attrvalue_t zc_value = {5, "02158"};
        FN_attrset_t *match_attrs = fn_attrset_create();

        fn_attribute_add(zip_code, &zc_value, 0);
        fn_attrset_add(match_attrs, zip_code, 0);
        fn_attrset_add(match_attrs, attr_employed, 0);

        return (match_attrs);
    }
```

SEE ALSO

FN_attrvalue_t, FN_attribute_t, FN_attrset_t, FN_composite_name_t, FN_ctx_t, FN_status_t, FN_string_t, *fn_attr_ext_search*( ), *fn_ctx_list_names*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_bind — bind a reference to a name

SYNOPSIS

```
#include <xfn/xfn.h>

int fn_ctx_bind(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    const    FN_ref_t *ref,
    unsigned int exclusive,
    FN_status_t *status);
```

DESCRIPTION

This operation binds the supplied reference *ref* to the supplied composite name *name* relative to *ctx*. The binding is made in the target context — that context named by all but the terminal atomic part of *name*. The operation binds the terminal atomic name to the supplied reference in the target context. The target context must already exist.

The value of *exclusive* determines what happens if the terminal atomic part of the name is already bound in the target context. If *exclusive* is non-zero and *name* is already bound, the operation fails. If *exclusive* is zero, the new binding replaces any existing binding.

RETURN VALUE

When the bind operation is successful it returns 1; on error it returns 0.

ERRORS

fn_ctx_bind( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes. Of special relevance for this operation is the following status code:

[FN_E_NAME_IN_USE]
The supplied name is already in use.

APPLICATION USAGE

The value of *ref* cannot be NULL. If the intent is to reserve a name using *fn_ctx_bind*( ), a reference containing no address should be supplied. This reference may be name service-specific or it may be the conventional NULL reference defined in Appendix G.

If multiple sources are updating a reference, they must synchronize amongst each other when adding, modifying, or removing from the address list of a bound reference.

In naming systems that support attributes and store the attributes along with the names, when binding a reference in non-exclusive mode, any attributes associated with the former binding are removed.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_ref_t**, **FN_status_t**, *fn_attr_bind*( ), *fn_ctx_lookup*( ), *fn_ctx_unbind*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_create_subcontext — create a subcontext in a context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ref_t *fn_ctx_create_subcontext(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

DESCRIPTION

This operation creates a new XFN context of the same type as the target context — that named by all but the terminal atomic component of *name* — and binds it to the supplied composite name.

As with *fn_ctx_bind*( ), the target context must already exist. The new context is created and bound in the target context using the terminal atomic name in *name*. The operation returns a reference to the newly created context.

RETURN VALUE

fn_ctx_create_subcontext( ) returns a reference to the newly created context; if the operation fails, it returns a NULL pointer (0).

ERRORS

fn_ctx_create_subcontext( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes. Of special relevance for this operation is the following status code:

[FN_E_NAME_IN_USE]
The terminal atomic name already exists in the target context.

APPLICATION USAGE

The new subcontext is an XFN context and is created in the same naming system as the target context. The new subcontext also inherits the same syntax attributes as the target context. XFN does not specify any further properties of the new subcontext. The target context and its naming system determine these.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_ref_t**, **FN_status_t**, *fn_attr_create_subcontext*( ), *fn_ctx_bind*( ), *fn_ctx_lookup*( ), *fn_ctx_destroy*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_destroy_subcontext — destroy the named context and remove its binding from the parent context

SYNOPSIS

```
#include <xfn/xfn.h>

int fn_ctx_destroy_subcontext(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

DESCRIPTION

This operation destroys the subcontext named by *name* relative to *ctx*, and unbinds the name.

As with *fn_ctx_unbind*( ), this operation succeeds even if the terminal atomic name is not bound in the target context — the context named by all but the terminal atomic name in *name*.

RETURN VALUE

fn_ctx_destroy_subcontext( ) returns 1 on success and 0 on failure.

ERRORS

fn_ctx_destroy_subcontext( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes. Of special relevance for *fn_ctx_destroy_subcontext*( ) are the following status codes:

[FN_E_CTX_NOT_A_CONTEXT]
name does not name a context.

[FN_E_CTX_NOT_EMPTY]
The naming system being asked to do the destroy does not support removal of a context that still contains bindings.

APPLICATION USAGE

Some aspects of this operation are not specified by XFN, but are determined by the target context and its naming system. For example, XFN does not specify what happens if the named subcontext is non-empty when the operation is invoked.

In naming systems that support attributes, and store the attributes along with names or contexts, this operation removes the name, the context, and its associated attributes.

Normal resolution always follows links. In a *fn_ctx_destroy_subcontext*( ) operation, resolution of *name* continues to the target context; the terminal atomic name is not resolved. If the terminal atomic name is bound to a link, the link is not followed and the operation fails with [FN_E_CTX_NOT_A_CONTEXT] because the name is not bound to a context.

SEE ALSO

FN_ctx_t, **FN_composite_name_t**, **FN_status_t**, *fn_attr_create_subcontext*( ), *fn_ctx_create_subcontext*( ), *fn_ctx_unbind*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_equivalent_name — construct an equivalent name in same context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_composite_name_t *fn_ctx_equivalent_name(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_string_t *leading_name,
    FN_status_t *status);
```

DESCRIPTION

Given the name of an object *name* relative to the context *ctx*, this operation returns an equivalent name for that object, relative to the same context *ctx*, that has *leading_name* as its initial atomic name. Two names are said to be equivalent if they have prefixes that resolve to the same context, and the parts of the names immediately following the prefixes are identical.

The existence of a binding for *leading_name* in *ctx* does not guarantee that a name equivalent to *name* can be constructed. The failure may be because such equivalence is not meaningful, or due to the inability of the system to construct a name with the equivalence. For example, supplying `_thishost` as *leading_name* when *name* starts with `_myself` to *fn_ctx_equivalent_name*( ) in the Initial Context would not be meaningful; this results in the return of the error code [FN_E_NO_EQUIVALENT_NAME].

RETURN VALUE

If an equivalent name cannot be constructed, the value 0 is returned and *status* is set appropriately.

ERRORS

fn_ctx_equivalent_name( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes. The following status code is especially relevant for this operation.

[FN_E_NO_EQUIVALENT_NAME]
: No equivalent name can be constructed, either because there is no meaningful equivalence between *name* and *leading_name*, or the system does not support constructing the requested equivalent name, for implementation-specific reasons.

EXAMPLES

In the Initial Context supporting the XFN enterprise policies described in Appendix D, a user `jsmith` is able to name one of her files relative to this context in several ways:

```
_myself/_fs/map.ps
_user/jsmith/_fs/map.ps
_orgunit/finance/_user/jsmith/_fs/map.ps
```

The first of these may be appealing to the user `jsmith` in her day-to-day operations. This name is not, however, appropriate for her to use when referring to the file in an electronic mail message sent to a colleague. The second of these names would be appropriate if the colleague were in the same organizational unit, and the third appropriate for anyone in the same enterprise.

When the following sequence of instructions is executed by the user `jsmith` in the organizational unit `finance`, *enterprise_wide_name* would contain the composite name `_orgunit/finance/_user/jsmith/_fs/map.ps`.

```
FN_composite_name_t* name = fn_composite_name_from_str(
    (const unsigned char *)"_myself/_fs/map.ps");
```

```
FN_string_t* org_lead =
    fn_string_from_str((const unsigned char*)"_orgunit");
FN_status_t* status = fn_status_create();
FN_composite_name_t* enterprise_wide_name;

FN_ctx_t* init_ctx = fn_ctx_handle_from_initial(0, status);
/* check status of from_initial() */

enterprise_wide_name = fn_ctx_equivalent_name(init_ctx, name, org_lead, status);
```

When the following sequence of instructions is executed by the user `jsmith` in the organizational unit `finance`, *shortest_name* would contain the composite name `_myself/_fs/map.ps`.

```
FN_composite_name_t* name = fn_composite_name_from_str(
    (const unsigned char *)"_orgunit/finance/_user/jsmith/_fs/map.ps");
FN_string_t* mylead = fn_string_from_str((const unsigned char*)"_myself");
FN_status_t* status = fn_status_create();
FN_composite_name_t* shortest_name;

FN_ctx_t* init_ctx = fn_ctx_handle_from_initial(0, status);
/* check status of from_initial() */

shortest_name = fn_ctx_equivalent_name(init_ctx, name, mylead, status);
```

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_status_t**, **FN_string_t**, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_get_ref — return a context's reference

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ref_t *fn_ctx_get_ref(
    const    FN_ctx_t *ctx,
    FN_status_t *status);
```

DESCRIPTION

This operation returns a reference to the supplied context object.

RETURN VALUE

fn_ctx_get_ref( ) returns a pointer to an **FN_ref_t** object if the operation succeeds, it returns 0 if the operation fails.

ERRORS

fn_ctx_get_ref( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes. The following status code is of particular relevance to this operation:

[FN_E_OPERATION_NOT_SUPPORTED]
Using the *fn_ctx_get_ref*( ) operation on the Initial Context returns this status code.

APPLICATION USAGE

fn_ctx_get_ref( ) cannot be used on the Initial Context. *fn_ctx_get_ref*( ) can be used on contexts bound in the Initial Context (in other words, the bindings in the Initial Context have references).

If the context handle was created earlier using the *fn_ctx_handle_from_ref*( ) operation, the reference returned by the *fn_ctx_get_ref*( ) operation may not necessarily be exactly the same in content as that originally supplied. For example, *fn_ctx_handle_from_ref*( ) may construct the context handle from one address from the list of addresses. The context implementation may return with a call to *fn_ctx_get_ref*( ) only that address, or a more complete list of addresses than what was supplied in *fn_ctx_handle_from_ref*( ).

SEE ALSO

FN_ctx_t, **FN_ref_t**, **FN_status_t**, *fn_ctx_handle_from_initial*( ), *fn_ctx_handle_from_ref*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_get_syntax_attrs — return syntax attributes associated with named context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_attrset_t *fn_ctx_get_syntax_attrs(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);
```

DESCRIPTION

Each context has an associated set of syntax-related attributes. This operation returns the syntax attributes associated with the context named by *name* relative to the context *ctx*.

The attributes must contain the attribute *fn_syntax_type* (FN_ID_STRING format). If the context supports a syntax that conforms to the XFN standard syntax model, *fn_syntax_type* is set to **standard** (ASCII attribute syntax) and the attribute set contains the rest of the relevant syntax attributes described the reference manual page for XFN_compound_syntax.

This operation is different from other XFN attribute operations in that these syntax attributes could be obtained directly from the context. Attributes obtained through other XFN attribute operations may not necessarily be associated with the context; they may be associated with the reference of context, rather than the context itself (see reference manual page for XFN_attributes).

RETURN VALUE

fn_ctx_get_syntax_attrs( ) returns an attribute set if successful; it returns a NULL pointer (0) if the operation fails.

ERRORS

fn_ctx_get_syntax_attrs( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

APPLICATION USAGE

Implementations may choose to support other syntax types in addition to, or in place of, the XFN standard syntax model, in which case, the value of the "fn_syntax_type" attribute would be set to an implementation-specific string, and different or additional syntax attributes will be in the set.

Syntax attributes of a context may be generated automatically by a context, in response to *fn_ctx_get_syntax_attrs*( ), or they may be created and updated using the base attribute operations. This is implementation-dependent.

SEE ALSO

FN_attrset_t, **FN_composite_name_t**, **FN_compound_name_t**, **FN_ctx_t**, **FN_status_t**, *fn_attr_get*( ), *fn_attr_multi_get*( ), XFN_compound_syntax, XFN_attribute_operations, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_handle_destroy — release storage associated with context handle

SYNOPSIS

```
#include <xfn/xfn.h>

void fn_ctx_handle_destroy(FN_ctx_t *ctx);
```

DESCRIPTION

This operation destroys the context handle *ctx* and allows the implementation to free resources associated with the context handle. This operation does not affect the state of the context itself. If the argument to *fn_ctx_handle_destroy*( ) is NULL, no action is taken.

SEE ALSO

FN_ctx_t, *fn_ctx_handle_from_initial*( ), *fn_ctx_handle_from_ref*( ), **<xfn/xfn.h>**.

NAME

fn_ctx_handle_from_initial — return a handle to the Initial Context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ctx_t *fn_ctx_handle_from_initial(
    unsigned int authoritative,
    FN_status_t *status);
```

DESCRIPTION

This operation returns a handle to the caller's Initial Context. On successful return, the handle points to a context which meets the specification of the XFN Initial Context.

authoritative specifies whether the handle to the context returned should be authoritative with respect to information the context obtains from the naming service. When the flag is non-zero, subsequent operations on the context will access the most authoritative information. When *authoritative* is zero, the handle to the context returned need not be authoritative.

RETURN VALUE

fn_ctx_handle_from_initial( ) returns a pointer to an **FN_ctx_t** object if the operation succeeds; it returns a NULL pointer (0) otherwise.

ERRORS

fn_ctx_handle_from_initial( ) sets only the status code portion of the status object *status*.

APPLICATION USAGE

Authoritativeness is determined by specific naming services. For example, in a naming service that supports replication using a master/slave model, the source of authoritative information would come from the master server. In some naming systems, bypassing the naming service cache may reach servers which provide the most authoritative information. The availability of an authoritative context might be lower due to the lower number of servers offering this service. For the same reason, it might also provide poorer performance than contexts that need not be authoritative.

Applications should set *authoritative* to zero for typical day-to-day operations. Applications should only set *authoritative* to a non-zero value when they require access to the most authoritative information, possibly at the expense of lower availability and/or poorer performance.

It is implementation-dependent whether authoritativeness is transferred from one context to the next as composite name resolution proceeds. Getting an authoritative context handle to the Initial Context means that operations on bindings in the Initial Context are processed using the most authoritative information. Contexts referenced implicitly through an authoritative Initial Context (for example, through the use of composite names) may not necessarily themselves be authoritative.

SEE ALSO

FN_ctx_t, **FN_status_t**, *fn_ctx_get_ref*( ), *fn_ctx_handle_from_ref*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_handle_from_ref — construct a handle to a context object using the given reference

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ctx_t *fn_ctx_handle_from_ref(
    const    FN_ref_t *ref,
    unsigned int authoritative,
    FN_status_t *status);
```

DESCRIPTION

This operation creates a handle to an **FN_ctx_t** object using an **FN_ref_t** object for that context.

authoritative specifies whether the handle to the context returned should be authoritative with respect to information the context obtains from the naming service. When the flag is non-zero, subsequent operations on the context will access the most authoritative information. When *authoritative* is zero, the handle to the context returned need not be authoritative.

RETURN VALUE

This operations returns a pointer to an **FN_ctx_t** object if the operation succeeds, otherwise, it returns a NULL pointer (0).

ERRORS

fn_ctx_handle_from_ref( ) sets *status* as described in the reference manual page for **FN_status_t** and XFN_status_codes. The following status code is of particular relevance to this operation.

[FN_E_NO_SUPPORTED_ADDRESS]
: A context object of the specified authoritativeness could not be constructed from a particular reference. The reference contained no address type over which the context interface was supported.

APPLICATION USAGE

Authoritativeness is determined by specific naming services. For example, in a naming service that supports replication using a master/slave model, the source of authoritative information would come from the master server. In some naming systems, bypassing the naming service cache may reach servers which provide the most authoritative information. The availability of an authoritative context might be lower due to the lower number of servers offering this service. For the same reason, it might also provide poorer performance than contexts that need not be authoritative.

Applications set *authoritative* to zero for typical day-to-day operations. Applications only set *authoritative* to a non-zero value when they require access to the most authoritative information, possibly at the expense of lower availability and/or poorer performance.

To control the authoritativeness of the target context, the application first resolves explicitly to the target context using *fn_ctx_lookup*( ). It then uses *fn_ctx_handle_from_ref*( ) with the appropriate authoritative argument to obtain a handle to the context. This returns a handle to a context with the specified authoritativeness. The application then uses the XFN operations, such as lookup and list, with this context handle.

It is implementation-dependent whether authoritativeness is transferred from one context to the next as composite name resolution proceeds. The application should use the approach recommended above to achieve the desired level of authoritativeness on a per context basis.

SEE ALSO

FN_ctx_t, **FN_ref_t**, **FN_status_t**, *fn_ctx_handle_destroy*( ), *fn_ctx_get_ref*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_list_bindings, FN_bindinglist_t, fn_bindinglist_next, fn_bindinglist_destroy — list the atomic names and references bound in a context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_bindinglist_t *fn_ctx_list_bindings(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);

FN_string_t *fn_bindinglist_next(
    FN_bindinglist_t *bl,
    FN_ref_t **ref,
    FN_status_t *status);

void fn_bindinglist_destroy(
    FN_bindinglist_t *bl);
```

DESCRIPTION

This set of operations is used to list the names and bindings in the context named by *name* relative to the context *ctx*. Note that *name* must name a context. If the intent is to list the contents of *ctx*, *name* should be an empty composite name.

The semantics of these operations are similar to those for listing names (see the reference manual page for *fn_ctx_list_names*( )). In addition to a name string being returned, *fn_bindinglist_next*( ) also returns the reference of the binding for each member of the enumeration. If the argument to *fn_binding_list_destroy*( ) is NULL, no action is taken.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_ref_t**, **FN_status_t**, **FN_string_t**, *fn_ctx_list_names*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_list_names, FN_namelist_t, fn_namelist_next, fn_namelist_destroy — list the atomic names bound in a context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_namelist_t *fn_ctx_list_names(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);

FN_string_t *fn_namelist_next(
    FN_namelist_t *nl,
    FN_status_t *status);

void fn_namelist_destroy(
    FN_namelist_t *nl);
```

DESCRIPTION

This set of operations is used to list the names bound in the target context named *name* relative to the context *ctx*. Note that *name* must name a context. If the intent is to list the contents of *ctx*, *name* should be an empty composite name.

The call to *fn_ctx_list_names*( ) initiates the enumeration process. It returns a handle to an **FN_namelist_t** object that can be used to enumerate the names in the target context.

The operation *fn_namelist_next*( ) returns the next name in the enumeration identified by *nl* and updates *nl* to indicate the state of the enumeration. Successive calls to *fn_namelist_next*( ) using *nl* return successive names in the enumeration and further update the state of the enumeration. *fn_namelist_next*( ) returns a NULL pointer when the enumeration has been completed.

fn_namelist_destroy( ) is used to release resources used during the enumeration. This may be invoked before the enumeration has completed to terminate the enumeration. If the argument to *fn_namelist_destroy*( ) is NULL, no action is taken.

RETURN VALUE

fn_ctx_list_names( ) returns a pointer to an **FN_namelist_t** object if the enumeration is successfully initiated; otherwise it returns a NULL pointer.

fn_namelist_next( ) returns a NULL pointer if no more names can be returned in the enumeration.

In the case of a failure, these operations return in *status* a code indicating the nature of the failure.

ERRORS

Each successful call to *fn_namelist_next*( ) returns a name and sets *status* to [FN_SUCCESS].

When *fn_namelist_next*( ) returns a NULL pointer, it indicates that no more names can be returned. *status* is set in the following way:

[FN_SUCCESS]
The enumeration has completed successfully.

[FN_E_INVALID_ENUM_HANDLE]
The supplied enumeration handle is not valid. Possible reasons could be that the handle was from another enumeration, or the context being enumerated no longer accepts the handle (due to such events as handle expiration or updates to the context).

[FN_E_PARTIAL_RESULT]
The enumeration is not yet complete but cannot be continued.

Other status codes, such as [FN_E_COMMUNICATION_FAILURE], are also possible in calls to *fn_ctx_list_names*( ) and *fn_namelist_next*( ). These functions set *status* for these other status codes as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

EXAMPLE

The following code fragment illustrates a how the list names operations may be used.

```
extern FN_string_t *user_input;
FN_ctx_t *ctx;
FN_composite_name_t *target_name = fn_composite_name_from_string(
    user_input);
FN_status_t *status = fn_status_create();
FN_string_t *name;
FN_namelist_t *nl;

ctx = fn_ctx_handle_from_initial(0, status);
/* error checking on 'status' */

if ((nl=fn_ctx_list_names(ctx, target_name, status)) == 0) {
    /* report 'status' and exit */
}

while (name=fn_namelist_next(nl, status)) {
    /* do something with 'name' */
    fn_string_destroy(name);
}

/* check 'status' for reason for end of enumeration */
/* and report if necessary                          */

/* clean up */
fn_namelist_destroy(nl);
```

APPLICATION USAGE

The names enumerated using *fn_namelist_next*( ) are not ordered in any way. There is no guaranteed relation between the order in which names are added to a context and the order of names obtained by enumeration. The specification does *not* guarantee that any two series of enumerations will return the names in the same order.

When a name is added to or removed from a context, this may or may not invalidate the enumeration handle that the client holds for that context. If the enumeration handle becomes invalid, the status code [FN_E_INVALID_ENUM_HANDLE] is returned in *status*. If the enumeration handle remains valid, the update may or may not be visible to the client.

In addition, there may be a relationship between the *ctx* argument supplied to *fn_ctx_list_names*( ) and the **FN_namelist_t** object it returns. For example, some implementations may store the context handle *ctx* within the **FN_namelist_t** object for subsequent *fn_namelist_next*( ) calls. In general, a *fn_ctx_handle_destroy*( ) should not be invoked on *ctx* until the enumeration has terminated.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_status_t**, **FN_string_t**, *fn_ctx_handle_destroy*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_lookup — look up name in context

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ref_t *fn_ctx_lookup(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

DESCRIPTION

This operation returns the reference bound to *name* relative to the context *ctx*.

RETURN VALUE

If the operation succeeds, the *fn_ctx_lookup*( ) function returns a handle to the reference bound to *name*. Otherwise, 0 is returned and *status* is set appropriately.

ERRORS

fn_ctx_lookup( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

APPLICATION USAGE

Some naming services may not always have reference information for all names in their contexts; for such names, such naming services may return a special reference whose type indicates that the name is not bound to any address. This reference may be name service specific or it may be the conventional NULL reference defined in Appendix G.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_ref_t**, **FN_status_t**, XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_lookup_link — look up the link reference bound to a name

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ref_t *fn_ctx_lookup_link(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

DESCRIPTION

This operation returns the XFN link bound to *name* if *name* is bound to an XFN link; otherwise, it returns the reference bound to *name*.

The normal *fn_ctx_lookup*( ) operation follows all links encountered, including any bound to the terminal atomic part of *name*. This operation differs from the normal lookup in that when the terminal atomic part of *name* is an XFN link, this link is not followed, and the operation returns the link.

RETURN VALUE

If *fn_ctx_lookup_link*( ) fails, a NULL pointer (0) is returned.

ERRORS

fn_ctx_lookup_link( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_ref_t**, **FN_status_t**, *fn_ctx_lookup*( ), XFN_status_codes, XFN_links, **<xfn/xfn.h>**.

NAME

fn_ctx_rename — rename the name of a binding

SYNOPSIS

```
#include <xfn/xfn.h>

int fn_ctx_rename(
    FN_ctx_t *ctx,
    const FN_composite_name_t *oldname,
    const FN_composite_name_t *newname,
    unsigned int exclusive,
    FN_status_t *status);
```

DESCRIPTION

The *fn_ctx_rename*( ) operation binds the reference currently bound to *oldname* relative to *ctx*, to the name *newname*, and unbinds *oldname*. *newname* is resolved relative to the target context (that named by all but the terminal atomic part of *oldname*).

If *exclusive* is zero, the operation overwrites any old binding of *newname*. If *exclusive* is non-zero, the operation fails if *newname* is already bound.

RETURN VALUE

fn_ctx_rename( ) returns 1 if the operation is successful, 0 otherwise.

ERRORS

fn_ctx_rename( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

APPLICATION USAGE

The only restriction that XFN places on *newname* is that it be resolved relative to the target context. XFN does not specify further restrictions on *newname*. For example, in some implementations, *newname* might be restricted to be a name in the same naming system as the terminal component of *oldname*. In another implementation, *newname* might be restricted to be an atomic name.

Normal resolution always follows links. In a *fn_ctx_rename*( ) operation, resolution of *oldname* continues to the target context; the terminal atomic name is not resolved. If the terminal atomic name is bound to a link, the link is not followed and the operation binds *newname* to the link and unbinds the terminal atomic name of *oldname*.

In naming systems that support attributes and store the attributes along with the names, any attributes associated with the terminal atomic name of *oldname* become associated with *newname*. Any attributes associated with the terminal atomic name of *oldname* are removed.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_ref_t**, **FN_status_t**, *fn_ctx_bind*( ), *fn_ctx_unbind*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

fn_ctx_unbind — unbind a name from a context

SYNOPSIS

```
#include <xfn/xfn.h>

int fn_ctx_unbind(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);
```

DESCRIPTION

This operation removes the terminal atomic name in *name* from the the target context — that named by all but the terminal atomic part of *name*.

This operation is successful even if the terminal atomic name was not bound in target context, but fails if any of the intermediate names are not bound. *fn_ctx_unbind*( ) is idempotent.

RETURN VALUE

The operation returns 1 if successful, and 0 otherwise.

ERRORS

fn_ctx_unbind( ) sets *status* as described in the reference manual pages for **FN_status_t** and XFN_status_codes.

Certain naming systems may disallow unbinding a name if the name is bound to an existing context in order to avoid orphan contexts that cannot be reached via any name. In such situations, the status code [FN_E_NAME_IN_USE] is returned.

APPLICATION USAGE

In naming systems that support attributes, and store the attributes along with the names, the unbind operation removes the name and its associated attributes.

Normal resolution always follows links. In an *fn_ctx_unbind*( ) operation, resolution of *name* continues to the target context; the terminal atomic name is not resolved. If the terminal atomic name is bound to a link, the link is not followed and the link itself is unbound from the terminal atomic name.

SEE ALSO

FN_composite_name_t, **FN_ctx_t**, **FN_ref_t**, **FN_status_t**, *fn_attr_bind*( ), *fn_ctx_bind*( ), *fn_ctx_lookup* ( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

XFN attribute operations — an overview of XFN attribute operations

SYNOPSIS

```
#include <xfn/xfn.h>

int fn_attr_bind(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_ref_t *ref,
    const FN_attrset_t *attrs,
    unsigned int exclusive,
    FN_status_t *status);

FN_searchlist_t *fn_attr_search(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrset_t *match_attrs,
    unsigned int return_ref,
    const FN_attrset_t *return_attr_ids,
    FN_status_t *status);

FN_string_t *fn_searchlist_next(
    FN_searchlist_t *sl,
    FN_ref_t **returned_ref,
    FN_attrset_t **returned_attrs,
    FN_status_t *status);

void fn_searchlist_destroy(
    FN_searchlist_t *sl);

FN_ref_t *fn_attr_create_subcontext(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrset_t *attrs,
    FN_status_t *status)

FN_ext_searchlist_t *fn_attr_ext_search(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_search_control_t *control,
    const FN_search_filter_t *filter,
    FN_status_t *status);

FN_composite_name_t *fn_ext_searchlist_next(
    FN_ext_searchlist_t *esl,
    FN_ref_t **returned_ref,
    FN_attrset_t **returned_attrs,
    FN_status_t *status);

void fn_ext_searchlist_destroy(
    FN_ext_searchlist_t *esl);
```

```
FN_attribute_t *fn_attr_get(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    const    FN_identifier_t *attribute_id,
    unsigned int follow_link,
    FN_status_t *status);

int fn_attr_modify(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    unsigned int mod_op,
    const    FN_attribute_t *attr,
    unsigned int follow_link,
    FN_status_t *status);

FN_attrset_t *fn_attr_get_ids(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    unsigned int follow_link,
    FN_status_t *status);

FN_valuelist_t *fn_attr_get_values(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    const     FN_identifier_t *attribute_id,
    unsigned int follow_link,
    FN_status_t *status);

FN_attrvalue_t *fn_valuelist_next(
    FN_valuelist_t *vl,
    FN_identifier_t **attr_syntax,
    FN_status_t *status);

void fn_valuelist_destroy(
    FN_valuelist_t *vl);

FN_multigetlist_t *fn_attr_multi_get(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    const     FN_attrset_t *attr_ids,
    unsigned int follow_link,
    FN_status_t *status);

FN_attribute_t *fn_multigetlist_next(
    FN_multigetlist_t *ml,
    FN_status_t *status);

void fn_multigetlist_destroy(
    FN_multigetlist_t *ml);

int fn_attr_multi_modify(
    FN_ctx_t *ctx,
```

```
    const    FN_composite_name_t *name,
    const    FN_attrmodlist_t *mods,
    unsigned int follow_link,
    FN_attrmodlist_t **unexecuted_mods,
    FN_status_t *status);

FN_attrset_t *fn_ctx_get_syntax_attrs(
    FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);
```

DESCRIPTION

XFN assumes the following model for attributes. A set of zero or more attributes is associated with a named object. Each attribute in the set has a unique attribute identifier, an attribute syntax and a (possibly empty) set of distinct data values. Each attribute value has an opaque data type. The attribute identifier serves as a name for the attribute. The attribute syntax indicates the how the value is encoded in the buffer.

The operations of the base attribute interface may be used to examine and modify the settings of attributes associated with existing named objects. These objects may be contexts or other types of objects.

The range of support for attribute operations may vary widely. Some naming systems may not support any attribute operations. Other naming systems may only support read operations, or operations on attributes whose identifiers are in some fixed set. A naming system may limit attributes to have a single value, or may require at least one value. Some naming systems may only associate attributes with context objects, while others may allow associating attributes with non-context objects.

The following describes briefly the operations in the base and extended attribute interfaces. Detailed descriptions are given in the respective reference manual pages for these operations:

fn_attr_bind( )
: binds the supplied reference *ref* to the supplied composite name *name*, resolved relative to *ctx*. In addition, it associates attributes specified in *attrs* with the named object.

fn_attr_create_subcontext( )
: creates a new context of the same type as the target context binds it to the composite name *name* relative to the context *ctx*, and returns a reference to the newly created context. In addition, it associates the attributes specified in *attrs* with the new context.

fn_attr_get( )
: returns the attribute identified

fn_attr_modify( )
: modifies the attribute identified as described by *mod_op*

fn_attr_get_ids( )
: returns the identifiers of the attributes of the named object

fn_attr_get_values( )
: and its set of related operations are used for returning the individual values of an attribute

fn_attr_multi_get( )
: and its set of related operations are used for returning the requested attributes associated with the named object

fn_attr_multi_modify( )
modifies multiple attributes associated with the named object in a single invocation

fn_ctx_get_syntax_attrs( )
returns the syntax attributes associated with the named context

fn_attr_search( )
and its related operations are used for returning the objects with attributes that match the given attributes

fn_attr_ext_search( )
and its related operations are used for returning the objects with attributes that match the specified search criteria.

The value of the *follow_link* parameter determines what happens when the terminal atomic part of *name* is bound to an XFN link. If *follow_link* is non-zero, such a link is followed and attributes associated with the final named object are examined or modified. If *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

ERRORS

status is set as described in the reference manual pages for **FN_status_t** and XFN_status_codes. The following status codes are of special relevance to attribute operations:

[FN_E_ATTR_IN_USE]
When an attribute is being modified using the operation FN_ATTR_OP_ADD_EXCLUSIVE and an attribute with the same identifier already exists, the operation fails with FN_E_ATTR_IN_USE.

[FN_E_ATTR_VALUE_REQUIRED]
The operation attempted to create an attribute without a value, and the specific naming system does not allow this.

[FN_E_ATTR_NO_PERMISSION]
The caller did not have permission to perform the attempted attribute operation.

[FN_E_INSUFFICIENT_RESOURCES]
There is insufficient resources to retrieve the requested attribute(s).

[FN_E_INVALID_ATTR_IDENTIFIER]
The attribute identifier was not in a format acceptable to the naming system, or its contents was not valid for the format specified for the identifier.

[FN_E_INVALID_ATTR_VALUE]
One of the values supplied was not in the appropriate form for the given attribute.

[FN_E_NO_SUCH_ATTRIBUTE]
The object did not have an attribute with the given identifier.

[FN_E_SEARCH_INVALID_FILTER]
The filter expression had a syntax error or some other problem.

[FN_E_SEARCH_INVALID_OP]
An operator in the filter expression is not supported or, if the operator is an extended operator, the number of types of arguments supplied does not match the signature of the operation.

[FN_E_SEARCH_INVALID_OPTION]
A supplied search control option could not be supported.

[FN_E_TOO_MANY_ATTR_VALUES]
The operation attempted to associate more values with an attribute than the naming system supported.

APPLICATION USAGE

Except for *fn_ctx_get_syntax_attrs*( ), an attribute operation using a composite name is not necessarily equivalent to an independent *fn_ctx_lookup*( ) operation followed by an attribute operation in which the caller supplies the resulting reference and an empty name. This is because there are a range of attribute models in which an attribute is associated with a name in a context, or an attribute is associated with the object named, or both. XFN accommodates all of these alternatives. Invoking an attribute operation using the target context and the terminal atomic name accesses either the attributes that are associated with the target name or target named object — this is dependent on the underlying attribute model. This document uses the term *attributes associated with a named object* to refer to all of these cases.

XFN specifies no guarantees about the relationship between the attributes and the reference associated with a given name. Some naming systems may store the reference bound to a name in one or more attributes associated with a name. Attribute operations might affect the information used to construct a reference.

To avoid undefined results, programmers must use the operations in the context interface and not attribute operations when the intention is to manipulate a reference. Programmers should avoid the use of specific knowledge about how an XFN context implementation over a particular naming system constructs references.

SEE ALSO

FN_attribute_t, **FN_attrvalue_t**, **FN_attrset_t**, **FN_composite_name_t**, **FN_ctx_t**, **FN_identifier_t**, **FN_ref_t**, **FN_status_t**, *fn_attr_bind*( ), *fn_attr_create_subcontext*( ), *fn_attr_get*( ), *fn_attr_get_ids*( ), *fn_attr_get_values*( ), *fn_attr_modify*( ), *fn_attr_multi_get*( ), *fn_attr_multi_modify*( ), *fn_attr_search*( ), *fn_attr_ext_search*( ), *fn_ctx_get_syntax_attrs*( ), *fn_ctx_lookup*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

XFN composite syntax — an overview of the syntax for XFN composite name

SYNOPSIS

```
#include <xfn/xfn.h>

FN_composite_name_t *fn_composite_name_from_string(
    const FN_string_t *str);

FN_string_t *fn_string_from_composite_name(
    const FN_composite_name_t *name);
```

DESCRIPTION

An *XFN composite name* consists of an ordered list of zero or more components. Each component is a string name from the namespace of a single naming system. It may be an atomic or a compound name in that namespace.

XFN defines an abstract data type, **FN_composite_name_t**, for representing the structural form of a composite name. XFN also defines a standard string form for composite names. This form is the concatenation of the components of a composite name from left to right with the *XFN component separator* ('/') character to separate each component.

The function *fn_composite_name_from_string*( ) parses the string representation of a composite name into its corresponding composite name object **FN_composite_name_t**. The function *fn_string_from_composite_name*( ) composes the string representation of a composite name given its composite name object form **FN_composite_name_t**.

The details of the syntax and the semantics of these functions are described in Section 4.1 on page 55.

APPLICATION USAGE

Special characters used in the XFN composite name syntax, such as the separator or escape characters, have the same encoding as they would in ISO 646.

The minimum requirement for all XFN implementations is to support the portable representation of ISO 646 (same encoding as ASCII) for communication of name strings. All other representations are optional. See Section 2.5 on page 17.

All characters of the string form of a XFN composite name use a single encoding. This does not preclude component names of a composite name in its structural form from having different encodings. Code set mismatches that occur during the process of converting a composite name structure to its string form are resolved in an implementation-dependent way. When an implementation discovers that a composite name has components with incompatible code sets, it returns the error code [FN_E_INCOMPATIBLE_CODE_SETS]. Incompatibility between language or territory locale information are indicated by use of the error code [FN_E_INCOMPATIBLE_LOCALES].

SEE ALSO

FN_string_t, **FN_composite_t**, **<xfn/xfn.h>**.

NAME

XFN compound syntax — an overview of XFN model for compound name parsing

SYNOPSIS

```
#include <xfn/xfn.h>

FN_attrset_t *fn_ctx_get_syntax_attrs(
    const     FN_ctx_t *ctx,
    const     FN_composite_name_t *name,
    FN_status_t *status);

FN_compound_name_t *fn_compound_name_from_syntax_attrs(
    const     FN_attrset_t *aset,
    const     FN_string_t *name,
    FN_status_t *status);
```

DESCRIPTION

Each naming system in an XFN federation has a naming convention. XFN defines a standard model of expressing compound name syntax that covers a large number of specific name syntaxes and is expressed in terms of syntax properties of the naming convention.

The model uses the attributes in the following table to describe properties of the syntax. Unless otherwise qualified, these syntax attributes have attribute identifiers that use the FN_ID_STRING format. A context that supports the XFN standard syntax model has an attribute set containing the *fn_syntax_type* (FN_ID_STRING format) attribute with the value **standard** (ASCII attribute syntax).

The XFN standard syntax attributes are interpreted according to the following rules:

1. In a string without quotes or escapes, any instance of the separator string delimits two atomic names.
2. A separator, quotation or escape string is escaped if preceded immediately (on the left) by the escape string.
3. A non-escaped begin-quote which precedes a component must be matched by a non-escaped end-quote at the end of the component. Quotes embedded in non-quoted names are treated as simple characters and do not need to be matched. An unmatched quotation fails with the status code [FN_E_ILLEGAL_NAME].
4. If there are multiple values for begin-quote and end-quote, a specific begin-quote value must be matched with its corresponding end-quote value.
5. When the separator appears between a (non-escaped) begin quote and the end quote, it is ignored.
6. When the separator is escaped, it is ignored. An escaped begin-quote or end-quote string is not treated as a quotation mark. An escaped escape string is not treated as an escape string.
7. A non-escaped escape string appearing within quotes is interpreted as an escape string. This can be used to embed an end-quote within a quoted string.
8. An escape string which precedes a character other than an escape string, a begin-quote or an end-quote is consumed (in other words, escaping a non-meta character returns the non-meta character itself).

After constructing a compound name from a string, the resulting component atoms have one level of escape strings and quotations interpreted and consumed.

fn_ctx_get_syntax_attrs( ) is used to obtain the syntax attributes associated with a context.

fn_compound_name_from_syntax( ) is used to construct a compound name object using the string form of the name and the syntax attributes of the name.

Attribute Identifier	Attribute Value
fn_syntax_type	Its value is the ASCII string **standard** if the context supports the XFN standard syntax model. Its value is an implementation-specific value if another syntax model is supported.
fn_std_syntax_direction	Its value is an ASCII string, one of **left_to_right**, **right_to_left**, or **flat**. This determines whether the order of components in a compound name string goes from left to right, right to left, or whether the namespace is flat (in other words, not hierarchical — all names are atomic).
fn_std_syntax_separator	Its value is the separator string for this name syntax. This attribute is required unless the *fn_std_syntax_direction* is **flat**.
fn_std_syntax_escape	If present, its value is the escape string for this name syntax.
fn_std_syntax_case_insensitive	If this attribute is present, it indicates that names that differ only in case are considered identical. If this attribute is absent, it indicates that case is significant. If a value is present, it is ignored.
fn_std_syntax_begin_quote1	If present, its value is one of the begin-quote strings for this syntax. If *fn_std_syntax_end_quote1* is absent but *fn_std_syntax_begin_quote1* is present, the quote-string specified in *fn_std_syntax_begin_quote1* is used as both the begin and end quote-strings. If *fn_std_syntax_end_quote1* is present but *fn_std_syntax_begin_quote1* is absent, the quote-string specified in *fn_std_syntax_end_quote1* is used as both the begin and end quote-strings.
fn_std_syntax_end_quote1	If present, its value is the end-quote string that matches the begin-quote string specified in *fn_std_syntax_begin_quote1*. If *fn_std_syntax_end_quote1* is absent but *fn_std_syntax_begin_quote1* is present, the quote-string specified in *fn_std_syntax_begin_quote1* is used as both the begin and end quote-strings. If *fn_std_syntax_end_quote1* is present but *fn_std_syntax_begin_quote1* is absent, the quote-string specified in *fn_std_syntax_end_quote1* is used as both the begin and end quote-strings.

Attribute Identifier	Attribute Value
fn_std_syntax_begin_quote2	If present, its value is one of the begin-quote strings for this syntax. If *fn_std_syntax_end_quote2* is absent but *fn_std_syntax_begin_quote2* is present, the quote-string specified in *fn_std_syntax_begin_quote2* is used as both the begin and end quote-strings. If *fn_std_syntax_end_quote2* is present but *fn_std_syntax_begin_quote2* is absent, the quote-string specified in *fn_std_syntax_end_quote2* is used as both the begin and end quote-strings.
fn_std_syntax_end_quote2	If present, its value is the end-quote string that matches the begin-quote string specified in *fn_std_syntax_begin_quote2*. If *fn_std_syntax_end_quote2* is absent but *fn_std_syntax_begin_quote2* is present, the quote-string specified in *fn_std_syntax_begin_quote2* is used as both the begin and end quote-strings. If *fn_std_syntax_end_quote2* is present but *fn_std_syntax_begin_quote2* is absent, the quote-string specified in *fn_std_syntax_end_quote2* is used as both the begin and end quote-strings.
fn_std_syntax_ava_separator	If present, its value is the attribute value assertion separator string for this syntax.
fn_std_syntax_typeval_separator	If present, its value is the attribute type-value separator string for this syntax.

fn_std_syntax_locales	If this attribute is not present, or if the value is empty, the only locale supported by the context is the ‘‘C’’ locale. The ‘‘C’’ locale’s repertoire of characters is restricted to that defined by the portable representation of ISO-646 (same encoding as ASCII).
	
	If present, the attribute’s value identifies the locales of string representations that can be supported by the context. The value consists of an array of structures. Each element in the array contains:
	`unsigned long code_set,`
	`unsigned long lang_terr`
	Arguments `code_set` and `lang_terr` together identify a locale. The values for the code sets are defined in the OSF code set registry currently defined in DCE RFC 40.1. This registry is being extended to include language / territory registrations.

ERRORS

[FN_E_ILLEGAL_NAME]
The name supplied to the operation was not a well-formed component according to the name syntax of the context.

[FN_E_INCOMPATIBLE_CODE_SETS]
Code set mismatches that occur during the construction of the compound name's string form are resolved in an implementation-dependent way. When an implementation discovers that a compound name has components with incompatible code sets, it returns the error code [FN_E_INCOMPATIBLE_CODE_SETS].

[FN_E_INCOMPATIBLE_LOCALES]
Mismatches in language or territory locale information that occur during the construction of the compound name's string form are resolved in an implementation-dependent way. When an implementation discovers that a compound name has components with incompatible locales, it returns the error code [FN_E_INCOMPATIBLE_LOCALES].

[FN_E_INVALID_SYNTAX_ATTRS]
The syntax attributes supplied are invalid or insufficient to fully specify the syntax.

[FN_E_SYNTAX_NOT_SUPPORTED]
The syntax specified is not supported.

APPLICATION USAGE

Most applications treat names as opaque data and hence, the majority of clients of the XFN interface will not need to parse compound names from specific naming systems. Some applications, however, such as browsers, need such capabilities. These applications would use *fn_ctx_get_syntax_attrs*( ) to obtain the syntax related attributes of a context and, if the context uses the XFN standard syntax model, it would examine these attributes to determine the name syntax of the context.

SEE ALSO

FN_attribute_t, **FN_attrset_t**, **FN_compound_t**, **FN_identifier_t**, **FN_string_t** , *fn_ctx_get_syntax_attrs*( ), **<xfn/xfn.h>**.

NAME

XFN links — an overview of XFN links

SYNOPSIS

```
#include <xfn/xfn.h>

FN_ref_t *fn_ref_create_link(const FN_composite_name_t *link_name);

int fn_ref_is_link(const FN_ref_t *ref);

FN_composite_name_t *fn_ref_link_name(const FN_ref_t *link_ref);

FN_ref_t *fn_ctx_lookup_link(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    FN_status_t *status);

unsigned int fn_status_link_code(const FN_status_t *stat);

const FN_composite_name_t *fn_status_link_remaining_name(
    const FN_status_t *stat);

const FN_composite_name_t *fn_status_link_resolved_name(
    const FN_status_t *stat);

const FN_ref_t *fn_status_link_resolved_ref(const FN_status_t *stat)

const FN_string_t* fn_status_link_diagnostic_message(
    const FN_status_t *stat);

int fn_status_set_link_code(FN_status_t *stat, unsigned int code);

int fn_status_set_link_remaining_name(
    FN_status_t *stat,
    const     FN_composite_name_t *name);

int fn_status_set_link_resolved_name(
    FN_status_t *stat,
    const     FN_composite_name_t *name);

int fn_status_set_link_resolved_ref(
    FN_status_t *stat,
    const FN_ref_t *ref);

int fn_status_set_diagnostic_message(
    FN_status_t *stat,
    const FN_string_t *msg);
```

DESCRIPTION

An *XFN link* is a special form of reference that contains a composite name, the *link name*, and that may be bound to an atomic name in an XFN context. Because the link name is a composite name, it may span multiple namespaces.

Normal resolution of names in context operations always follows XFN links. If the first composite name component of the link name is the atomic name ".", the link name is resolved relative to the same context in which the link is bound, otherwise, the link name is resolved relative to the XFN Initial Context of the client. The link name may itself cause resolution to pass through other XFN links. This gives rise to the possibility of a cycle of links whose resolution could not terminate normally. As a simple means to avoid such non-terminating resolutions, implementations may define limits on the number of XFN links that may be resolved in any single operation invoked by the caller.

Links are bound to names using the normal *fn_ctx_bind*( ) and unbound using the normal *fn_ctx_unbind*( ) operation. The operation *fn_ref_create_link*( ) is provided for constructing a link reference from a composite name. Since normal resolution always follows links, a separate operation, *fn_ctx_lookup_link*( ) is provided to lookup the link itself.

In the case that an error occurred while resolving an XFN link, the status object set by the operation contains additional information about that error and sets the corresponding link status fields using *fn_status_set_link_code*( ), *fn_status_set_link_remaining_name*( ), *fn_status_set_link_resolved_name*( ), *fn_status_set_link_resolved_ref*( ) and *fn_status_set_link_diagnostic_message*( ). The link status fields can be retrieved using *fn_status_link_code*( ), *fn_status_link_remaining_name*( ), *fn_status_link_resolved_name*( ), *fn_status_link_resolved_ref*( ) and *fn_status_link_diagnostic_message*( ).

ERRORS

The following status codes are of special relevance when performing operations involving XFN links:

[FN_E_LINK_ERROR]
There was an error encountered resolving an XFN link encountered during resolution of the supplied name. Check the link part of the status object to determine cause of the link error.

[FN_E_LINK_LOOP_LIMIT]
A non-terminating loop (cycle) in the resolution can arise due to XFN links encountered during the resolution of a composite name. This code indicates either the definite detection of such a cycle, or that resolution exceeded an implementation-defined limit on the number of XFN links allowed for a single operation invoked by the caller.

[FN_E_MALFORMED_LINK]
A malformed link reference was encountered.

APPLICATION USAGE

For the *fn_ctx_bind*( ), *fn_ctx_unbind*( ), *fn_ctx_rename*( ), *fn_ctx_lookup_link*( ), *fn_ctx_create_subcontext*( ), *fn_ctx_destroy_subcontext*( ), *fn_attr_bind*( ) and *fn_attr_create_subcontext*( ) operations, resolution of the given name continues to the target context — that named by all but the terminal atomic part of the given name; the terminal atomic name is not resolved. Consequently, for operations that involve unbinding the terminal atomic part such as *fn_ctx_unbind*( ), if the terminal atomic name is bound to a link, the link is not followed and the link itself is unbound from the terminal atomic name.

Operations in the base attribute interface that involve name resolution accept a *follow_link* parameter. The value of *follow_link* determines the behaviour of the operation when the terminal atomic part of the name being resolved is bound to an XFN link. If *follow_link* is non-zero, such a link is followed and the attribute associated with the final named object is examined or modified. If *follow_link* is zero, such a link is not followed. Any XFN links encountered before the terminal atomic name are always followed.

Many naming systems support a native notion of link that may be used within the naming system itself. XFN does not determine whether there is any relationship between such native links and XFN links.

SEE ALSO

FN_composite_name_t, **FN_ref_t**, **FN_status_t**, *fn_ctx_bind*( ), *fn_ctx_destroy_subcontext*( ), *fn_ctx_lookup*( ), *fn_ctx_lookup_link*( ), *fn_ctx_rename*( ), *fn_ctx_unbind*( ), XFN_status_codes, **<xfn/xfn.h>**.

NAME

XFN status codes — descriptions of XFN status codes

SYNOPSIS

```
#include <xfn/xfn.h>
```

DESCRIPTION

The result status of operations in the context interface and the attribute interfaces is encapsulated in an **FN_status_t** object. The caller may supply a NULL pointer for this parameter, in which case, no status information is returned. If the caller supplies an **FN_status_t** object to the operation, upon return from the operation, this object will contain information about how the operation completed: whether an error occurred in performing the operation, the nature of the error, and information that helps locate where the error occurred. In the case that the error occurred while resolving an XFN link, the status object contains additional information about that error.

The context status object consists of several items of information. One of them is the primary status code, describing the disposition of the operation. In the case that an error occurred while resolving an XFN link, the primary status code has the value [FN_E_LINK_ERROR], and link status code describes the error that occurred while resolving the XFN link.

Both the primary status code and the link status code are values of type **unsigned int** that are drawn from the same set of meaningful values. XFN reserves the values 0 through 127 for standard meanings. Currently values and interpretations for the following codes are determined by XFN.

Code	Meaning
FN_SUCCESS	The operation succeeded.
FN_E_ATTR_IN_USE	When an attribute is being modified using the operation FN_ATTR_OP_ADD_EXCLUSIVE and an attribute with the same identifier already exists, the operation fails with FN_E_ATTR_IN_USE.
FN_E_ATTR_NO_PERMISSION	The caller did not have permission to perform the attempted attribute operation.
FN_E_ATTR_VALUE_REQUIRED	The operation attempted to create an attribute without a value, and the specific naming system does not allow this.
FN_E_AUTHENTICATION_FAILURE	The identity of the client principal could not be verified.
FN_E_COMMUNICATION_FAILURE	An error occurred in communicating with one of the contexts involved in the operation.

Code	Meaning
FN_E_CONFIGURATION_ERROR	A problem was detected that indicated an error in the installation of the XFN implementation.
FN_E_CONTINUE	The operation should be continued using the remaining name and the resolved reference returned in the status.
FN_E_CTX_NO_PERMISSION	The client did not have permission to perform the operation.
FN_E_CTX_NOT_EMPTY	(Applies only to *fn_ctx_destroy_subcontext*( ).) The naming system required that the context be empty before its destruction, and it was not empty.
FN_E_CTX_UNAVAILABLE	Service could not be obtained from one of the contexts involved in the operation. This may be because the naming system is busy, or is not providing service. In some implementations this may not be distinguished from a communication failure.
FN_E_ILLEGAL_NAME	The name supplied to the operation was not a well-formed XFN composite name, or one of the component names was not well-formed according to the syntax of the naming system(s) involved in its resolution.
FN_E_INCOMPATIBLE_CODE_SETS	The operation involved character strings of incompatible code sets; or the supplied code set is not supported by the implementation.
FN_E_INCOMPATIBLE_LOCALES	The operation involved character strings of incompatible language or territory locale information, or the specified locale is not supported by the implementation.
FN_E_INSUFFICIENT_RESOURCES	Either the client or one of the involved contexts could not obtain sufficient resources (for example, memory, file descriptors, communication ports, stable media space, and so on.) to complete the operation successfully.
FN_E_INVALID_ATTR_IDENTIFIER	The attribute identifier was not in a format acceptable to the naming system, or its contents was not valid for the format specified for the identifier.

Code	Meaning
FN_E_INVALID_ATTR_VALUE	One of the values supplied was not in the appropriate form for the given attribute.
FN_E_INVALID_ENUM_HANDLE	The enumeration handle supplied was invalid, either because it was from another enumeration, or because an update operation occurred during the enumeration, or because of some other reason.
FN_E_INVALID_SYNTAX_ATTRS	The syntax attributes supplied are invalid or insufficient to fully specify the syntax.
FN_E_LINK_ERROR	There was an error encountered resolving an XFN link encountered during resolution of the supplied name.
FN_E_LINK_LOOP_LIMIT	A non-terminating loop (cycle) in the resolution can arise due to XFN links encountered during the resolution of a composite name. This code indicates either the definite detection of such a cycle, or that resolution exceeded an implementation-defined limit on the number of XFN links allowed for a single operation invoked by the caller.
FN_E_MALFORMED_LINK	A malformed link reference was encountered.
FN_E_MALFORMED_REFERENCE	A context object could not be constructed from the supplied reference, because the reference was not properly formed.
FN_E_NAME_IN_USE	(Only for operations that bind names.) The supplied name was already in use.
FN_E_NAME_NOT_FOUND	Resolution of the supplied composite name proceeded to a context in which the next atomic component of the name was not bound.
FN_E_NO_EQUIVALENT_NAME	No equivalent name can be constructed, either because there is no meaningful equivalence between *name* and *leading_name*, or the system does not support constructing the requested equivalent name, for implementation-specific reasons.

Code	Meaning
FN_E_NO_SUCH_ATTRIBUTE	The object did not have an attribute with the given identifier.
FN_E_NO_SUPPORTED_ADDRESS	A context object could not be constructed from a particular reference. The reference contained no address type over which the context interface was supported.
FN_E_NOT_A_CONTEXT	Either one of the intermediate atomic names did not name a context, and resolution could not proceed beyond this point, or the operation required that the caller supply the name of a context, and the name did not resolve to a reference for a context.
FN_E_OPERATION_NOT_SUPPORTED	The operation attempted is not supported.
FN_E_PARTIAL_RESULT	The operation attempted is returning a partial result.
FN_E_SEARCH_INVALID_FILTER	The filter expression had a syntax error or some other problem.
FN_E_SEARCH_INVALID_OP	An operator in the filter expression is not supported or, if the operator is an extended operator, the number of types of arguments supplied does not match the signature of the operation.
FN_E_SEARCH_INVALID_OPTION	A supplied search control option could not be supported.
FN_E_SYNTAX_NOT_SUPPORTED	The syntax type specified is not supported.
FN_E_TOO_MANY_ATTR_VALUES	The operation attempted to associate more values with an attribute than the naming system supported.
FN_E_UNSPECIFIED_ERROR	An error occurred that could not be classified by any of the other error codes.

SEE ALSO

FN_status_t, **<xfn/xfn.h>**.

NAME

<xfn/xfn.h> — header file for XFN interface

SYNOPSIS

```
#include <xfn/xfn.h>
```

DESCRIPTION

This header file contains the interface declarations, as defined by this specification, for

1. the XFN base context interface
2. the XFN extended attribute interface
3. the XFN base attribute interface
4. status object and status codes used by operations in these three interfaces
5. abstract data types passed as parameters to and returned as values from operations in these three interfaces
6. the interface for the XFN standard syntax model for parsing compound names.

SEE ALSO

FN_ctx_t, **FN_status_t**, XFN_attribute_operations, XFN_compound_syntax, XFN_status_codes.

Appendix A

XFN Protocols: Preliminary Specification

The whole of Appendix A is assigned X/Open **Preliminary Specification** status (not **CAE Specification** status). For explanation of the difference between **Preliminary** and **CAE** specifications, see the description under **X/Open Technical Publications** in the **Preface**. Readers should appreciate that the *header* title on each page of this Appendix correctly identifies its status as **Preliminary**, while the *footer* on each odd-numbered pages also correctly identifies this Appendix as a part of the **XFN CAE Specification**.

This Appendix contains definitions of the XFN protocol for the DCE and ONC+ platforms.

An XFN implementation is not required to support any particular XFN protocol but if a protocol for a particular platform is supported, it must comply with the behaviour specified herein.

Three different types of servers are expected to export an XFN protocol:

- a legacy naming system that exports its own native protocol as well as an XFN protocol
- a new naming system that only exports an XFN protocol
- a server that is a surrogate XFN client.

These types of servers are described in more detail Appendix C.

A.1 DCE RPC Protocol for XFN

This section defines the RPC interface of the XFN service for DCE platforms. The interface is specified in eight IDL files:

fn_dce_ctxb.idl	Definition of the data types for the XFN base context interface.
fn_dce_ctx.idl	Definition of the XFN base context interface except for create and destroy context operations.
fn_dce_ctx_mgmt.idl	Definition of the XFN context create and destroy interface.
fn_dce_attrb.idl	Definition of the data types for the XFN base attribute interface.
fn_dce_attr.idl	Definition of the XFN base attribute interface.
fn_dce_srchb.idl	Definition of the data types for the XFN search interface.
fn_dce_ctx_locate.idl	Definition of the XFN context location interface.
fn_dce_srch.idl	Definition of the XFN attribute search interface.

See X/Open DCE RPC for descriptions of DCE RPC and IDL.

A.1.1 fn_dce_ctxb.idl: Data Types for Context Interface

```
[
pointer_default(ptr)
]

interface fn_dce_ctxb {

import "dce/nbase.idl";

/*
 * Basic context datatypes
 */

typedef struct {
    unsigned32              len;
    [size_is(len)] byte     byte_array[];
} fn_dce_byte_str_t, *fn_dce_byte_str_p_t;

/*
 * XFN string
 *
 * "code_set" and "lang_terr" identify the string's locale.
 *
 * All XFN implementations must support the "C" locale.
 * The "C" locale's repertoire of characters is restricted to that
 * defined by the portable representation of ISO-646 (same encoding
 * as ASCII).
 *
 * Support for non-ASCII code sets is implementation-dependent.
 */

typedef struct {
    unsigned32              code_set;
```

```
    unsigned32              lang_terr;
    unsigned32              charcount;
    fn_dce_byte_str_p_t     string;
} fn_dce_string_t, *fn_dce_string_p_t;

typedef struct {
    unsigned32              num_comps;
    [size_is(num_comps)]
    fn_dce_string_p_t       components[];
} fn_dce_composite_name_t, *fn_dce_composite_name_p_t;

typedef [context_handle] void    *fn_dce_cursor_t;

/*
 * Status report
 *
 * standard DCE error and some additional, optional text
 */

typedef struct {
    error_status_t          status;
    fn_dce_string_p_t       diag_msg;
} fn_dce_status_t;

/*
 * Identifier
 */

typedef struct {
    unsigned32              format;
    fn_dce_byte_str_p_t     contents;
} fn_dce_id_t, *fn_dce_id_p_t;

/*
 * Address
 */

typedef struct {
    fn_dce_id_t             addr_type;
    fn_dce_byte_str_p_t     address;
} fn_dce_addr_t, *fn_dce_addr_p_t;

/*
 * Object Reference
 *
 * A name in an XFN context must be bound to a reference.
 * The reference may be an XFN NULL reference which has the
 * following values:
 *
 * ref_type    FN_ID_STRING    format
 *             "fn_null_ref"   value
 * num_addrs   0
```

```
 */

typedef struct {
    fn_dce_id_t            ref_type;
    unsigned32             num_addrs;
    [size_is(num_addrs)]
    fn_dce_addr_p_t        addrs[];
} fn_dce_ref_t, *fn_dce_ref_p_t;

/*
 * Binding
 */

typedef struct {
    fn_dce_string_p_t      name;
    fn_dce_ref_p_t         ref;
} fn_dce_binding_t;

/*
 * Caching information:
 * A caching hint is passed from the client to the server with
 * each reference and attribute that is read.
 */

typedef enum {
    /*
     * unspecified
     */
    cache_unspec,

    /*
     * don't cache
     */
    dont_cache,

    /*
     * already cached
     */
    already_cached,

    /*
     * cache until failure
     */
    cache_til_fail,

    /*
     * cache for time to live seconds
     */
    cache_ttl

} fn_dce_cache_type_t;
```

```
/*
 * Cache information:
 *
 * A caching hint is passed from the server to the client with
 * each reference and attribute that is read.
 *
 * cache_type    kind of caching, if any, that should be done.
 *
 * ttl           if type is cache_ttl, number of seconds that
 *               the data should be cached.
 *
 * public        TRUE if data may be read by any principal;
 *               otherwise false.
 */

typedef struct {
    fn_dce_cache_type_t     type;
    unsigned32              ttl;
    boolean32               public;
} fn_dce_cache_info_t, *fn_dce_cache_info_p_t;

/*
 * Most operations take a starting context and composite name,
 * relative to the starting context, to identify the target
 * of the operation.  Before the real work of the operation
 * can be done, the operation's target name must be resolved.
 * The progress record is used to pass information about
 * the path resolution phase of an operation between
 * the client and server.
 *
 *************  On Input  ************************************
 *
 * ref           reference of starting context
 *
 * unresolved_name
 *               composite name to be resolved relative to "ref"
 *
 * resolved_name
 *               NULL or a composite name
 *
 * link_ctx_ref  NULL
 *
 *************  Intermediate Output  **************************
 *
 * Sometimes path resolution will traverse more than one server
 * and will use a referral method to carry the resolution from
 * one server to the next.  In such cases, the intermediate
 * servers must return the following information to the client;
 * the client will then invoke the next server to continue resolution.
 *
 * When an XFN link is encountered during path resolution,
 * it is returned to the client for further resolution.
```

```
 *
 * resolved_name
 *               the part of the composite name that was resolved
 *
 * ref           reference that is bound to "resolved_name"
 *
 * unresolved_name
 *               the part of the name that remains to be resolved.
 *
 * cache_info    caching hints for the resolved name and its reference
 *
 * link_ctx_ref  if "resolved_name" is bound to an XFN link,
 *               the reference of the context in which "resolved_name"
 *               is bound.
 *               Otherwise NULL.
 *
 ***************** Final Output ************************************
 *
 * When the name has been completely resolved the
 * progress record contains:
 *
 * resolved_name
 *               depends on the operation
 *
 * ref           reference that is bound to "resolved_name".
 *
 * unresolved_name
 *               NULL
 *
 * cache_info    caching hints for the resolved name and its reference
 *
 * link_ctx_ref  if "resolved_name" is bound to an XFN link,
 *               the reference of the context in which "resolved_name"
 *               is bound.
 *               Otherwise NULL.
 *
 * Implementations that support the FN DCE protocol must support
 * the path resolution phase of all operations even though they
 * may not support the actual operation.
 */

typedef struct {
    fn_dce_ref_p_t               ref;
    fn_dce_composite_name_p_t    unresolved_name;
    fn_dce_composite_name_p_t    resolved_name;
    fn_dce_cache_info_t          cache_info;
    fn_dce_ref_p_t               link_ctx_ref;
} fn_dce_progress_t, *fn_dce_progress_p_t;

}
```

A.1.2 fn_dce_ctx.idl: Context Interface

```
[
uuid(80981362-aba2-11cc-87d4-08000932b6f8),
version(1.0),
pointer_default(ptr)
]

interface fn_dce_ctx {

import "fn_dce_ctxb.idl";

/*
 * Context interface (DCE RPC)
 *
 * The FN context interface specifies operations to lookup a composite
 * name and get its reference.  The interface also includes operations
 * to bind a name to a reference, to unbind a name, to rename, to get
 * the list of names bound in a context, and to get a list of the
 * bindings in a context.
 *
 * Most operations in the context interface take the following two
 * initial arguments:
 *
 *  h             a handle that identifies the server to which
 *                the operation should be directed.
 *
 *  progress      structure that guides the path resolution phase of
 *                an operation.
 *
 *  input:
 *
 *  ref           reference of starting context
 *
 *  unresolved_name
 *                target name for the operation, relative to "ref"
 *
 *
 *  resolved_name
 *                NULL
 *
 *  link_ctx_ref  NULL
 *
 *
 *  final output:
 *
 *  resolved_name
 *                depends on operation (described with each operation)
 *
 *  ref           reference that is bound to "resolved_name"
 *
 *  unresolved_name
 *                NULL
```

```
 *
 *  cache_info    caching hints for the resolved name and its reference
 *
 *  link_ctx_ref  if "resolved_name" is bound to an XFN link,
 *                the reference of the context in which "resolved_name"
 *                is bound.
 *                Otherwise NULL.
 *
 *      fn_dce_ctxb.idl describes the use of the progress structure.
 */


/*
 * F N _ D C E _ C T X _ B I N D
 *
 * Bind a name to a reference.
 *
 * Name is specified in "progress".  The binding is made in the
 * target context, that context named by all but the final atomic
 * part of the supplied name.  The operation binds the final
 * atomic name to the supplied reference in the target context.
 *
 * The target context must already exist.
 *
 *  progress      final output:
 *                  resolved_name    target context
 *                (other fields described above)
 *
 *  ref           reference to be bound to name
 *                A name in an XFN context must be bound to a
 *                reference.  The reference may be an XFN NULL
 *                reference, as specified in fn_dce_ctxb.idl.
 *
 *  exclusive     if TRUE, the operation fails if the name is
 *                already bound.  If FALSE, the new binding
 *                supersedes any existing binding, and any
 *                attributes associated with the name are removed.
 */
void fn_dce_ctx_bind(
    [in] handle_t                  h,
    [in, out] fn_dce_progress_t    *progress,
    [in] fn_dce_ref_p_t            ref,
    [in] boolean32                 exclusive,
    [out] fn_dce_status_t          *status
);

/*
 * F N _ D C E _ C T X _ U N B I N D
 *
 * Unbind a name from a context.  If any attributes are associated
 * with the name, this operation removes the name and its attributes.
 *
```

```
 * The operation succeeds if the name is not bound in the context.
 *
 *  progress      output:
 *                  resolved_name    target context
 *                (other fields described above)
 */
void fn_dce_ctx_unbind(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [out] fn_dce_status_t           *status
);

/*
 * F N _ D C E _ C T X _ L O O K U P
 *
 * Lookup a composite name.  Return a reference to the named object.
 *
 *  progress      final output:
 *                  resolved_name    target name
 *                (other fields described above)
 */
void fn_dce_ctx_lookup(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [out] fn_dce_status_t           *status
);

/*
 * F N _ D C E _ C T X _ L I S T _ N A M E S
 *
 * Return the list of names that are bound in a context.
 *
 *  progress      final output:
 *                  resolved_name    target context
 *                (other fields described above)
 *
 *  iter_pos      a cursor into the context's list of names. It is
 *                maintained by the server.
 *                Its state is initialized by the server at the first
 *                request (client passes in a pointer to NULL) and
 *                freed when the last name has been returned.
 *                Its state may also be freed by the operation
 *                fn_dce_ctx_free_iterator.
 *
 *  max_names     maximum length of the output array "names"
 *
 *  num_names     count of names listed in "names"
 *
 *  names         list of atomic names that are bound in the context.
 */

void fn_dce_ctx_list_names(
```

```
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [in, out] fn_dce_cursor_t       *iter_pos,
    [in] unsigned32                 max_names,
    [out] unsigned32                *num_names,
    [out, size_is(max_names), length_is(*num_names)]
        fn_dce_string_p_t           names[],
    [out] fn_dce_status_t           *status
);

/*
 * F N _ D C E _ C T X _ L I S T _ B I N D I N G S
 *
 * Return the set of bindings present in a context.  A binding
 * consists of an atomic name and the object reference that
 * is bound to it.
 *
 *  progress      final output:
 *                  resolved_name    target context
 *                (other fields described above)
 *
 *  iter_pos      a cursor into the context's list of bindings.
 *                It is maintained by the server.  Its state is
 *                initialized by the server at the first request
 *                (client passes in a pointer to NULL) and freed
 *                when the last binding has been returned.
 *                Its state may also be freed by the operation
 *                fn_dce_ctx_free_iterator.
 *
 *  max_bindings  maximum length of the output array "bindings"
 *
 *  num_bindings  count of bindings listed in "bindings"
 *
 *  bindings      list of bindings in the context.
 */

void fn_dce_ctx_list_bindings(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [in, out] fn_dce_cursor_t       *iter_pos,
    [in] unsigned32                 max_bindings,
    [out] unsigned32                *num_bindings,
    [out, size_is(max_bindings), length_is(*num_bindings)]
        fn_dce_binding_t            bindings[],
    [out] fn_dce_status_t           *status
);

/*
 * F N _ D C E _ C T X _ F R E E _ I T E R A T O R
 *
 * Free any state associated with the list cursor "iter_pos".
 *
```

```
 *  ctx           reference of context with which the list cursor
 *                is associated.
 *
 *  iter_pos      list cursor whose state is to be freed.
 */
void fn_dce_ctx_free_iterator(
    [in] handle_t                h,
    [in] fn_dce_ref_p_t          ctx,
    [in, out] fn_dce_cursor_t    *iter_pos,
    [out] fn_dce_status_t        *status
);

/*
 * F N _ D C E _ C T X _ R E N A M E
 *
 * Bind the reference and attributes associated with old_name to
 * new_name and unbind old_name.
 *
 * This operation fails if old_name does not exist.
 *
 * If the operation fails, old_name remains bound to its reference
 * and attributes.
 *
 * XFN does not specify whether this operation is atomic; that is
 * left to the underlying name system.  Also, XFN does not specify
 * the restrictions placed on new_name.  In some name systems,
 * new_name might be restricted to be a name in the same naming
 * system as the final component of old_name.  In others new_name
 * might be restricted to be a name in the same context in which
 * the final atomic part of old_name is bound.
 *
 *  h             a handle that identifies the server to which the
 *                operation should be directed.
 *
 *  old_name_progress
 *                name whose reference and attributes will be bound
 *                with new_name.  See above for a description of
 *                the fields and use of the progress argument.
 *
 *                final output:
 *                  resolved_name    target context of "old_name"
 *
 *  new_name_ctx  target context in which new_name will be bound.
 *
 *  new_name      atomic name which will be bound to old_name's
 *                reference and attributes in new_name_ctx context.
 *
 *  exclusive     if TRUE, the operation fails if new_name is
 *                already bound.  If FALSE, old_name's reference
 *                and attributes supersede any existing information
 *                associated with new_name.
 *
```

```
 *  status        the status of the operation, returned by the server.
 *                  FN_DCE_E_CTX_NOT_LOCAL
 *                  FN_DCE_E_RENAME_TARGET_CTXS_DIFFERENT
 *
 *                Other possible status codes: TBD
 */
void fn_dce_ctx_rename(
    [in] handle_t                  h,
    [in, out] fn_dce_progress_t    *old_name_progress,
    [in] fn_dce_ref_p_t            new_name_ctx,
    [in] fn_dce_string_p_t         new_name,
    [in] boolean32                 exclusive,
    [out] fn_dce_status_t          *status
);

}
```

A.1.3 fn_dce_ctx_mgmt.idl: Context Management Interface

```
[
uuid(8381dc5e-0e9c-11cd-bd6e-08000932b6f8),
version(1.0),
pointer_default(ptr)
]

interface fn_dce_ctx_mgmt {

import "fn_dce_ctxb.idl";
import "fn_dce_attrb.idl";
import "dce/aclbase.idl";

/*
 * Context management interface (DCE RPC)
 *
 * The FN context management interface specifies
 * operations to create and destroy contexts.
 *
 * Some operations in the context management
 * interface take the following two initial
 * arguments:
 *
 *  h             a handle that identifies the server to which
 *                the operation should be directed.
 *
 *  progress      structure that guides the path resolution phase
 *                of an operation.
 *
 *  input:
 *
 *  ref           reference of starting context
 *
 *  unresolved_name
 *                target name for the operation, relative to "ref"
```

```
 *
 *
 *  resolved_name
 *                NULL
 *
 *  link_ctx_ref  NULL
 *
 *  progress      final output:
 *                  resolved_name    target context.
 *                The target context is the context named by all
 *                but the final atomic part of the supplied name.
 *
 *  ref           reference that is bound to "resolved_name".
 *
 *  unresolved_name
 *                NULL
 *
 *  cache_info    caching hints for the resolved name and its reference
 *
 *  link_ctx_ref  NULL
 *
 *        fn_dce_ctxb.idl describes the use of the progress structure.
 */

/*
 * F N _ D C E _ C T X _ M G M T _ C R E A T E _ S U B C O N T E X T
 *
 * Create a context, bind its reference to the name specified
 * in "progress", and associate any supplied attributes with
 * the new context.
 *
 * The target context, that context named by all but the final atomic
 * part of the supplied name, must already exist.  The new context
 * is created and its reference is bound in the target context using
 * the final atomic name.
 *
 * The new context is created at the same nameserver as the target
 * context and inherits the target context's acls.  This operation
 * fails if "name" already exists in the target context.
 *
 *
 *  num_attrs     count of attributes in "attrs".
 *                0 if attrs is NULL.
 *
 *  attrs         attributes to be associated with the new context.
 *
 *  ref           reference to the new context.
 *
 *  cache_info    contains caching hints for the new context's name
 *                and its reference.
 */
```

```
void fn_dce_ctx_create_subcontext(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [in] unsigned32                 num_attrs,
    [in, size_is(num_attrs)]        fn_dce_attr_p_t    attrs[],
    [out] fn_dce_ref_p_t            *ref,
    [out] fn_dce_cache_info_t       cache_info,
    [out] fn_dce_status_t           *status
);

/*
 * F N _ D C E _ C T X _ M G M T _ D E S T R O Y _ S U B C O N T E X T
 *
 * Destroy a subcontext and unbind its name, which is specified
 * in "progress".
 *
 * This operation fails if the subcontext is not empty.
 */

void fn_dce_ctx_destroy_subcontext(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [out] fn_dce_status_t           *status
);

/*
 * F N _ D C E _ C T X _ M G M T _ C R E A T E _ C O N T E X T
 *
 * Create a context and associate the specified
 * acls and attributes with the new context.
 *
 *  ctx_server      reference of the server where the context should
 *                  be created.
 *
 *  acl_list        initial acls for the new context
 *
 *  num_attrs       count of attributes in "attrs".
 *                  0 if attrs is NULL.
 *
 *  attrs           attributes to be associated with the new context.
 *
 *  ref             reference to the new context.
 */
void fn_dce_ctx_create_context(
    [in] handle_t               h,
    [in] fn_dce_ref_p_t         ctx_server,
    [in] sec_acl_list_t         *acl_list,
    [in] unsigned32             num_attrs,
    [in, size_is(num_attrs)]    fn_dce_attr_p_t    attrs[],
    [out] fn_dce_ref_p_t        *ref,
    [out] fn_dce_status_t       *status
);
```

```
/*
 * F N _ D C E _ C T X _ M G M T _ D E S T R O Y _ C O N T E X T
 *
 * Destroy a context. This operation fails if the context is not empty.
 *
 *  target_ctx     the context to be destroyed.
 */

void fn_dce_ctx_destroy_context(
    [in] handle_t            h,
    [in] fn_dce_ref_p_t      target_ctx,
    [out] fn_dce_status_t    *status
);

}
```

A.1.4 fn_dce_attrb.idl: Data Types for Attribute Interface

```
[ pointer_default(ptr) ]

interface fn_dce_attrb {

import "fn_dce_ctxb.idl";


/*
 * Common Data Types for Attribute Interfaces
 */

/*
 * Locale identifier
 */
typedef struct {
    unsigned32            code_set;
    unsigned32            lang_terr;
} fn_dce_attr_locale_t;

typedef struct {
    unsigned32            num_locales;
    [size_is(num_locales)] fn_dce_attr_locale_t
                          locales[];
} fn_dce_attr_locales_t, *fn_dce_attr_locales_p_t;


/* attribute value syntaxes  */

typedef enum {
    fn_dce_attr_val_stx_int,
    fn_dce_attr_val_stx_uns_int,
    fn_dce_attr_val_stx_uuid,
    fn_dce_attr_val_stx_str,
    fn_dce_attr_val_stx_fn_str,
    fn_dce_attr_val_stx_fn_name,
```

```
    fn_dce_attr_val_stx_bytes,
    fn_dce_attr_val_stx_locales
} fn_dce_attr_value_syntax_t;


/* attribute value */

typedef union switch ( fn_dce_attr_value_syntax_t syntax_of_value )
    syntax {
    case fn_dce_attr_val_stx_int:
        signed32                  integer;

    case fn_dce_attr_val_stx_uns_int:
        unsigned32                unsigned_integer;

    case fn_dce_attr_val_stx_uuid:
        uuid_t                    uuid;

    case fn_dce_attr_val_stx_str:
    [string] unsigned char        *string;

    case fn_dce_attr_val_stx_fn_str:
        fn_dce_string_p_t         fn_string;

    case fn_dce_attr_val_stx_fn_name:
        fn_dce_composite_name_p_t  fn_name;

    case fn_dce_attr_val_stx_bytes:
        fn_dce_byte_str_p_t       bytes;

    case fn_dce_attr_val_stx_locales:
        fn_dce_attr_locales_p_t   locales;

} fn_dce_attr_value_t, *fn_dce_attr_value_p_t;


/* attribute ( identifier/syntax/value(s) ) */

typedef struct {
    fn_dce_id_t        attrib_id;
    fn_dce_id_t        attrib_syntax;
    unsigned32         num_vals;
    [size_is(num_vals)] fn_dce_attr_value_t
                       attrib_values[];
} fn_dce_attr_t, *fn_dce_attr_p_t;

typedef struct {
    fn_dce_status_t      status;
    fn_dce_cache_info_t  cache_info;  /* caching hints for attribute */
    fn_dce_attr_p_t      attrib;
} fn_dce_attr_bulk_get_t, *fn_dce_attr_bulk_get_p_t;
```

```
/*
 * Operations on an attribute that is associated with a name
 * in a context
 */

typedef enum {
    /*
     * Add an attribute with the given identifier and set of values.
     * If the attribute already exists, replace its value(s).
     */
    fn_dce_attr_op_add,

    /*
     * Add an attribute with the given identifier and set of values.
     * If the attribute already exists, return an error.
     */
    fn_dce_attr_op_add_exclusive,

    /*
     * Remove the attribute, with the given identifier, and all its
     * values. Succeeds even if the attribute does not exist.
     */
    fn_dce_attr_op_remove,

    /*
     * Add the given values to those of the given attribute, resulting
     * in the attribute having the union of its prior value set with
     * the set given.
     * Create the attribute if the attribute does not already exist.
     */
    fn_dce_attr_op_add_values,

    /*
     * Remove the given values from those of the given attribute,
     * resulting in the attribute having the set difference of its
     * prior value set and the set given.
     * This succeeds even if some of the values are not in the set
     * of values that the attribute has.
     * In naming systems that require an attribute to have at least
     * one value, removing the last value will remove the attribute
     * as well.
     */
    fn_dce_attr_op_remove_values

} fn_dce_attr_op_t;

typedef struct {
    /*
     * Operation to be performed on an attribute
     */
    fn_dce_attr_op_t    operation;
```

```
    /*
     * Attribute identifier, syntax, and value(s) for the operation
     */
    fn_dce_attr_p_t     attribute;

} fn_dce_attr_modify_t;

}
```

A.1.5 fn_dce_attr.idl: Attribute Interface

```
[
uuid(da605554-ef53-11cc-8c52-08000932b6f8),
version(1.0),
pointer_default(ptr)
]

interface fn_dce_attr {

import "fn_dce_ctxb.idl";
import "fn_dce_attrb.idl";

/*
 * Attribute interface (DCE RPC)
 *
 * The attribute interface provides operations to read and modify
 * the attributes associated with a name.
 *
 * XFN recommends that attributes which are used to store a reference
 * are not examined or modified through the attribute interface.
 */

/*
 * Most operations in the attribute interface take the following
 * three initial arguments:
 *
 *  h             a handle that identifies the server to which
 *                the operation should be directed.
 *
 ************************************************************************
 *
 *  progress      a structure that guides the path resolution phase
 *                of an operation.
 *
 *  input:
 *
 *  ref           reference of starting context
 *
 *  unresolved_name
*                 target name for the operation, relative to "ref"
 *
 *
 *  resolved_name
```

```
 *                NULL
 *
 *  link_ctx_ref  NULL
 *
 *  final output:
 *
 *  resolved_name
 *                normally the target context.
 *
 *                For all operations except fn_dce_attr_get_ctx_syntax()
 *                and fn_dce_attr_get_ctx_locales(), the target context
 *                is the context named by all but the final atomic part
 *                of the supplied name.
 *
 *                For operations that return an iterator,
 *                "resolved_name" and its "ref" will be used to free
 *                the iterator.  If the target context is not the
 *                target for the free iterator operation, then the
 *                target for that operation should be returned in
 8                "resolved_name" and "ref".
 *
 *  ref           reference that is bound to "resolved_name".
 *
 *  unresolved_name
 *                NULL
 *
 *  cache_info    caching hints for the resolved name and its reference
 *
 *  link_ctx_ref  if "resolved_name" is bound to an XFN link, the
 *                reference of the context in which "resolved_name"
 *                is bound.
 *                Otherwise NULL.
 *
 *       fn_dce_ctxb.idl describes the use of the progress structure.
 *
 ************************************************************************
 *
 *  follow_link   The value of "follow_link" determines the behaviour
 *                of the operation when the terminal atomic part of
 *                the name being resolved is bound to an XFN link.
 *
 *                If "follow_link" is TRUE, such a link is returned
 *                for further resolution.  If "follow_link" is FALSE,
 *                the attributes associated with the name bound to
 *                the link are examined or modified.
 *
 *                Any XFN links encountered before the terminal atomic
 *                name are always followed.
 */


/*
```

```
 * F N _ D C E _ A T T R _ B I N D
 *
 * Bind a name to a reference and associate attributes with the name.
 *
 * Name is specified in "progress".
 *
 * The binding is made in the target context, that context named by
 * all but the final atomic part of the supplied name.  The operation
 * binds the final atomic name to the supplied reference and
 * associates the specified attributes with the name.
 *
 * The target context must already exist.
 *
 *  ref         reference to be bound to name.
 *
 *              A name in an XFN context must be bound to a reference.
 *              The reference may be an XFN NULL reference, as
 *              specified in fn_dce_ctxb.idl.
 *
 *  num_attrs   count of attributes in "attrs".
 *              0 if attrs is NULL.
 *
 *  attrs       attributes to be associated with name.
 *
 *  exclusive   if TRUE, the operation fails if the name is already
 *              bound in the target context.
 *              If FALSE, the new binding supersedes an existing
 *              binding.  If "attrs" is NULL, any attributes
 *              associated with the name are not changed.
 *              If attributes are specified in "attrs" these
 *              attributes replace any attributes associated with
 *              the name.
 */

void fn_dce_attr_bind(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [in] fn_dce_ref_p_t             ref,
    [in] unsigned32                 num_attrs,
    [in, size_is(num_attrs)]        fn_dce_attr_p_t   attrs[],
    [in] boolean32                  exclusive,
    [out] fn_dce_status_t           *status
);


/*
 * F N _ D C E _ A T T R _ G E T
 *
 * The operation that looks up a name in a context and returns the
 * specified attribute associated with the name.  This operation
 * can be called multiple times if all of the attribute's values
 * cannot fit in the returned list of values, until the last value
```

```
 * has been returned.
 *
 * attr_id      identifies the attribute whose values are to be
 *              returned.
 *
 * max_num_vals
 *              the maximum length of the output array "values"
 *
 * max_vals_size
 *              the maximum aggregate size of the buffers that return
 *              the attribute's values.  If zero, any buffer size
 *              may be returned.
 *
 * iter_pos     a cursor into the list of values associated with
 *              "attr_id", maintained by the server. Its state is
 *              initialized by the server at the first request
 *              (client passes in a pointer to NULL) and freed when
 *              the last value has been returned. Its state may also
 *              be freed by the operation fn_dce_ctx_free_iterator().
 *
 * attr_syntax  the syntax of the attribute's values.
 *
 * attr_cache_info
 *              caching hints for the attribute's values.
 *
 * num_vals     the count of values returned in "values"
 *
 * values       the list of values
 *
 * status       the status of the operation, returned by the server.
 *              FN_DCE_E_GET_ATTRS_FROM_OBJECT
 *              Other possible status codes: TBD
 */

[ idempotent ] void fn_dce_attr_get(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [in] boolean32                  follow_link,
    [in] fn_dce_id_p_t              attr_id,
    [in] unsigned32                 max_num_vals,
    [in] unsigned32                 max_vals_size,
    [in, out] fn_dce_cursor_t       *iter_pos,
    [out] fn_dce_id_t               *attr_syntax,
    [out] fn_dce_cache_info_t       *attr_cache_info,
    [out] unsigned32                *num_vals,
    [out, size_is(max_num_vals), length_is(*num_vals)]
        fn_dce_attr_value_t         values[],
    [out] fn_dce_status_t           *status
);


/*
```

```
 * F N _ D C E _ A T T R _ G E T _ I D S
 *
 * The operation that looks up a name in a context and returns
 * the identifiers of the attributes associated with the name.
 * This operation can be called multiple times if the buffer size
 * is exceeded, until the last attribute id has been returned.
 *
 * max_num_attr_ids
 *              the maximum size of the output array "attr_ids".
 *
 * iter_pos     a cursor into the list of attribute ids associated
 *              with "name", maintained by the server. Its state is
 *              initialized by the server at the first request
 *              (client passes in a pointer to NULL) and freed when the
 *              last identifier has been returned. Its state may also
 *              be freed by the operation fn_dce_ctx_free_iterator().
 *
 * num_attr_ids
 *              the count of attribute ids returned in "attr_ids".
 *
 * attr_ids     the list of attribute ids
 *
 * status       the status of the operation, returned by the server.
 *              Possible status codes: TBD
 */

[ idempotent ] void fn_dce_attr_get_ids(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [in] boolean32                  follow_link,
    [in] unsigned32                 max_num_attr_ids,
    [in, out] fn_dce_cursor_t       *iter_pos,
    [out] unsigned32                *num_attr_ids,
    [out, size_is(max_num_attr_ids), length_is(*num_attr_ids)]
        fn_dce_id_t                 attr_ids[],
    [out] fn_dce_status_t           *status
);


/*
 * F N _ D C E _ A T T R _ M O D I F Y
 *
 * The operation that adds, deletes, or modifies an attribute
 * associated with a name.  The name must exist in the context.
 *
 * mod_op       the modify operation that is to be applied to "attr".

 * attr         the attribute identifier, its syntax, and values.
 *
 *              When deleting an attribute, IDL requires that all the
 *              attribute's fields be filled with data that is valid
 *              for transmission.  This means that the attrib_syntax
```

```
 *             must be set to a NULL char string pointer and that the
 *             num_vals field must be set to zero.
 *
 * status      the status of the operation, returned by the server.
 *             Possible status codes: TBD
 */

void fn_dce_attr_modify(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [in] boolean32                  follow_link,
    [in] fn_dce_attr_op_t           mod_op,
    [in] fn_dce_attr_p_t            attr,
    [out] fn_dce_status_t           *status
);


/*
 * F N _ D C E _ A T T R _ M U L T I _ G E T
 *
 * The operation that looks up a name in a context and returns
 * the specified attributes or all of the attributes associated
 * with the name.
 *
 * This operation can be called multiple times if either of the
 * buffer sizes is exceeded, until the last attribute has been
 * returned.  The arguments "num_attr_ids" and "attr_ids" must
 * be the same for all iterations of this call; otherwise the
 * results are indeterminate.
 *
 * num_attr_ids
 *            the count of attribute identifiers listed in attr_ids.
 *            0 if all attributes associated with the name are to be
 *            returned.
 *
 * attr_ids   identifies the attributes that are to be returned.
 *            NULL if all attributes are to be returned.
 *
 * max_num_attrs
 *            the maximum length of the output array "attribs".
 *
 * max_vals_size
 *            the maximum aggregate size of the buffers that return
 *            the attributes' values.  If zero, any buffer size may
 *            be returned.
 *
 * iter_pos   a cursor into the name's attribute list, maintained
 *            by the server.  Its state is initialized by the server
 *            at the first request (client passes in a pointer to
 *            NULL) and freed when the last attribute has been
 *            returned.  Its state may also be freed by the
 *            operation fn_dce_ctx_free_iterator.
```

```
 *
 * num_attrs  the count of attributes listed in "attribs".
 *
 * attribs    a list of the name's attribute ids with each attribute's
 *            syntax and values.  If an attribute's syntax and values
 *            could not be returned, an error status is returned with
 *            the attribute id.
 *
 * status     the status of the operation, returned by the server.
 *            FN_DCE_E_GET_ATTRS_FROM_OBJECT
 *            Other possible status codes: TBD
 */

[ idempotent ] void fn_dce_attr_multi_get(
    [in] handle_t                  h,
    [in, out] fn_dce_progress_t    *progress,
    [in] boolean32                 follow_link,
    [in] unsigned32                num_attr_ids,
    [in, size_is(num_attr_ids)]
        fn_dce_id_t                attr_ids[],
    [in] unsigned32                max_num_attrs,
    [in] unsigned32                max_vals_size,
    [in, out] fn_dce_cursor_t      *iter_pos,
    [out] unsigned32               *num_attrs,
    [out, size_is(max_num_attrs), length_is(*num_attrs)]
        fn_dce_attr_bulk_get_t     attribs[],
    [out] fn_dce_status_t          *status
);

/*
 * F N _ D C E _ A T T R _ G E T _ C T X _ S Y N T A X
 *
 * The operation that gets the syntax attributes associated with the
 * context, named in "progress".  The attribute set must contain the
 * attribute "syntax type".
 *
 * This operation can be called multiple times if either of the
 * buffer sizes is exceeded, until the last attribute has been
 * returned.
 *
 * max_num_attrs
 *            the maximum length of the output array "attribs".
 *
 * max_vals_size
 *            the maximum aggregate size of the buffers that return
 *            the attributes' values.  If zero, any buffer size may
 *            be returned.
 *
 * iter_pos   a cursor into the name's attribute list, maintained
 *            by the server.  Its state is initialized by the server
 *            at the first request (client passes in a pointer to
 *            NULL) and freed when the last attribute has been
```

```
 *            returned.  Its state may also be freed by the operation
 *            fn_dce_ctx_free_iterator().
 *
 * num_attrs  the count of attributes listed in "attribs".
 *
 * attribs    a list of the name's attribute ids with each
 *            attribute's syntax, values, and caching hint.  If an
 *            attribute's syntax and values could not be returned,
 *            an error status is returned with the attribute id.
 *
 * status     the status of the operation, returned by the server.
 *            FN_DCE_E_GET_ATTRS_FROM_OBJECT
 *            Other possible status codes: TBD
 */

[ idempotent ] void fn_dce_attr_get_ctx_syntax(
    [in] handle_t                  h,
    [in, out] fn_dce_progress_t    *progress,
    [in] unsigned32                max_num_attrs,
    [in] unsigned32                max_vals_size,
    [in, out] fn_dce_cursor_t      *iter_pos,
    [out] unsigned32               *num_attrs,
    [out, size_is(max_num_attrs), length_is(*num_attrs)]
        fn_dce_attr_bulk_get_t     attribs[],
    [out] fn_dce_status_t          *status
);

/*
 * F N _ D C E _ A T T R _ M U L T I _ M O D I F Y
 *
 * The operation that adds, deletes, and/or modifies one or more
 * attributes associated with a name.  The name must exist in
 * the context.
 *
 * num_mods     length of "modify_attrs".
 *
 * modify_attrs
 *              a list of operations on attributes.  Each entry in
 *              the list includes a modify operation and the
 *              attribute identifier, syntax and value(s) for the
 *              operation. Operations are executed in the order in
 *              which they appear in the list.
 *
 *              When deleting an attribute, IDL requires that all
 *              the attribute's fields be filled with data that is
 *              valid for transmission.  This means that the
 *              attrib_syntax must be set to a NULL char string
 *              pointer and that the num_vals field must be set
 *              to zero.
 *
 * num_unexecuted_mods
 *              number of operations that failed.  0 if all
```

```
 *              operations succeeded or all operations failed.
 *
 * unexecuted_mods
 *              list of operations that failed.
 *              This array must be able to hold up to num_mods
 *              elements.  The array is empty if all operations
 *              succeeded or all operations failed.
 *
 * status       the status of the operation, returned by the server.
 *              It is set to success if all operations succeeded.
 *              If status is bad, it is the status associated with
 *              the first unexecuted operation.
 *
 *              Possible status codes: TBD
 */

void fn_dce_attr_multi_modify(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [in] boolean32                  follow_link,
    [in] unsigned32                 num_mods,
    [in, size_is(num_mods)]
        fn_dce_attr_modify_t        modify_attrs[],
    [out] unsigned32                *num_unexecuted_mods,
    [out, size_is(num_mods), length_is(*num_unexecuted_mods)]
        fn_dce_attr_modify_t        *unexecuted_mods[],
    [out] fn_dce_status_t           *status
);

}
```

A.1.6 fn_dce_ctx_locate.idl: Context Location Interface

```
[
uuid(e693d1de-f3f4-11cc-9357-08000932b6f8),
version(1.0),
pointer_default(ptr)
]

interface fn_dce_ctx_loc {

import "fn_dce_ctxb.idl";

/*
 * Context location interface (DCE RPC)
 *
 * The FN context location interface specifies operations to
 * locate servers for the root contexts of the global naming
 * systems, to locate surrogate client servers for various
 * naming systems, and to locate the update or authoritative
 * sites for a context.
 */
```

```
/*
 * F N _ D C E _ C T X _ L O C A T E _ R O O T S
 *
 * Locate servers for the global root contexts.
 *
 *  ns_id       Identifier for the global name service to be located.
 *              If "ns_id" is NULL, then locate any global name service.
 *
 *  max_num_refs
 *              maximum length of the output array "global_roots".
 *
 *  num_refs    count of references in "global_roots".
 *
 *  global_roots
 *              array of references.  Each reference identifies the
 *              servers for the root context of a global naming service.
 */

[broadcast]
void fn_dce_ctx_locate_roots
(
    [in] handle_t          h,
    [in] fn_dce_id_p_t     ns_id,
    [in] unsigned32        max_num_refs,
    [out] unsigned32       *num_refs,
    [out, size_is(max_num_refs), length_is(*num_refs)]
        fn_dce_ref_p_t     global_roots[]
);

/*
 * F N _ D C E _ C T X _ L O C A T E _ N S _ C L I E N T
 *
 * Locate a server that is a surrogate client for for a name service.
 *
 *  ns_id       Identifier for the name service whose surrogate client
 *              is to be located.  If "ns_id" is NULL, then locate a
 *              server that is a surrogate client for any name service.
 *
 *  max_num_refs
 *              maximum length of the output array "ns_clients".
 *
 *  num_refs    count of references in "ns_clients".
 *
 *  ns_clients
 *              array of references.  Each reference identifies a
 *              server that is a surrogate client for one or more
 *              naming services.
 */

[broadcast]
void fn_dce_ctx_locate_ns_client
(
```

```
    [in] handle_t          h,
    [in] fn_dce_id_p_t     ns_id,
    [in] unsigned32        max_num_refs,
    [out] unsigned32       *num_refs,
    [out, size_is(max_num_refs), length_is(*num_refs)]
        fn_dce_ref_p_t     ns_clients[]
);

/*
 * F N _ D C E _ C T X _ L O C _ L I S T _ N S _ C L I E N T S
 *
 * Get the identifiers of the name services that a server, which
 * is a surrogate client, supports.
 *
 *  h           a handle that identifies the server to which the
 *              operation should be directed.
 *
 *  max_num_ids
 *              maximum length of the output array "ns_ids".
 *
 *  num_ids     count of identifiers in "ns_ids".
 *
 *  ns_ids      array of name service identifiers.  An identifier
 *              of each naming service, for which this server is
 *              a surrogate client, is listed.
 */

void fn_dce_ctx_loc_list_ns_clients
(
    [in] handle_t            h,
    [in] unsigned32          max_num_ids,
    [out] unsigned32         *num_ids,
    [out, size_is(max_num_ids), length_is(*num_ids)]
        fn_dce_id_p_t        ns_ids[],
    [out] fn_dce_status_t    *status
);

/*
 * F N _ D C E _ C T X _ L O C A T E _ U P D A T E
 *
 * Get update server(s) for a context
 *
 *  h          a handle that identifies the server to which the
 *             operation is directed.
 *
 *  ctx        reference of context whose update servers are requested.
 *
 *  max_num_refs
 *             maximum length of the output array "update_ctx".
 *
 *  num_refs
 *             count of references in "update_ctx".
```

```
 *
 *  update_ctx
 *             array of references.  Each reference identifies update
 *             server(s) for the context.
 */

void fn_dce_ctx_locate_update
(
    [in] handle_t            h,
    [in] fn_dce_ref_p_t      ctx,
    [in] unsigned32          max_num_refs,
    [out] unsigned32         *num_refs,
    [out, size_is(max_num_refs), length_is(*num_refs)]
        fn_dce_ref_p_t       update_ctx[],
    [out] fn_dce_status_t    *status
);

/*
 * F N _ D C E _ C T X _ L O C A T E _ A U T H O R I T A T I V E
 *
 * Get authoritative server(s) for a context
 *
 *  h          a handle that identifies the server to which the
 *             operation is directed.
 *
 *  ctx        reference of context whose authoritative servers
 *             are requested.
 *
 *  max_num_refs
 *             maximum length of the output array "authoritative_ctx".
 *
 *  num_refs   count of references in "authoritative_ctx".
 *
 *  authoritative_ctx
 *             array of references.  Each reference identifies
 *             authoritative server(s) for the context.
 */

void fn_dce_ctx_locate_authoritative
(
    [in] handle_t            h,
    [in] fn_dce_ref_p_t      ctx,
    [in] unsigned32          max_num_refs,
    [out] unsigned32         *num_refs,
    [out, size_is(max_num_refs), length_is(*num_refs)]
        fn_dce_ref_p_t       authoritative_ctx[],
    [out] fn_dce_status_t    *status
);

}
```

A.1.7 fn_dce_srchb.idl: Data Types for Attribute Search Interface

```
[
pointer_default(ptr)
]

interface fn_dce_srchb {

import "fn_dce_ctxb.idl";
import "fn_dce_attrb.idl";

/*
 * Base Datatypes for Search Operations
 */

/*
 * Scope of the search
 *
 * srch_named_object    search named object
 * srch_one_context     search given context
 * srch_subtree         search given context and its sub-contexts
 * srch_local_subtree   search given context and its sub-contexts
 *                      within local server.
 */

typedef enum {
    srch_named_object,
    srch_one_context,
    srch_subtree,
    srch_local_subtree
} fn_dce_srch_scope_t;

/*
 * Controls the scope and other characteristics of the search.
 *
 * follow_links  if TRUE, follow XFN links encountered during the
 *               search.  if FALSE, do not follow XFN links.
 *
 *               If XFN links are followed, the name of the link
 *               and its reference are returned for further
 *               resolution.  The attributes associated with the
 *               name of the link are not included in the search.
 *
 *               If XFN links are not followed, the attributes
 *               associated with the name of an XFN link are
 *               searched and the name of the link is returned
 *               if its attribute values satisfy the search filter.
 *
 *               The effect of the follow_links control on native
 *               links is implementation specific.
 *
 * return_ref    if TRUE, return object references bound to names
 *               whose attribute values satisfy the filter expression.
```

```
 *               If FALSE, do not return references.
 */

typedef struct {
        fn_dce_srch_scope_t     scope;
        boolean32               follow_links;
        boolean32               return_ref;
} fn_dce_srch_control_t, *fn_dce_srch_control_p_t;


/*
 * Search Filter Expression
 *
 * The search filter is an expression that is evaluated against the
 * attributes of named objects bound in the scope of the search
 * operation "fn_dce_srch_ext()".
 *
 * For a complete description of the search filter see the man page
 * for FN_search_filter_t.
 *
 * The filter expression and arguments passed to the filter contructor
 * "fn_search_filter_create()" are the same as those passed to
 * "fn_dce_srch_ext()".
 */

/*
 * Search Filter Arguments
 */

/* argument syntaxes  */

typedef enum {
    fn_dce_srch_arg_stx_id,
    fn_dce_srch_arg_stx_attr,
    fn_dce_srch_arg_stx_val,
    fn_dce_srch_arg_stx_str
} fn_dce_srch_arg_syntax_t;


/*
 * argument
 */

typedef union switch ( fn_dce_srch_arg_syntax_t syntax_of_arg ) syntax {
    case fn_dce_srch_arg_stx_id:
        fn_dce_id_p_t           id;

    case fn_dce_srch_arg_stx_attr:
        fn_dce_attr_p_t         attr_id;

    case fn_dce_srch_arg_stx_val:
        fn_dce_attr_value_p_t   value;
```

```
        case fn_dce_srch_arg_stx_str:
            fn_dce_string_p_t       str;

    } fn_dce_srch_arg_t, *fn_dce_srch_arg_p_t;

    /*
     * Results of Search
     *
     * name           Name of object.
     *
     *                An implicit next naming system pointer is identified
     *                by a name with a trailing '/'.
     *
     * passed_filter  TRUE if attributes associated with "name" passed the
     *                filter; otherwise FALSE.
     *
     * continue_search
     *                TRUE if "name" is bound to a link that should be
     *                followed or a context that should be searched.
     *
     * ref            Object reference to which "name" is bound. NULL if
     *                the reference was not requested or is not needed
     *                to continue the search.
     *
     * cache_info     Caching hints for "name" and its "ref"
     */

    typedef struct {
            fn_dce_composite_name_p_t       name;
            boolean32                       passed_filter;
            boolean32                       continue_search;
            fn_dce_ref_p_t                  ref;
            fn_dce_cache_info_t             cache_info;
    } fn_dce_srch_name_t, *fn_dce_srch_name_p_t;

    }
```

A.1.8 fn_dce_srch.idl: Attribute Search Interface

```
    ************************ fn_dce_srch.idl ***************************

    [
    uuid(4e13f1ca-c4f3-11cd-80d8-080009352555),
    version(1.0),
    pointer_default(ptr)
    ]

    interface fn_dce_srch_ {

    import "fn_dce_ctxb.idl";
    import "fn_dce_attrb.idl";
    import "fn_dce_srchb.idl";
```

```
/*
 * Search interface (DCE RPC)
 *
 * The search interface provides operations to lookup names of objects
 * with specific attribute values.
 *
 * The operations in the search interface take the following two
 * initial arguments:
 *
 *
 *  h             a handle that identifies the server to which the
 *                operation should be directed.
 *
 *  progress      structure that guides the path resolution phase of
 *                an operation.
 *
 *  input:
 *
 *  ref           reference of starting context
 *
 *  unresolved_name
 *                name of target object or context for the search,
 *                relative to "ref"
 *
 *
 *  resolved_name
 *                NULL
 *
 *  link_ctx_ref  NULL
 *
 *  final output:
 *
 *  resolved_name
 *                target context for the search.
 *                For extended search, this is the starting context
 *                of the search.
 *
 *  ref           reference that is bound to "resolved_name".
 *
 *  unresolved_name
 *                NULL
 *
 *  cache_info    caching hints for the resolved name and its reference
 *
 *  link_ctx_ref  NULL
 *
 *      fn_dce_ctxb.idl describes the use of the progress structure.
 */

/*
 * Enumerate the names, bound in the target context, whose attribute
 * values match those specified in "attrs".  If "attrs" is NULL,
```

```
 * return all names bound in the context.
 *
 * For multi-valued attributes, order of attributes is ignored and
 * attributes values not specified in "attrs" are ignored.  If no
 * values are specified for an attribute, the presence of the
 * attribute is tested.
 *
 *  num_attrs     the count of attributes listed in "attrs".
 *                0 if "attrs" is NULL.
 *
 *  attrs         the attributes to compare with those of named
 *                objects bound in the target context.
 *                Each attribute in "attrs" contains an identifier,
 *                syntax, and 0 or more values.
 *                NULL if all names are to be returned.
 *
 *  return_ref    if TRUE, return object references bound to names
 *                whose attribute values match "attrs".
 *                If FALSE, do not return references.
 *
 *  iter_pos      a cursor into the list of names. It is maintained
 *                by the server.  Its state is initialized by the
 *                server at the first request (client passes in a
 8                pointer to NULL) and freed when the last name has
 *                been returned. Its state may also be freed by the
 *                operation fn_dce_ctx_free_iterator.
 *
 *  max_names     maximum length of the output array "names"
 *
 *  num_names     count of names listed in "names"
 *
 *  names         list of atomic names whose attribute values match
 *                those specified in "attrs". Optionally, the object
 *                reference to which the name is bound and its caching
 *                information are returned with each name.
 */

void fn_dce_srch(
    [in] handle_t                   h,
    [in, out] fn_dce_progress_t     *progress,
    [in] unsigned32                 num_attrs,
    [in, size_is(num_attrs)]
        fn_dce_attr_p_t             attrs[],
    [in] boolean32                  return_ref,
    [in, out] fn_dce_cursor_t       *iter_pos,
    [in] unsigned32                 max_names,
    [out] unsigned32                *num_names,
    [out, size_is(max_names), length_is(*num_names)]
        fn_dce_srch_name_p_t        names[],
    [out] fn_dce_status_t           *status
);
```

```
/*
 * Enumerate the names, bound in the scope of the search, whose
 * attribute values satisfy the specified filter expression.
 *
 *  control       controls the scope of the search, whether links
 *                are followed during the search, and whether
 *                object references bound to the names whose
 *                attribute values satisfy the filter expression
 *                are returned.
 *
 *  expression    expression that is evaluated against the attributes
 *                of named objects bound in the scope of the search.
 *
 *                For a complete description of the search filter
 *                expression and its arguments see the man page for
 *                FN_search_filter_t.
 *
 *  num_args      count of arguments on array "srch_args"
 *
 *  srch_args     arguments to the search filter expression
 *                "expression"
 *
 *  iter_pos      a cursor into the list of names. It is maintained
 *                by the server.  Its state is initialized by the
 *                server at the first request (client passes in a
 *                pointer to NULL) and freed when the last name has
 *                been returned. Its state may also be freed by the
 *                operation fn_dce_ctx_free_iterator.
 *
 *  max_names     maximum length of the output array "names"
 *
 *  num_names     count of names listed in "names"
 *
 *  names         list of names whose attribute values satisfy the
 *                filter expression.  Optionally, the object reference
 *                to which the name is bound and its caching
 *                information are returned with each name.
 *
 *                The names of links to follow and sub-contexts to
 *                search may also appear in the list.
 *
 *                Names are relative to the starting context for
 *                the search.
 */

void fn_dce_srch_ext(
    [in] handle_t                 h,
    [in, out] fn_dce_progress_t   *progress,
    [in] fn_dce_srch_control_p_t  control,
    [in, string] unsigned char    *expression,
    [in] unsigned32               num_args,
    [in, size_is(num_args)]
```

```
        fn_dce_srch_arg_p_t        srch_args[],
    [in, out] fn_dce_cursor_t      *iter_pos,
    [in] unsigned32                max_names,
    [out] unsigned32               *num_names,
    [out, size_is(max_names), length_is(*num_names)]
        fn_dce_srch_name_p_t       names[],
    [out] fn_dce_status_t          *status
);

}
```

A.2 ONC+ RPC Protocol for XFN

This section defines the RPC interface of the XFN service for ONC+ platforms. The interface is specified in ONC/RPCL and closely mirrors the XFN client context and attribute interfaces.

See Solaris NIPG for descriptions of RPC and RPCL in ONC+.

```
/*
 * The interface definition for the XFN service for ONC+/RPC.
 */

/* Basic data structures used in interface.
 * Mirror the data structures defined in the context and attribute
 * interfaces defined in the XFN specification.
 */
struct xFN_string {
    unsigned long        code_set;
    unsigned long        lang_terr;
    unsigned long        char_count;
    opaque               contents<>;
};

struct xFN_composite_name {
    xFN_string           compound<>;
};

struct xFN_ctx_handle {
    opaque               cookie[8];
};

struct xFN_identifier {
    unsigned int         format;
    opaque               contents<>;
};

struct xFN_attrvalue {
    opaque               value<>;
};

struct xFN_attribute {
    xFN_identifier       *attr_id;
    xFN_identifier       *attr_syntax;
    xFN_attrvalue        values<>;
};

struct xFN_attrmod {
    xFN_attribute        *attr;
    unsigned int         mod_op;
};

struct xFN_attrmodlist {
    xFN_attrmod          mods<>;
};
```

```
struct xFN_attrset {
    xFN_attribute        attrs<>;
};

struct xFN_namelist {
    opaque               cookie[8];
};

struct xFN_bindinglist {
    opaque               cookie[8];
};

struct xFN_valuelist {
    opaque               cookie[8];
};

struct xFN_multigetlist {
    opaque               cookie[8];
};

struct xFN_ref_addr {
    xFN_identifier       *type;
    opaque               data<>;
};

struct xFN_ref {
    xFN_identifier       *type;
    xFN_ref_addr         addrs<>;
};

struct xFN_status {
    unsigned int         code;
    xFN_ref              *resolved_ref;
    xFN_composite_name   *resolved_name;
    xFN_composite_name   *remaining_name;
    xFN_string           *diagnostic_msg;
    unsigned int         link_code;
    xFN_ref              *link_resolved_ref;
    xFN_composite_name   *link_resolved_name;
    xFN_composite_name   *link_remaining_name;
    xFN_string           *link_diagnostic_msg;
};

/*
 * The context interface argument and result types.
 */
struct fn_ctx_handle_from_initial_arg {
    unsigned int         authoritative;
    xFN_string           *user_info;
    xFN_string           *host_info;
};
struct fn_ctx_handle_from_initial_res {
```

```
    xFN_ctx_handle       ctx;
    xFN_status           *status;
};

struct fn_ctx_lookup_arg {
    xFN_ctx_handle       ctx;
    xFN_composite_name   *name;
};
struct fn_ctx_lookup_res {
    xFN_ref              *ref;
    xFN_status           *status;
};

/*
 * 'batch_size' is the maximum number of entries that the client
 * wants the server to return at one time.  The server can return
 * any number of entries at one time up to a limit of 'batch_size'.
 * If 'batch_size' is 0, the client does not place a limit on
 * how many entries are returned at a time.
 *
 * This definition of 'batch_size' applies to all the iteration
 * operations in this interface.
 */

struct fn_ctx_list_names_arg {
    xFN_ctx_handle       ctx;
    xFN_composite_name   *name;
    unsigned int         batch_size;
};
struct fn_ctx_list_names_res {
    xFN_namelist         namelist;
    xFN_string           names<>;
    xFN_status           *status;
};
struct fn_namelist_next_arg {
    xFN_namelist         namelist;
};
struct fn_namelist_next_res {
    xFN_namelist        namelist;
    xFN_string          names<>;
    xFN_status          *status;
};

struct fn_namelist_destroy_arg {
    xFN_namelist        namelist;
};

struct xFN_binding {
    xFN_string          *name;
    xFN_ref             *ref;
};
```

```
struct fn_ctx_list_bindings_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    unsigned int        batch_size;
};
struct fn_ctx_list_bindings_res {
    xFN_bindinglist     bindinglist;
    xFN_binding         bindings<>;
    xFN_status          *status;
};
struct fn_bindinglist_next_arg {
    xFN_bindinglist     bindinglist;
};
struct fn_bindinglist_next_res {
    xFN_bindinglist     bindinglist;
    xFN_binding         bindings<>;
    xFN_status          *status;
};

struct fn_bindinglist_destroy_arg {
    xFN_bindinglist     bindinglist;
};

struct fn_ctx_bind_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_ref             *ref;
    unsigned int        exclusive;
};
struct fn_ctx_bind_res {
    int                 rval;
    xFN_status          *status;
};
struct fn_ctx_unbind_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
};
struct fn_ctx_unbind_res {
    int                 rval;
    xFN_status          *status;
};
struct fn_ctx_create_subcontext_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
};
struct fn_ctx_create_subcontext_res {
    xFN_ref             *ref;
    xFN_status          *status;
};
struct fn_ctx_destroy_subcontext_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
```

```
};
struct fn_ctx_destroy_subcontext_res {
    int                 rval;
    xFN_status          *status;
};
struct fn_ctx_rename_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *oldname;
    xFN_composite_name  *newname;
    unsigned int        exclusive;
};
struct fn_ctx_rename_res {
    int                 rval;
    xFN_status          *status;
};
struct fn_ctx_lookup_link_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
};
struct fn_ctx_lookup_link_res {
    xFN_ref             *ref;
    xFN_status          *status;
};

struct fn_ctx_get_ref_arg {
    xFN_ctx_handle      ctx;
};
struct fn_ctx_get_ref_res {
    xFN_ref             *ref;
    xFN_status          *status;
};

struct fn_ctx_handle_from_ref_arg {
    xFN_ref             *ref;
    unsigned int        authoritative;
};
struct fn_ctx_handle_from_ref_res {
    xFN_ctx_handle      ctx;
    xFN_status          *status;
};

struct fn_ctx_equivalent_name_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_string          *leading_name;
};

struct fn_ctx_equivalent_name_res {
    xFN_composite_name  *name;
    xFN_status          *status;
};
```

```
/*
 * The attribute interface argument and result types.
 */
struct fn_attr_get_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_identifier      *attr_id;
    unsigned int        follow_link;
};
struct fn_attr_get_res {
    xFN_attribute       *attr;
    xFN_status          *status;
};
struct fn_attr_modify_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    unsigned int        mod_op;
    xFN_attribute       *attr;
    unsigned int        follow_link;
};
struct fn_attr_modify_res {
    int                 rval;
    xFN_status          *status;
};
struct fn_attr_get_values_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_identifier      *attr_id;
    unsigned int        follow_link;
    unsigned int        batch_size;
};
struct fn_attr_get_values_res {
    xFN_valuelist       valuelist;
    xFN_identifier      *attr_syntax;
    xFN_attrvalue       attrvals<>;
    xFN_status          *status;
};

struct fn_valuelist_next_arg {
    xFN_valuelist       valuelist;
};

struct fn_valuelist_next_res {
    xFN_valuelist       valuelist;
    xFN_identifier      *attr_syntax;
    xFN_attrvalue       attrvals<>;
    xFN_status          *status;
};

struct fn_valuelist_destroy_arg {
    xFN_valuelist       valuelist;
};
```

```
struct fn_attr_get_ids_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    unsigned int        follow_link;
};
struct fn_attr_get_ids_res {
    xFN_attrset         *attrset;
    xFN_status          *status;
};
struct fn_attr_multi_get_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_attrset         *attr_ids;
    unsigned int        follow_link;
    unsigned int        batch_size;
};
struct fn_attr_multi_get_res {
    xFN_multigetlist    multigetlist;
    xFN_attribute       attrs<>;
    xFN_status          *status;
};

struct fn_multigetlist_next_arg {
    xFN_multigetlist    multigetlist;
};
struct fn_multigetlist_next_res {
    xFN_multigetlist    multigetlist;
    xFN_attribute       attr<>;
    xFN_status          *status;
};
struct fn_multigetlist_destroy_arg {
    xFN_multigetlist    multigetlist;
};

struct fn_attr_multi_modify_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_attrmodlist     *mods;
    unsigned int        follow_link;
};
struct fn_attr_multi_modify_res {
    int                 rval;
    xFN_status          *status;
    xFN_attrmodlist     *unexecuted_mods;
};

/* Arguments for the extended attribute interface */

struct fn_attr_bind_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_ref             *ref;
```

```
    xFN_attrset         *attrs;
    unsigned int        exclusive;
};
struct fn_attr_bind_res {
    int                 rval;
    xFN_status          *status;
};
struct fn_attr_create_subcontext_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_attrset         *attrs;
};
struct fn_attr_create_subcontext_res {
    xFN_ref             *ref;
    xFN_status          *status;
};

struct xFN_searchlist {
    opaque              cookie[8];
};

struct xFN_ext_searchlist {
    opaque              cookie [8];
};

struct xFN_search_control {
    unsigned int        scope;
    unsigned int        follow_links;
    unsigned int        max_names;
    unsigned int        return_ref;
    xFN_attrset         *return_attrs;
};

enum xFN_search_filter_item_type {
    XFN_ATTR_TYPE = 1,
    XFN_ATTRVALUE_TYPE = 2,
    XFN_STRING_TYPE = 3,
    XFN_IDENTIFIER_TYPE = 4
};

union xFN_search_filter_item switch (xFN_search_filter_item_type ftype) {
case XFN_ATTR_TYPE:
    xFN_attribute       *attr;
case XFN_ATTRVALUE_TYPE:
    xFN_attrvalue       *attrvalue;
case XFN_STRING_TYPE:
    xFN_string          *str;
case XFN_IDENTIFIER_TYPE:
    xFN_identifier      *identifier;
default:
    void;
};
```

```
struct xFN_search_filter {
    xFN_string                *expression;
    xFN_search_filter_item    filter_item<>;
};

struct xFN_searchitem {
    xFN_string          *name;
    xFN_ref             *ref;
    xFN_attrset         *attrs;
};

struct fn_attr_search_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_attrset         *match_attrs;
    unsigned int        return_ref;
    xFN_attrset         *return_attr_ids;
    unsigned int        batch_size;
};

struct fn_attr_search_res {
    xFN_searchlist      searchlist;
    xFN_searchitem      items<>;
    xFN_status          *status;
};
struct fn_searchlist_next_arg {
    xFN_searchlist      searchlist;
};
struct fn_searchlist_next_res {
    xFN_searchlist      searchlist;
    xFN_searchitem      items<>;
    xFN_status          *status;
};
struct fn_searchlist_destroy_arg {
    xFN_searchlist      searchlist;
};

struct xFN_ext_searchitem {
    xFN_composite_name  *name;
    xFN_ref             *ref;
    xFN_attrset         *attrs;
};

struct fn_attr_ext_search_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
    xFN_search_control  *control;
    xFN_search_filter   *filter;
    unsigned int        batch_size;
};
struct fn_attr_ext_search_res {
    xFN_ext_searchlist  ext_searchlist;
```

```
    xFN_ext_searchitem  items<>;
    xFN_status          *status;
};
struct fn_ext_searchlist_next_arg {
    xFN_ext_searchlist  ext_searchlist;
};
struct fn_ext_searchlist_next_res {
    xFN_ext_searchlist  ext_searchlist;
    xFN_ext_searchitem  items<>;
    xFN_status          *status;
};
struct fn_ext_searchlist_destroy_arg {
    xFN_ext_searchlist  ext_searchlist;
};

struct fn_ctx_get_syntax_attrs_arg {
    xFN_ctx_handle      ctx;
    xFN_composite_name  *name;
};
struct fn_ctx_get_syntax_attrs_res {
    xFN_attrset         *attrset;
    xFN_status          *status;
};

/*
 * This is the basic context interface.
 */
program XFN_SERVICE_PROG {
version XFN_SERVICE_VERS {
    fn_ctx_handle_from_initial_res fn_ctx_handle_from_initial(
        fn_ctx_handle_from_initial_arg) = 100;
    fn_ctx_lookup_res fn_ctx_lookup(fn_ctx_lookup_arg) = 101;
    fn_ctx_list_names_res fn_ctx_list_names(
        fn_ctx_list_names_arg) = 102;
    fn_namelist_next_res fn_namelist_next(
        fn_namelist_next_arg) = 103;
    void fn_namelist_destroy(fn_namelist_destroy_arg) = 104;
    fn_ctx_list_bindings_res fn_ctx_list_bindings(
        fn_ctx_list_bindings_arg) = 105;
    fn_bindinglist_next_res fn_bindinglist_next(
        fn_bindinglist_next_arg) = 106;
    void fn_bindinglist_destroy(fn_bindinglist_destroy_arg) = 107;
    fn_ctx_bind_res fn_ctx_bind(fn_ctx_bind_arg) = 108;
    fn_ctx_unbind_res fn_ctx_unbind(fn_ctx_unbind_arg) = 109;
    fn_ctx_create_subcontext_res fn_ctx_create_subcontext(
        fn_ctx_create_subcontext_arg) = 110;
    fn_ctx_destroy_subcontext_res fn_ctx_destroy_subcontext(
        fn_ctx_destroy_subcontext_arg) = 111;
    fn_ctx_rename_res fn_ctx_rename(fn_ctx_rename_arg) = 112;
    fn_ctx_lookup_link_res fn_ctx_lookup_link(
        fn_ctx_lookup_link_arg) = 113;
    fn_ctx_get_ref_res fn_ctx_get_ref(fn_ctx_get_ref_arg) = 114;
```

```
fn_ctx_handle_from_ref_res fn_ctx_handle_from_ref(
    fn_ctx_handle_from_ref_arg) = 115;
void fn_ctx_handle_destroy(xFN_ctx_handle) = 116;
fn_ctx_equivalent_name_res fn_ctx_equivalent_name(
    fn_ctx_equivalent_name_arg) = 117;

/*
 * The basic attribute interface.
 */
fn_attr_get_res fn_attr_get(fn_attr_get_arg) = 200;
fn_attr_modify_res fn_attr_modify(fn_attr_modify_arg) = 201;

/*
 * Operations on multiple attribute values.
 */
fn_attr_get_values_res fn_attr_get_values(
    fn_attr_get_values_arg) = 202;
fn_valuelist_next_res fn_valuelist_next(
    fn_valuelist_next_arg) = 203;
void fn_valuelist_destroy(fn_valuelist_destroy_arg) = 204;
fn_attr_get_ids_res fn_attr_get_ids(fn_attr_get_ids_arg) = 205;
fn_attr_multi_get_res fn_attr_multi_get(
    fn_attr_multi_get_arg) = 206;
fn_multigetlist_next_res fn_multigetlist_next(
    fn_multigetlist_next_arg) = 207;
void fn_multigetlist_destroy(fn_multigetlist_destroy_arg) = 208;
fn_attr_multi_modify_res fn_attr_multi_modify(
    fn_attr_multi_modify_arg) = 209;

/*
 * The extended attribute interface.
 */
fn_attr_bind_res fn_attr_bind(fn_attr_bind_arg) = 210;
fn_attr_create_subcontext_res fn_attr_create_subcontext(
    fn_attr_create_subcontext_arg) = 211;
fn_attr_search_res fn_attr_search(fn_attr_search_arg) = 212;
fn_searchlist_next_res fn_searchlist_next(
    fn_searchlist_next_arg) = 213;
void fn_searchlist_destroy(fn_searchlist_destroy_arg) = 214;
fn_attr_ext_search_res fn_attr_ext_search(
    fn_attr_ext_search_arg) = 215;
fn_ext_searchlist_next_res fn_ext_searchlist_next(
    fn_ext_searchlist_next_arg) = 216;
void fn_ext_searchlist_destroy(
    fn_ext_searchlist_destroy_arg) = 217;
```

```
        /*
         * Operation for syntax attributes.
         */
        fn_ctx_get_syntax_attrs_res fn_ctx_get_syntax_attrs(
            fn_ctx_get_syntax_attrs_arg) = 300;
    } = 1;
    } = 100220;
```

Appendix B

Mapping XFN

This appendix contains descriptions of how the XFN interfaces can be implemented using some existing naming systems.

B.1 Mapping XFN to DNS

This section describes how DNS can be used to federate enterprise-level naming systems and how the XFN interfaces can be implemented using DNS operations.

B.1.1 Overview

DNS is the Internet Domain Name System. It is a naming system that names resources in a large global network — the Internet. It has been used primarily to name enterprises, such as universities, institutions and companies, and entities within enterprises such as machines. The DNS namespace is hierarchically structured, in which atomic names are ordered right-to-left and are delimited by dot characters ('.'). Names representing non-leaf nodes identify *domains*. Leaf nodes typically identify machines (or *hosts*).

In DNS, *resource records* are associated with names. Each resource record has a type that indicates the type of information stored.

See Internet RFC 1034 and Internet RFC 1035 for a description of DNS.

B.1.2 Representation of XFN Concepts in DNS

XFN policies specify that DNS is one of the global naming systems that can be reached from the Initial Context using the atomic name "...".

B.1.2.1 Name Syntax

The syntax of DNS names is described in Internet RFC 1035. It maps into the XFN standard syntax model (see Section 3.8 on page 50) using the following syntax-related attribute values:

Attribute Identifier	Attribute Value
fn_syntax_type	standard
fn_std_syntax_direction	right-to-left
fn_std_syntax_separator	.
fn_std_syntax_escape	\
fn_std_syntax_case_insensitive	
fn_std_syntax_begin_quote1	"
fn_std_syntax_end_quote1	"
fn_std_syntax_code_sets	0x00010020 (ASCII)

B.1.2.2 XFN References

The XFN context for DNS uses the following reference types: ''inet_domain'' and ''inet_host''. DNS domains and hosts, respectively. These references contain a list of IP addresses. An IP address has type ''inet_ipaddr_string'' and is represented as an ASCII string of Internet addresses (for example, ''128.199.235.10'').

Description	Identifier Format	Identifier Value
Internet domain reference type	FN_ID_STRING	inet_domain
Internet host reference type	FN_ID_STRING	inet_host
Internet IP address type	FN_ID_STRING	inet_ipaddr_string

B.1.3 Federating DNS With Other Naming Systems

When DNS is federated, composite name resolution is supported using strong separation and implicit next naming system pointers, as described in Section 4.3.1.1 on page 63.

B.1.3.1 Next Naming System Reference

DNS provides resource records of type TXT. The reference of the implicit next naming system pointer of a DNS domain name is derived from information stored in TXT records of that domain. For example, given a DNS domain name **Wiz.COM**, the information required for constructing the corresponding next naming system reference (**FN_ref_t**) is obtained using the TXT records associated with the DNS entry **Wiz.COM**.

Each DNS domain that is to be federated has TXT records of the following format:

TXT	XFNREF	*rformat*	*reftype*
TXT	*addrtag*	*addrinfo*	

A TXT record with the tag **XFNREF** specifies the reference type of the reference. *rformat* specifies the format of the reference type identifier. It can be one of **STRING**, **UUID**, or **OID**. *reftype* is the contents of the reference type. If this TXT record is not present, the default reference type is "XFN_SERVICE" (FN_ID_STRING format).

TXT records with the **XFN** prefix in their *addrtag* field identify addresses within a reference. *addrtag* specifies how the information in *addrinfo* is to be interpreted to construct the contents of an **FN_ref_t** object. Each TXT record so tagged may generate one or more addresses — this is determined by the implementation for each *addrtag*. There can be multiple TXT records with the same *addrtag*; how such multiple records are related is, again, determined by the implementation for each *addrtag*. If a single address spans multiple TXT records, or if the order of records with the same *addrtag* is significant, sequencing must be done amongst TXT records with the same *addrtag*. It is up to the implementation identified by *addrtag* to specify how such sequencing is supported.

B.1.3.2 Examples of Reference Data

Assume that the following TXT records are associated with the domain **Wiz.COM**:

```
TXT   XFNONC    doggone.Wiz.COM 100220 1 3
TXT   XFNONC    pagin.Wiz.COM 100220 1 3
TXT   XFNLONG   2 0 longaddressbegin
TXT   XFNLONG   2 1 longaddressend
```

These records represent three addresses, two with the *addrtag* **XFNONC** and one with the *addrtag* **XFNLONG**. The address information for **XFNONC** consists of a host name, a program number, and a version number range. The address information for **XFNLONG** spans two TXT records. The address format is that of a sequence size and sequence number, followed by the address contents.

The resulting reference for this set of records contains:

```
XFN_SERVICE (reference type)
    SUNW_xfn_onc (address type)
    doggone.Wiz.COM 100220 1 3 (address contents)

    SUNW_xfn_onc (address type)
    pagin.Wiz.COM 100220 1 3 (address contents)

    VENDORX_xfn_xprot (address type)
    someveryverylongaddress (address contents)
```

In another example, assume that the following TXT records are associated with the domain **Biz.COM**:

```
TXT   XFNREF   OID 1.3.22.1.6.1.3
TXT   XFNDCE   (1 fd33328c4-2a4b-11ca-af85-09002b1c89bb .....)
TXT   XFNCDS   (1 fd33328c4-2a4b-11ca-af85-09002b1c89bb .....)
```

The first record represents the reference type. The TXT record with the *addrtag* **XFNDCE** represents an XFN DCE service address. The TXT record with the *addrtag* **XFNCDS** represents a XFN / CDS context service address. The resulting reference for this set of records contains:

```
1.3.22.1.6.1.3  (reference type)
    1.3.22.1.6.2.1 (address type)
    <encoded address> (address contents)

    1.3.22.1.6.2.1 (address type)
    <encoded address> (address contents)
```

B.1.3.3 Registry of addrtag

addrtag can be general purpose or platform-specific. For example, a general purpose *addrtag* might define a way of encoding address information using *uuencode* and ISO BER. Such a scheme would allow arbitrary addresses to be encoded. XFNONC and XFNDCE are examples of platform-specific *addrtag*s.

A registry will be maintained by X / Open for *addrtag* and their formats.

B.1.3.4 Recommendations for the DNS Context Implementation

A DNS context implementation should be flexible in supporting various *addrtag* and reference type combinations. It should use the *addrtag* and reference type to select code that does the translation of *addrinfo* into data for the **FN_ref_t** object. Such a mechanism should be extensible and not hardwired for any specific *addrtag*.

Note that such a mechanism is private to the DNS context implementation and is unrelated to the context implementation for the specified reference and address types except for the fact that they both know the specific address formats expected in the reference data structure, **FN_ref_t**.

B.1.3.5 Resolving Through DNS

The primary operation on the implicit next naming system pointer of a DNS domain is expected to be lookup. Resolution through DNS proceeds exactly as described in Section 4.3.1.1 on page 63:

1. The DNS name component is resolved in DNS.

2. The query to DNS returns TXT records associated with the supplied DNS name.
3. A reference for the implicit next naming system pointer is constructed based on information obtained from the appropriate TXT records associated with the DNS name.
4. *fn_ctx_handle_from_ref*( ) is used to return a handle to the context in the naming system subordinate to DNS.
5. Resolution of the remaining components of the composite name proceeds from this context.

For example, resolution of the name:

```
.../Wiz.COM/_orgunit/finance
```

involves resolving **...** in the Initial Context to get a handle to the global context, from which the DNS name **Wiz.COM** is resolved in DNS to obtain a reference (using the TXT records associated with **Wiz.COM**) to a context in which the rest of the name **_orgunit/finance** can be resolved.

B.1.4 XFN API Function Mapping

B.1.4.1 XFN Operations on DNS names

The lookup operation is supported for host and domain names by sending queries to DNS.

DNS supports the ability to use a query to list all entries in a particular DNS domain. The list operations are supported using this type of queries.

No update operations are supported because DNS does not support updates at the protocol level. All updates must be effected through changes to the data file on DNS servers.

The get syntax attributes operation is supported by generating the attributes algorithmically upon demand. All other attribute operations are not supported.

B.1.4.2 XFN Operations on Implicit Next Naming System Pointer

Updates operations on the implicit next naming system pointer are not supported through the XFN interfaces because the DNS protocol does not support updates. The status [FN_E_OPERATION_NOT_SUPPORTED] is returned for *fn_ctx_bind*( ), *fn_ctx_unbind*( ), *fn_ctx_rename*( ), *fn_ctx_create_subcontext*( ) and *fn_ctx_destroy_subcontext*( ).

The effects of the *fn_ctx_bind*( ), *fn_ctx_unbind*( ) and *fn_ctx_rename*( ) operations on the implicit next naming system pointer can be achieved by editing the data file of the domain. A simple administrative command can be provided to give a more user-friendly means of entering reference information.

List operations are supported by invoking the function *fn_ctx_handle_from_ref*( ) on the implicit next naming system pointer's reference and performing the corresponding list operations on the resulting context.

Only the get syntax attributes operation is supported. This is done by looking up the syntax attributes of the context bound to the implicit next naming system pointer.

B.2 Mapping XFN to X.500: Preliminary Specification

The whole of this section (**Mapping XFN to X.500**) is assigned X/Open **Preliminary Specification** status (not **CAE Specification** status). For explanation of the difference between **Preliminary** and **CAE** specifications, see the description under **X/Open Technical Publications** in the **Preface**. Readers should appreciate that the *header* title on each page of this section of Section B.2 correctly identifies its status as **Preliminary**, while the *footer* on each of its odd-numbered pages also correctly identifies this section of the Appendix as a part of the **XFN CAE Specification**.

This section describes how XFN concepts can be supported in the X.500 directory service. It defines X.500 object classes and attributes and describes how X.500 can be used to federate enterprise-level naming systems.

B.2.1 X.500 Overview

The X.500 directory service is a global directory service. Its components cooperate to manage information about objects in a worldwide scope. Such objects include countries, organizations, people and machines. It provides the capability to lookup information by name (a *white-pages* service), and to browse and search for information (a *yellow-pages* service).

The information is held in a directory information base (DIB). Entries in the DIB are arranged in a tree structure. Each entry is a named object and comprises a set of attributes. Each attribute has a defined attribute type and one or more values. The directory schema defines the mandatory and optional attributes for each class of object.

The X.500 namespace is hierarchical. An entry is unambiguously identified by a distinguished name. A distinguished name is the concatenation of selected attributes from each entry in the tree along a path leading from the root down to the named entry.

Users of the X.500 directory may, subject to access controls, interrogate and modify the entries and attributes in the DIB.

For more information on X.500, refer to CCITT X.500 (1988/1993)/ISO Directory and X/Open DCE Directory.

B.2.2 Representation of XFN Concepts in X.500

XFN policies specify that X.500 is one of the global naming systems that can be reached from the Initial Context using the atomic name ‘‘`...`’’.

B.2.2.1 Name Syntax

The distinguished name (DN) syntax described in X/Open DCE Directory is used to name X.500 entries when they appear as the global component of an XFN composite name. From the Initial Context, the atomic name ‘‘`...`’’ prefixes such global components. For example,

```
.../c=us/o=wiz/ou=sales/cn=smith
```

is a string representation of a distinguished name which identifies a person named `Smith` in the `Sales` unit of the `Wiz` organization in the USA. (Note that countries are identified by the two-letter codes defined in ISO Country Codes.)

In this string representation name parts are separated by the ‘‘`/`’’ (slash) character and attribute type/value pairs are separated by the ‘‘`=`’’ (equals) character. Abbreviations are defined for some commonly used attribute types (for example, ‘‘`c`’’ represents Country Name, ‘‘`o`’’ represents Organization Name).

The distinguished name syntax of X/Open DCE Directory adheres to the XFN standard syntax model (see Section 3.8 on page 50). The syntax-related attribute values are as follows:

Attribute Identifier	Attribute Value
fn_syntax_type	standard
fn_std_syntax_direction	left_to_right
fn_std_syntax_separator	/
fn_std_syntax_escape	\
fn_std_syntax_begin_quote1	‘
fn_std_syntax_begin_quote2	"
fn_std_syntax_end_quote1	’
fn_std_syntax_end_quote2	"
fn_std_syntax_ava_separator	,
fn_std_syntax_typeval_separator	=
fn_std_syntax_code_sets	0x00010001 (ISO 8859-1/Latin-1)

B.2.2.2 XFN References

The X.500 directory holds information about named objects. A new X.500 object class is introduced to provide XFN support for such objects. Each X.500 object can have an XFN object reference and may contain additional XFN information. The new object class is defined in ASN.1 as follows:

```
xFN OBJECT-CLASS ::= {
    SUBCLASS OF         { top }
    KIND                auxiliary
    MAY CONTAIN         { objectReferenceId |
                          objectReferenceAddresses |
                          nNSReferenceId |
                          nNSReferenceAddresses }
    ID                  id-oc-xFN
}

id-oc-xFN OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-oc-xFN(24)
}
```

It is defined as an auxiliary object class so that it may be inherited by all X.500 object classes. This permits XFN support to be present at any X.500 object.

The xFN object class contains four optional attributes. The attributes nNSReferenceId and nNSReferenceAddresses are described in Section B.2.3.2 on page 236. The attributes objectReferenceId and objectReferenceAddresses are defined in ASN.1 below.

```
objectReferenceId ATTRIBUTE ::= {
    WITH SYNTAX                 XFNId
    EQUALITY MATCHING RULE      XFNIdMatch
    SINGLE VALUE                TRUE
    ID                          id-at-objectReferenceId
}

XFNId ::= CHOICE {
    stringId                    [0] IA5String,
    uUId                        [1] PrintableString,
    objectId                    [2] OBJECT IDENTIFIER
}

XFNIdMatch MATCHING-RULE ::= {
    SYNTAX                      XFNId
    ID                          id-mr-xFNIdMatch
}

id-at-objectReferenceId OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-at-objectReferenceId(26)
}

id-mr-xFNIdMatch OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-mr-xFNId(32)
}
```

The attribute objectReferenceId stores the reference identifier for a given object. It may be a string (for example, onc_fn_user), a universal unique identifier in string format (for example, fd3328c4-2a4b-11ca-af85-09002b1c89bb), or an object identifier.

The attribute `objectReferenceAddresses` is defined in ASN.1 as follows:

```
objectReferenceAddresses ATTRIBUTE ::= {
    WITH SYNTAX                 XFNReference
    EQUALITY MATCHING RULE      XFNReferenceMatch
    ID                          id-at-objectReferenceAddresses
}

XFNReference ::= SEQUENCE {
    addressId                   XFNId,
    addressValue                OCTET STRING
}

XFNReferenceMatch MATCHING-RULE ::= {
    SYNTAX                      XFNReference
    ID                          id-mr-xFNReferenceMatch
}

id-at-objectReferenceAddresses OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-at-objectReferenceAddresses(27)
}

id-mr-xFNReferenceMatch OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-mr-xFNReference(33)
}
```

The attribute `objectReferenceAddresses` stores a list of addresses for a given object. The `addressId` component contains an address identifier. It may be a string (for example, `onc_fn_nisplus`), a universal unique identifier (UUID) in string format (for example, `fd3328c4-2a4b-11ca-af85-09002b1c89cc`), or an object identifier. The `addressValue` component encodes an address in an `OCTET STRING`.

If the `addressId` is a string or a UUID string then the actual address is stored in the `addressValue` component. However, if the `addressId` is an object identifier then it identifies the attribute at the X.500 entry which actually stores the address; the `addressValue` component is then unused.

B.2.2.3 String Encoding for XFN References

As an interim measure, the following X.500 object class is also defined. It accommodates those X.500 implementations that have schemas which cannot easily support new attributes with compound ASN.1 syntaxes. The object class `xFNSupplement` is defined in ASN.1 as follows:

```
xFNSupplement OBJECT-CLASS ::= {
    SUBCLASS OF         { top }
    KIND                auxiliary
    MAY CONTAIN         { objectReferenceString |
                          nNSReferenceString }
    ID                  id-oc-xFNSupplement
}

id-oc-xFNSupplement OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-oc-xFNSupplement(25)
}
```

It contains two optional attributes. Both attributes store a string encoding of an XFN reference. The `nNSReferenceString` attribute is described further in Section B.2.3.2 on page 236. The attribute `objectReferenceString` is defined in ASN.1 as follows:

```
objectReferenceString ATTRIBUTE ::= {
    WITH SYNTAX                 OCTET STRING
    EQUALITY MATCHING RULE      octetStringMatch
    SINGLE VALUE                TRUE
    ID                          id-at-objectReferenceString
}

id-at-objectReferenceString OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-at-objectReferenceString(30)
}
```

Its `OCTET STRING` syntax is further constrained to conform to the following BNF definition:

```
<ref>          ::= <id> '$' <ref-addr-set>
<ref-addr-set> ::= <ref-addr> | <ref-addr> '$' <ref-addr-set>
<ref-addr>     ::= <id> '$' <addr>
<addr>         ::= <hex-string>

<id>           ::= 'id'   '$' <string> |
                   'uuid' '$' <uuid-string> |
                   'oid'  '$' <oid-string>

<string>       ::= <char> | <char> <string>
<char>         ::= <PCS> | '' <PCS>
<PCS>          ::= // Portable Character Set:
                   // !"#$%&'()*+,-./0123456789:;<=>?
                   // @ABCDEFGHIJKLMNOPQRSTUVWXYZ[]^_
                   // `abcdefghijklmnopqrstuvwxyz{|}~

<uuid-string>  ::= <uuid-char>  | <uuid-char> <uuid-string>
<uuid-char>    ::= <hex-digit> | '-'

<oid-string>   ::= <oid-char>  | <oid-char> <oid-string>
<oid-char>     ::= <digit> | '.'

<hex-string>   ::= <hex-octet> | <hex-octet> <hex-string>
<hex-octet>    ::= <hex-digit> <hex-digit>
<hex-digit>    ::= <digit> |
                   'a' | 'b' | 'c' | 'd' | 'e' | 'f' |
                   'A' | 'B' | 'C' | 'D' | 'E' | 'F'
<digit>        ::= '0' | '1' | '2' | '3' | '4' | '5' |
                   '6' | '7' | '8' | '9'
```

The two attributes defined in the `xFNSupplement` object class enable XFN references to be conveniently stored as strings in X.500. The following are examples of string form XFN references:

```
"id$x500$oid$2.5.4.29$00"

"id$onc_fn_user$id$onc_fn_nisplus$aabbccddeeff"

"uuid$f5fd3328c4-2a4b-11ca-af85-09002b1c89bb$uuid$f5fd3328c4-
 2a4b-11ca-af85-09002b1c89cc$aabbccddeeff"
```

In the first example the reference identifier is `x500`. This indicates that it is a reference to an X.500 object. The address identifier is the object identifier for the OSI presentation address attribute. The address value is stored at that specified attribute.

In the second example the reference identifier is `onc_fn_user`. This indicates that it is a reference to a user in an enterprise. The address identifier is `onc_fn_nisplus`. The address value is a hexadecimal string.

The third example has a reference identifier and an address identifier in UUID format. The address value is a hexadecimal string.

Appendix G provides a registry of reference and address identifiers.

B.2.3 Federating X.500 with Other Naming Systems

X.500 federates other naming systems by supplying the necessary support to permit other namespaces to appear to be seamlessly attached below the X.500 namespace.

When X.500 is used to federate other naming systems it supports composite name resolution using weak separation and implicit next naming system pointers. These concepts are described in Section 4.3 on page 63.

B.2.3.1 Weak Separation

An X.500 context uses a syntactic policy to determine the boundary between the global naming system and the subordinate naming system. According to this policy, components from a composite name that form a distinguished name have the distinguishing feature that each component contains the ‘‘=’’ (equals) character. The first name component that does not have an unescaped and unquoted ‘‘=’’ character is considered to be in the subordinate naming system.

This imposes the minor restriction that a subordinate naming system must not permit the ‘‘=’’ character to appear unescaped or unquoted in its top level names.

For example, given the composite name

```
.../c=us/o=wiz/_orgunit/finance
```

the syntactic policy specifies that only the two components `c=us` and `o=wiz` should be extracted to form a distinguished name for resolution in X.500.

B.2.3.2 Implicit Next Naming System Pointers

An implicit next naming system pointer is an XFN reference to a subordinate naming system. Such an XFN reference is stored in the X.500 entry at the boundary between X.500 and the subordinate naming system.

The `xFN` object class defined above has support for next naming system references. The attribute `nNSReferenceId` stores a reference identifier and has the same syntax as `objectReferenceId`. The attribute `nNSReferenceAddresses` stores a list of addresses and has the same syntax as `objectReferenceAddresses`.

They are defined in ASN.1 as follows:

```
nNSReferenceId ATTRIBUTE ::= {
    WITH SYNTAX                 XFNId
    EQUALITY MATCHING RULE      XFNIdMatch
    SINGLE VALUE                TRUE
    ID                          id-at-nNSReferenceId
}

nNSReferenceAddresses ATTRIBUTE ::= {
    WITH SYNTAX                 XFNReference
    EQUALITY MATCHING RULE      XFNReferenceMatch
    SINGLE VALUE                TRUE
    ID                          id-at-nNSReferenceAddresses
}

id-at-nNSReferenceId OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-at-nNSReferenceId(28)
}

id-at-nNSReferenceAddresses OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-at-nNSReferenceAddresses(29)
}
```

The `xFNSupplement` object class also defines a string form for next naming system references. The attribute `nNSReferenceString` has the same syntax as `objectReferenceString`. It is defined in ASN.1 as follows:

```
nNSReferenceString ATTRIBUTE ::= {
    WITH SYNTAX                 OCTET STRING
    EQUALITY MATCHING RULE      octetStringMatch
    SINGLE VALUE                TRUE
    ID                          id-at-nNSReferenceString
}

id-at-nNSReferenceString OBJECT IDENTIFIER ::= {
    iso(1) member-body(2) ansi(840) sun(113536)
    ds-at-nNSReferenceString(31)
}
```

The next naming system reference attribute enables an XFN reference to a subordinate naming system to be stored in X.500. The following example is an XFN reference which may be used to attach an NIS+ namespace below an entry in the X.500 namespace. The example uses the string form.

```
"id$onc_fn_enterprise$id$onc_fn_nisplus_root$0000000f77697a2e
 636f6d2e2062696762696700"
```

It points to the root of an NIS+ domain in an enterprise. Its reference identifier is `onc_fn_enterprise`. Its address identifier is `onc_fn_nisplus_root` and it contains a single address value. The address value is an XDR encoded string, comprising the domain name ‘‘`wiz.com.`’’ followed by the hostname `bigbig`.

A next naming system reference may be added and removed from an X.500 entry by naming the entry with a trailing ''/'' character when using the functions *fn_ctx_bind*( ) and *fn_ctx_unbind*( ) respectively. For example, the next naming system reference at the X.500 entry `.../c=us/o=wiz` is identified by the name,

```
.../c=us/o=wiz/
```

Note however that in order to add or remove such references the X.500 entry must already exist.

B.2.3.3 Resolving Through X.500

Once a next naming system pointer is in place at an X.500 entry then name resolution through X.500 proceeds as described in Section 4.3.1.2 on page 63:

1. The X.500 name components are first extracted from the composite name using the syntactic policy described above.
2. A distinguished name is constructed using these components and a read operation is performed at the X.500 directory. The next naming system reference attribute is requested.
3. An **FN_ref_t** object is constructed from the next naming system reference attribute(s).
4. *fn_ctx_handle_from_ref*( ) is called with that reference to return a handle to the root context object in the naming system subordinate to X.500.
5. Resolution of the remaining components in the composite name now proceeds from this new context.

For example, resolution of the name:

```
.../c=us/o=Wiz/_orgunit/finance
```

involves resolving the atomic name ''`...`'' in the Initial Context to get a handle to the global context. From there the distinguished name `/c=us/o=Wiz` is resolved in X.500 to obtain a next naming system reference. That reference identifies a context in which the rest of the name `_orgunit/finance` can be resolved.

B.2.4 XFN API Function Mapping

The X.500 directory access protocol offers a rich set of query and manipulation operations which makes it suitable for supporting the context and attribute operations of the XFN API. The one-to-one correspondence between XFN and X.500 operations ensures atomicity.

B.2.4.1 Context Operations

All XFN context operations (except as noted below) can be fully supported by X.500.

The *fn_ctx_bind*( ) and *fn_ctx_create_subcontext*( ) functions cannot be used to introduce new names in the X.500 namespace. This is because the creation of a new entry in X.500 requires that all mandatory attributes (for example, object class) be supplied. Instead, the functions *fn_attr_bind*( ) and *fn_attr_create_subcontext*( ) may be used to create a new entry and set its attributes in a single atomic operation.

If *fn_ctx_lookup*( ) is performed on an X.500 object then its XFN reference is returned. The default reference identifier is `x500`; it denotes a reference to *any* X.500 object. If a more specific reference identifier is available at the X.500 object then that is used instead. The object reference will contain at least one reference address. That address identifier will be `x500` and its address value will be the string form of the X.500 object's distinguished name. If more addresses are present at the X.500 object then they too are included in the XFN reference.

B.2.4.2 Attribute Operations

All XFN attribute operations can be fully supported by X.500.

The functions *fn_attr_bind*( ) and *fn_attr_create_subcontext*( ) may be used to create a new entry and set its attributes in a single atomic operation.

B.3 Mapping XFN to CDS: Preliminary Specification

The whole of this section (**Mapping XFN to CDS**) is assigned X/Open **Preliminary Specification** status (not **CAE Specification** status). For explanation of the difference between **Preliminary** and **CAE** specifications, see the description under **X/Open Technical Publications** in the **Preface**. Readers should appreciate that the *header* title on each page of this section of Section B.3 correctly identifies its status as **Preliminary**, while the *footer* on each of its odd-numbered pages also correctly identifies this section of the Appendix as a part of the **XFN CAE Specification**.

This section describes the mapping of XFN to CDS.

B.3.1 Overview

DCE Cell Directory Service (CDS) is a naming system in the enterprise namespace. It is a generic directory service and provides a replicated information system. Its primary use is as an information service for server locations and bindings.

CDS controls the DCE cell namespaces that consist of names denoting directory, object (terminal nodes), and soft link entries. Any entry controlled by CDS consists of a set of operational attributes and optional application specific attributes. Attributes are identified by attribute identifiers. Attributes have associated values with a defined syntax (data type). Attributes are either single-valued or set-valued.

The cell namespace is a hierarchical structure of untyped names. The syntax for naming entries is a left-to-right ordered set of atomic names, corresponding to the top-to-bottom lineage of the named entry. Atomic names are separated by a slash ('/'), same as the XFN component separator.

For more information on CDS, see reference X/Open DCE Directory.

Policies and rules defined in this appendix section must be applied to client context implementations that provide for integration of CDS into the naming federation. Portability of applications requires conformance to the specified API mappings and the appropriate reference, address, and attribute types. Furthermore, the conformance requirements also apply to a CDS server implementation that directly exports the XFN DCE IDL, as specified in Section A.1.

Note that the CDS programming interface is internal to CDS implementations and not a published API. The mappings specified here define the behaviour of the CDS service as specified in X/Open DCE Directory. CDS context implementations may use the internal CDSPI but its behaviour is neither guaranteed nor specified here.

B.3.2 Representation of XFN Concepts in CDS

The following subsections specify how the XFN concepts and policies are represented in CDS.

B.3.2.1 Context Operations

Context operations are fully supported, with the exception of *fn_ctx_rename*( ) (unless a CDS directory version greater than 3.0 - equivalent to DCE 1.1 and up - is supported).

B.3.2.2 XFN Links

XFN links are controlled by the attribute **XFN_Link**, associated with an CDS object entry. The reference and address types of a XFN link are **FN_LINK_REF** and **FN_LINK_ADDR**. XFN links are not integrated with CDS soft links.

B.3.2.3 Attribute Operations

The full set of attribute operations is supported.

The operations on multiple attributes are supported but it shall be noted that this may be a limited support due to the lack of appropriate protocol elements in the native CDS IDL interface.

fn_attr_get_ids( )
No restrictions apply.

fn_attr_multi_get( )
The CDS protocol has a read-ahead option (flag *maybemore* in operation *cds_ReadAttribute*( )) that implementations may use. However, in cases where CDS servers do not support the *maybemore* option and where the requested information exceeds the buffer size, multiple calls to the server have to be performed. Therefore, no atomicity guarantees for the returned set can be made. Also, the CDS specification does not guarantee that CDS servers return this "read-ahead" buffer if the request on the initially specified attribute fails (for example, attribute does not exist).

fn_attr_multi_modify ( )
This function may be implemented as multiple calls of the remote procedure *cds_ModifyAttribute*( ) that performs a single modification operation.

If the *name* supplied to an attribute operation is non-empty, the operation performs on the attributes of the named CDS entry (CDS child pointer, object entry, or soft link entry). If the *name* argument is empty, the attribute operation performs on the entity that is referenced by the resolved *ctx*. This can be a CDS directory entry or an object external to CDS.

Unless further specified in the following sections, the behaviour of attribute operations conforms to the specification X/Open DCE Directory. For instance, size limitations, and some operational attributes in CDS are determined as read-only or write-once.

B.3.2.4 Attribute Identifiers and Syntax

The representation of attribute identifiers is FN_ID_ISO_OID_BER. Operational XFN attributes registered by OSF contain the following object identifier prefix:

```
1.3.22.1.6
{iso(1) identified-org(3) osf(22) dce(1) xfn(6) ... }
```

CDS implementations store the syntax of attribute values typically in an attributes table local to the CDS clerk.

B.3.2.5 Attribute Values

Any XFN operation that creates an attribute, instantiates the attribute as multi-valued. The creation of attributes with no values (empty set) is permitted.

XFN attribute *fn_attr_get*( ) operations read both single-valued and set-valued attributes. No distinction can be made through the XFN interfaces.

The behaviour of the XFN attribute *fn_attr_modify*( ) operation is further specified in *fn_attr_modify*( ) on page 129.

B.3.2.6 Syntax Attributes

The CDS context implementation supports the XFN standard syntax model. These attributes are maintained by the context implementations (not stored as attributes in CDS) and contain fixed values as follows:

Attribute Identifier	Attribute Value
fn_syntax_type	standard
fn_std_syntax_direction	left_to_right
fn_std_syntax_separator	/
fn_std_syntax_escape	\
fn_std_syntax_case_insensitive	*absent*
fn_std_syntax_begin_quote1	‘
fn_std_syntax_end_quote1	’
fn_std_syntax_begin_quote2	"
fn_std_syntax_end_quote2	"
fn_std_syntax_ava_separator	*absent*
fn_std_syntax_typeval_separator	*absent*
fn_std_syntax_code_sets	0x00010001 (ISO 8859-1/Latin-1)
fn_std_syntax_locale_info	Its interpretation depends on the values of fn_std_syntax_code_sets.

B.3.2.7 Atomicity of Operations

There is no guarantee of atomicity for XFN interface functions that translate into multiple CDS operations. Target entities may change during the course of executing the XFN function due to concurrent access to CDS servers. Data may have become partially invalid or errors may be detected (such as [FN_E_NAME_NOT_FOUND]) that indicate changed state.

Particularly, the atomicity of the following XFN API functions is not guaranteed, unless the application provides other means for synchronization:

- *fn_ctx_list_bindings*( )
- *fn_attr_multi_get*( )
- *fn_attr_multi_modify*( ).

B.3.2.8 DCE Group References

A DCE group is a reference that contains a set of addresses (type **FN_DCE_GROUP_MEMBER_ADDR**) whose value is a composite name, called group member.

The DCE group describes a group of equivalent servers. The group members are names that may point to further groups and finally resolve into references that contain communications endpoints of servers. DCE groups can be used as an aid for replication.

The values of the **RPC_Group** attribute are used to construct a XFN reference with a set of addresses of type **FN_DCE_GROUP_MEMBER_ADDR**.

These addresses are resolved in an implementation specific order (preferably random) until one valid communications endpoint (for example, reference address of type **FN_DCE_RPC_SERVER_ADDR**) has been found for further resolution of the (residual) name. This may be a recursive operation due to nested groups, therefore, may involve multiple calls to CDS for resolving the compound name of a group member entry. The context implementation maintains the state of this execution loop to permit an exhaustive resolution of a DCE group

(which is possibly nested) by avoiding duplications.

To take advantage of the additional reliability provided by group references, the CDS context will repeatedly select different members of this group to retry an XFN operation if the operation fails. An algorithm for retry of failed operations is given below. It is not necessary for an implementation to follow this algorithm, however, the externally visible results appearing at the XFN API must be indistinguishable from the results produced by this algorithm.

The algorithms assume that calls made on behalf of a user call to the XFN API are all part of the same thread of execution and that this execution is independent of any other user calls. For example, if concurrent calls are made by users to the XFN API, these calls do not interfere with each other. A separate instantiation of the CDS context is associated with each user call.

In order to detect possible cycles in nested group references, this model assumes there exists state global to the thread of execution.

Let FN_<op> represent a call to the XFN API invoking the function denoted by <op>. For each FN_<op>, there exists a procedure CDS_CTX_<op> which will be invoked when the parameters associated with the FN_<op> call indicate use of the CDS context for the operation.

Each invocation of CDS_CTX_<op> on the CDS context operates as follows.

1. Extract the name components to be resolved to a context from the parameters passed on invocation. Note that for some operations, such as listing the names in a context, the entire name must be resolved. Other operations, such as bind, must resolve all but the terminal atomic name. Resolve as much of the relevant name as possible in CDS and retrieve the attributes associated with the resolved name. If this fails, then return an error; otherwise go to step 2.
2. If attributes retrieved from CDS map to an object reference, then go to step 5; otherwise, go to step 3.
3. If there are any unresolved residual name components, then return an error, along with the unresolved residual name components, because the name could not be resolved; otherwise, go to step 4.
4. Since there are no residual name components, the target context for the operation is within CDS. Perform <op> on the object specified by the call parameters and return the results.
5. If the object reference formed from the attributes is a group reference, then go to step 6; otherwise, return the object reference and any unresolved residual name along with other relevant parameters.
6. If the group is empty, then return an error because the reference to the context could not be found; otherwise, go to step 7.
7. Select a member (a name) from the group and remove this member from the group. Check for a cycle. A cycle occurs if the selected member of the group was previously resolved as part of this thread of execution. If a cycle is detected, then go to step 6; otherwise, go to step 8.
8. Compose this member name with the unresolved residual name, if any, such that the residual name components trail the member name components. Based upon the input parameters and using the initial context, generate call parameters and invoke FN_<op>. If FN_<op> returned successfully, then return the results; otherwise, go to step 6.

Notes:

1. The CDS context implementation expects the **XFN_Reference** attribute to be present to identify a XFN reference. However, in order to preserve backward

compatibility with RPC server registration of group entries that does not create this attribute, the CDS context implementation must also identify a DCE group even in the absence of the **XFN_Reference** attribute.

2. The system administration must take precautions that a group resolves into addresses that logically (it may be represented by different replicas or multiple communication protocols) bind the same context.
3. CDS does not preclude the existence of both **CDS_Towers** and **RPC_Group** attributes in a single object entry. If an object entry contains both attributes (and the **XFN_Reference** attribute contains pointers to both), the **CDS_Towers** attribute takes precedence for constructing the address; that is, **RPC_Group** is ignored for resolution.
4. Although DCE groups provide a mechanism for organizing RPC servers into groups, the target of the names listed in groups do not need to be RPC server references. Refer to context implementation specifications for details.

B.3.3 Federating CDS With Other Naming Systems

The following subsections specify how other naming systems can be federated using CDS and what the composition rules are.

B.3.3.1 Weak and Strong Separation

The CDS context implementation supports weak separation. The CDS atomic name separator is the same as the XFN component separator and the CDS compound name does not need to be quoted or escaped.

The weak separation model is implemented on a resolution basis. The unresolved residual of a name is returned if a name cannot be further resolved within CDS. The only syntactical restriction is that the first name component of the unresolved name must be a syntactically valid CDS name.

The strong separation model is implicitly supported by the context implementation. Both models behave exactly the same on the interface level.

B.3.3.2 Junctions (Explicit Next Naming System Pointers)

Subordinate naming systems are federated with CDS using junctions. Junctions are CDS object entries that are identified by the presence of the attribute **XFN_Reference** that identifies a XFN reference to a context of a subordinate naming system. The reference types controlled by junctions is determined by the bound context.

Note that CDS object entries may control references (presence of **XFN_Reference**) to objects external to CDS that are not contexts of subordinate next naming systems. For instance, referenced RPC servers might not support naming and do not export any XFN interfaces. The same mechanisms are used for identifying and managing these types of references.

B.3.3.3 Implicit Next Naming System Pointers

The implementation of *implicit next naming system pointers* in CDS is not supported.

B.3.4 Registered Values and their Encodings

This section defines the encodings for XFN references and addresses, and for XFN specific CDS attributes. The mapping from CDS attributes to XFN references is defined where appropriate.

B.3.4.1 Reference Types

The following reference types are registered for DCE CDS XFN implementations. They are represented as ISO object identifiers (FN_ID_ISO_OID_STRING).

FN_DCE_CDS_REF — 1.3.22.1.6.1.1
This is the reference type denoting a CDS entity. The valid address type for this reference type is **FN_DCE_RPC_SERVER_ADDR**. Each address defines a set of communication endpoints (protocol towers) for a clearinghouse replica. Multiple replicas are referenced in multiple addresses.

For reference type **FN_DCE_CDS_REF**, the fields of the address (type **FN_DCE_RPC_SERVER_ADDR**) are interpreted as follows:

tower_set
A set of protocol towers for the clearinghouse named in *address_option*.

object_uuids
This field contains one UUID that is the identifier of the clearinghouse that is named in *address_option* and whose communication endpoints are defined in *tower_set*.

address_option
This field contains information necessary for communicating with a clearinghouse and identifying the named entry in a clearinghouse and is structured as follows:

```
typedef struct {
    unsigned small       ch_type;
    cds_FullName_t       ch_name;
    cds_FullName_t       entry_name;
};
```

ch_type
The type of the clearinghouse that is either a **RT_master** or **RT_readonly** replica.

ch_name
The name of the clearinghouse that holds the replica of the requested entry.

entry_name
The CDS compound name of the requested entry.

FN_DCE_RPC_SERVER_REF — 1.3.22.1.6.1.2
This reference type is used to generate a reference that consists of a **FN_DCE_RPC_SERVER_ADDR** address constructed with the absence of a **XFN_Reference** attribute in a DCE RPC server entry in CDS (see the definition of **FN_DCE_RPC_SERVER_ADDR** for details).

FN_DCE_XFN_SERVER_REF — 1.3.22.1.6.1.3
The reference type that identifies a DCE RPC server exporting the XFN protocol (see Section A.1 on page 178).

FN_DCE_GROUP_REF — 1.3.22.1.6.1.4
This reference type is used to generate a reference that consists of a **FN_DCE_GROUP_MEMBER_ADDR** address constructed with the absence of a **XFN_Reference** attribute in a DCE group entry in CDS (see Section B.3.2.8 on page 242 for details).

FN_DCE_DFS_REF — 1.3.22.1.6.1.5
This reference type denotes a CDS junction to DCE Distributed File Service. Its usage is not further specified here.

FN_DCE_SEC_REF — 1.3.22.1.6.1.6
This reference type denotes a CDS junction to DCE Security Service. Its usage is not further specified here.

B.3.4.2 Address Types

Following address types represented as ISO object identifiers (FN_ID_ISO_OID_STRING) are registered for DCE CDS:

FN_DCE_RPC_SERVER_ADDR — 1.3.22.1.6.2.1
Its value is encoded as:

```
typedef struct {
    protocol_tower_set_t     *tower_set;
    object_uuids_t           *object_uuids;
    fn_dce_byte_str_t        *address_option;
};

typedef struct {
    unsigned16                num_towers;
    [length_is(num_towers)] protocol_tower_t *towers;
} protocol_tower_set_t;

typedef struct {
    unsigned16                num_objects;
    [length_is(num_objects)] uuid_t     *objects;
} object_uuids_t;
```

For the encodings of the **uuid_t** and **protocol_tower_t** data types refer to X/Open DCE RPC.

The DCE RPC server address describes an instance of a single RPC server and its protocol and addressing information that is used for building the XFN reference with addresses of type **FN_DCE_RPC_SERVER_ADDR**. If present, the values of attribute **RPC_ObjectUUIDs** are also used for constructing the XFN reference.

If the **XFN_Reference** attribute is present (see also note below), the address is constructed based on the contents of the *attribute_id* array. This *attribute_id* array must always contain **CDS_Towers** as first and **RPC_ObjectUUIDs** as second element. A third attribute identified by this array is optionally permitted. The *address_id* field is set to **FN_DCE_RPC_SERVER_ADDR**. The reference type (*reference_id* field) is determined by the **XFN_Reference** attribute.

Note: The CDS context implementation expects the **XFN_Reference** attribute to be present to identify a XFN reference. However, in order to preserve backward compatibility with RPC server registration of server entries that does not create this attribute, the CDS context implementation must also identify a RPC server address even in the absence of the **XFN_Reference** attribute. In this case, the type

of the constructed reference is **FN_DCE_RPC_SERVER_REF**, and **CDS_Towers** and **RPC_ObjectUUIDs** are the only attributes used to construct the address.

FN_DCE_GROUP_MEMBER_ADDR — 1.3.22.1.6.2.2
Its value is encoded as:

```
typedef struct {
    unsigned16                      member_size;
    [length_is(member_size)] fn_dce_byte_str_t     member;
};
```

Note: In order to preserve backward compatibility to RPC NSI function calls, RPC group entries shall only contain composite names with the global initial context (in DCE terms, fully qualified global names).

B.3.4.3 CDS Attributes

Following XFN related attribute identifiers represented as BER encoded object identifiers (FN_ID_ISO_OID_BER) are registered for DCE CDS. For further information refer to **X/Open DCE Directory**:

XFN_Reference — 1.3.22.1.6.3.1
Its value's syntax is *VT_byte*, encoded as:

```
typedef struct {
    fn_dce_id_t        reference_id;
    fn_dce_id_t        address_id;
    unsigned32         num_attrs;
    [length_is(num_attrs)]    fn_dce_id_t    attribute_id[];
};
```

The attribute can be multi-valued, each value defining the attribute information for different address types. The *reference_id* field must be the same for every value, otherwise an [FN_E_MALFORMED_REFERENCE] error will be generated.

The *attribute_id* array defines the attribute or possibly set of attributes that contain the address value information.

If the *attribute_id* array defines an **XFN_Addresses** attribute, a single address (type identifier and value) is constructed for each attribute value, according to the **XFN_Addresses** structure defined below. If the *attribute_id* array defines **XFN_Addresses**, the *address_id* field in the **XFN_Reference** attribute is not significant and the array must not have multiple values.

For any other attribute type defined in the *attribute_id* array, an address is constructed by taking the *address_id* from the **XFN_Reference** attribute and copying the value of the specified attribute into the address value. If the attribute contains multiple values, these are concatenated in an unspecified order to a single address value. If multiple attribute identifiers are defined in the *attribute_id* array, the address value is constructed by concatenating the values of these attributes in the listed order.

XFN_Addresses — 1.3.22.1.6.3.2
Its value's syntax is *VT_byte*, encoded as:

```
typedef struct {
    fn_dce_id_t        address_id;
    unsigned32         addr_size;
    [length_is(addr_size)]    fn_dce_addr_p_t    address_value;
```

```
};
```

XFN_Link — 1.3.22.1.6.3.3
Its value's syntax is *VT_byte*, encoded as **fn_dce_composite_name_t**.

CDS_Towers — 1.3.22.1.3.30
Its value's syntax is *VT_byte*. See also X/Open DCE RPC.

RPC_ObjectUUIDs — 1.3.22.1.1.2
Its value's syntax is *VT_byte*. See also X/Open DCE RPC.

RPC_Group — 1.3.22.1.1.3
Its value's syntax is *VT_byte*. See also X/Open DCE RPC.

RPC_CodeSets — 1.3.22.1.1.5
Its value's syntax is *VT_char*, encoded as:

```
typedef struct {
    unsigned32         version; /* version of this structure   */
    long               size;    /* number of code sets defined */
    [size_is(size)] long     *code_sets;
};
```

This code set attribute is encoded by the IDL Encoding Services before it is exported to the namespace. Refer to DCE RFC 40.1 for more details on code sets.

B.3.5 XFN API Function Mapping

B.3.5.1 Base Context Interface

fn_ctx_lookup( )
The *fn_ctx_lookup*( ) function resolves a name in CDS and returns the reference bound to the name.

XFN links are followed by resolution. For DCE groups, one of the group entries is resolved to its completion according to the algorithm described in Section B.3.2.8 on page 242.

fn_ctx_list_names( )
If the *fn_ctx_list_names*( ) operation resolves to a context controlled by CDS (directory entry), the listed names represent CDS directory, soft link, or object entries.

If *fn_ctx_list_names*( ) resolves to a CDS object entry, the operation can only proceed to its completion if the name can be identified as a junction (appropriate address and reference information); otherwise *fn_ctx_list_names*( ) returns with status [FN_E_NOT_A_CONTEXT].

If the reference type identifies a junction to a federated naming system, the names bound to that referenced context are listed and returned in the name set.

fn_ctx_list_bindings( )
If the *fn_ctx_list_bindings*( ) operation resolves to a context controlled by CDS (directory entry), the returned list represents names and bindings of CDS directory, soft link, or object entries.

If *fn_ctx_list_bindings*( ) resolves to a CDS object entry, the operation can only proceed to its completion if the name can be identified as a junction (similar to the behaviour of *fn_ctx_list_names*( )).

Due to the mapping to multiple remote operations in the native CDS protocol, it is not guaranteed that the results reflect a consistent snapshot of the CDS directory (if updates have occurred within the window of the call).

fn_ctx_bind( )

The
function binds a reference to the supplied *name* in the specified context. The terminal atomic name in the *name* argument always determines an CDS object entry. Depending on the reference type supplied, the following actions are taken:

FN_NULL_REF

A regular CDS object entry is created. This function creates the initial operational attributes, specified for CDS object entries. The **CDS_ObjectUUID** attribute cannot be written by performing this function. Other attributes may be added by performing subsequent attribute manipulation operations.

If the CDS object entry already exists, the call fails, even if the exclusive flag is zero.

FN_LINK_REF

If the reference contains a **FN_LINK_ADDR** address, a CDS object entry is created including the creation of the **XFN_Reference** and **XFN_Link** attributes. The **XFN_Link** attribute is a multi-valued type with a single value, containing the link-text.

If the CDS object entry already exists, the exclusive flag is zero, and the entry has no **XFN_Reference** attribute, then the **XFN_Reference** and **XFN_Link** attributes are created.

If the CDS object entry already exists, the exclusive flag is zero, and the entry is already a XFN link, then the **XFN_Link** attribute is updated.

If the CDS object entry exists and the exclusive flag is non-zero, the call fails.

Any other reference type

This is treated as a request for binding an external object that is named by the CDS object entry. The CDS object entry is created or updated with the presence of an **XFN_Reference** attribute.

If the CDS object entry does not exist, an object entry is created including the creation of the associate attributes (**XFN_Reference** and those related to the supplied address types in the reference). Implementations must provide appropriate support for addresses of type **FN_DCE_RPC_SERVER_ADDR** and **FN_DCE_GROUP_MEMBER_ADDR**. For any other address type contained in the reference, the operation creates the default attributes **XFN_Reference** and **XFN_Addresses**.

If the CDS object entry already exists, the exclusive flag is zero, and the entry is not a XFN link, then all attributes used to represent a XFN reference are replaced.

If the CDS object entry exists and the exclusive flag is non-zero, the call fails.

fn_ctx_unbind( )

The *fn_ctx_unbind*( ) function deletes CDS object or soft link entries.

fn_ctx_create_subcontext( )

The *fn_ctx_create_subcontext*( ) function creates a new CDS directory entry with its associated operational attributes in the same clearinghouse as its parent directory. The appropriate child pointer in the parent directory is created implicitly by the CDS server.

fn_ctx_destroy_subcontext( )

The *fn_ctx_destroy_subcontext*( ) function deletes an existing CDS directory entry (and implicitly, the child pointer in the parent directory). If a directory entry is not empty (contains child pointers), the call fails.

fn_ctx_rename( )
The *fn_ctx_rename*( ) function is only supported for CDS directory versions greater than V3.0 (DCE Release 1.1 and up). The *newname* must reside in the same context (parent directory) as *oldname.*

fn_ctx_lookup_link( )
The *fn_ctx_lookup_link*( ) function returns the reference bound to *name* if *name* is a XFN link entry (CDS object entry, containing an **XFN_Link** attribute).

fn_ctx_handle_from_ref( )
The *fn_ctx_handle_from_ref*( ) function constructs a context handle from the list of addresses in the given reference. The type of this handle differs depending on whether the context implementation maps XFN to the native CDS interfaces or the request is directed to a CDS server that exports the XFN DCE IDL remote interface.

If the reference denotes a DCE group (address type **FN_DCE_GROUP_MEMBER_ADDR**), the implementation of *fn_ctx_handle_from_ref*( ) resolves the DCE group to its completion according to the algorithm specified in Section B.3.2.8 on page 242.

fn_ctx_get_ref( )
The *fn_ctx_get_ref*( ) function returns a reference containing the address that was taken by *fn_ctx_handle_from_ref*( ) for constructing the context handle.

The reference may contain a list of addresses that is different to the one originally supplied to *fn_ctx_handle_from_ref*( ) if an **FN_DCE_GROUP_MEMBER_ADDR** address was processed. It then contains the resolved reference addresses.

fn_ctx_get_syntax_attrs( )
The *fn_ctx_get_syntax_attrs*( ) function retrieves the syntax attributes that are fixed and known to the CDS context implementation. These syntax attributes are not actually stored as attributes in CDS servers.

B.3.5.2 Base Attribute Interface

fn_attr_get( )
The *fn_attr_get*( ) function reads the attribute values of the specified attribute in the target entry (can be directory, child pointer, object, or soft link).

fn_attr_modify( )
The *fn_attr_modify*( ) function updates the specified attribute in the target entry. Only attributes in entries of type directory or object can be modified. If the entry is of type directory, only certain attributes may be modified (see X/Open DCE Directory for details).

If the operation code is FN_ATTR_OP_ADD and the attribute does not exist, it creates a set-valued attribute with the supplied values (the set of values may be empty). If the attribute already exists as set-valued, CDS performs a delete and subsequent create attribute operation with no guarantee of atomicity (The operation may fail if another create operation takes place concurrently). If the attribute exists as single-valued, the operation replaces the value (in this case, the supplied value set must contain exactly one member).

If the operation code is FN_ATTR_OP_ADD_EXCLUSIVE and the attribute does not exist, it creates a set-valued attribute with the supplied values (the set of values may be empty).

If the operation code is FN_ATTR_OP_REMOVE, it removes single-valued as well as set-valued attributes from the entry.

If the operation code is FN_ATTR_OP_ADD_VALUES, it adds the supplied values to a set-valued attribute. If the attribute is a single-valued type, the operation fails with

[FN_E_TOO_MANY_ATTR_VALUES].

If the operation code is FN_ATTR_OP_REMOVE_VALUES, it removes the supplied values from a set-valued attribute. This operation may result in an empty set-valued attribute. If the specified attribute is single-valued and the supplied value set contains exactly one member, a match will result in the deletion of the attribute. If the supplied values set contains more than one member, the call fails on single-valued attributes.

fn_attr_get_ids( )
The *fn_attr_get_ids*( ) function enumerates the attributes of the specified entry (whose type can be any of the supported CDS entry types).

fn_attr_get_values( )
The *fn_attr_get_values*( ) function behaves the same as *fn_attr_get*( ).

fn_attr_multi_get( )
The *fn_attr_multi_get*( ) function reads the set of specified attributes.

If the operation performs over the native CDS protocol, this operation is not atomic.

fn_attr_multi_modify( )
The *fn_attr_multi_modify*( ) function has a behaviour similar to *fn_attr_modify*( ). If the operation performs over the native CDS protocol, neither is the operation atomic nor can performance gains be expected (separate remote operation for each modification in the specified sequence).

B.3.6 Support Level of CDS Service

The XFN context implementation for CDS supports the following basic service primitives of CDS, as specified in X/Open DCE Directory. Possible restrictions and specific behaviour is specified elsewhere in this appendix section:

- *CreateDirectory*( )
- *CreateObject*( )
- *DeleteDirectory*( )
- *DeleteObject*( )
- *DeleteSoftLink*( )
- *EnumerateAttribute*( )
- *EnumerateChildren*( )
- *EnumerateObject*( )
- *EnumerateSoftLinks*( )
- *ModifyAttribute*( )
- *ReadAttribute*( )
- *ResolveName*( ).

The following CDS service primitives are not supported by the XFN context implementation for CDS:

- *Advertise*( )
- *CreateSoftLink*( )

- *CreateChild*( )
- *DeleteChild*( )
- *Solicit*( )
- *SolicitServer*( )
- *TestAttribute*( ).

Note, that a number of flags that control the CDS clerk to server communication such as the confidence level, cache timeout, soft link resolution, and *maybemore* cannot be manipulated through the XFN API. CDS context implementations support default settings (for example, confidence level to LOW) and other appropriate values (for example, *maybemore* set TRUE for *fn_attr_multi_get*( )). Refer to implementation specifications for identifying the behaviour.

B.4 Mapping XFN to NIS+

This section describes the mapping of XFN to NIS+. It describes how the XFN interface and XFN enterprise-level policies are be implemented and represented in NIS+.

B.4.1 Overview

NIS+ is the network information service in Solaris. It is an information retrieval system for well-known UNIX databases, such as the password tables, host tables and mail aliases maps. It also supports Solaris specific databases such as the automount maps and the credential tables. NIS+ is an enterprise-wide information service. The enterprise is partitioned into organizational units that are arranged into a tree and assigned hierarchical *domain names*. For example, an enterprise with the domain name, **wiz.com.**, may have the following organizational units:

```
sales.wiz.com.
engineering.wiz.com.
marketing.wiz.com.
corp.wiz.com.
```

Any of these organizational units might have suborganizations, which have further suborganizations, and so on.

The type of objects that NIS+ understands are principals, directories, tables, entries and groups. There is no concept of a user or host context, *per se*. Information about an object, such as a user, appears in various different tables, such as the credentials table, the password table, the automount maps and the mail aliases map. This information is retrieved using NIS+ *index names*. For example, the password entry for a user **mjones** is obtained by using the name:

```
[name=mjones]passwd.org_dir.sales.wiz.com.
```

The credentials for the same user are obtained using the name:

```
[name=mjones.sales.wiz.com.]cred.org_dir.sales.wiz.com.
```

See Administering NIS+ for an overview of NIS+ concepts and Solaris NIPG for a description of the NIS+ programming interface.

B.4.2 Representation of XFN Concepts in NIS+

Because NIS+ is an enterprise-wide service, it makes sense for the mapping of XFN over NIS+ to adopt the XFN enterprise-level policies (see Appendix D) as well as supporting the XFN interface. This means the XFN / NIS+ service should provide an Initial Context that conforms to the XFN Initial Context (see Section 5.3 on page 73 and Section D.4 on page 285), and naming contexts for organizations, users and hosts. Consequently, an XFN service implemented on top of NIS+ provides additional naming functionality over NIS+:

- XFN / NIS+ associates contexts to organizations, users and hosts and allows one to name objects and services relative to these entities.
- XFN / NIS+ can support the resolution of composite names across enterprise boundaries.
- XFN / NIS+ provides an XFN Initial Context.

Composite name resolution is supported using strong separation and implicit next naming system pointers, as described in Section 4.3.1.1 on page 63.

B.4.2.1 Mapping XFN Enterprise-level Policies to NIS+

XFN organizations are analogous to NIS+ domains. A NIS+ domain comprises logical collections of users and machines and information about them, arranged to reflect some form of hierarchical organizational structure within an enterprise.

NIS+ Domains and XFN Organizations

The top XFN organizational namespace is mapped to the NIS+ root domain. It is accessed using the name **_hostorg** or **_userorg** from the Initial Context (depending on whether the enterprise of interest is that of the host or user, respectively). Non-root XFN organizational units are mapped to NIS+ non-root domains.

In NIS+, users and hosts have a notion of a *home domain*. It is the primary NIS+ domain that maintains information associated with them. A user's or host's home domain can be determined directly using its NIS+ *principal name*. An NIS+ principal name is composed of the atomic user (login) name or the atomic host name with the name of the NIS+ home domain. For example, user **jsmith** in the home domain **wiz.com.** has an NIS+ principal name of **jsmith.wiz.com.**.

A user's NIS+ home domain corresponds to the user's XFN organizational unit, **_userorgunit**. Similarly, a host's home domain corresponds to the host's XFN organizational unit, **_hostorgunit**. These correspondences are used to determine the bindings for **_user**, **_host**.

NIS+ Users and XFN Users

An XFN user corresponds to an NIS+ user. Users in NIS+ are found in the **passwd.org_dir** table of a domain. Users in a particular XFN organizational unit correspond to the users in the **passwd.org_dir** table of the corresponding NIS+ domain. The **passwd.org_dir** table is used to obtain the list of user names for the context of **_user**.

NIS+ Hosts and XFN Hosts

An XFN host corresponds to an NIS+ host. Hosts in NIS+ are found in the **hosts.org_dir** table of a domain. Hosts in a particular XFN organizational unit correspond to the hosts in the **hosts.org_dir** table of the corresponding NIS+ domain. The **hosts.org_dir** table is used to obtain the list of host names for the context of **_host**.

XFN Sites and Services

There are no corresponding NIS+ concepts for the XFN site and service contexts. These are represented using additional NIS+ tables.

B.4.2.2 Name Syntax

The XFN / NIS+ contexts have the following syntax attributes:

Context Type	Attribute Identifier	Attribute Value
Organizational Unit or Site	fn_syntax_type fn_std_syntax_direction fn_std_syntax_separator fn_std_syntax_case_insensitive fn_std_syntax_escape fn_std_syntax_begin_quote1 fn_std_syntax_end_quote1 fn_std_syntax_code_sets	standard right_to_left . \ " " 0x00010020 (ASCII)
Host	fn_syntax_type fn_std_syntax_direction fn_std_syntax_case_insensitive fn_std_syntax_escape fn_std_syntax_begin_quote1 fn_std_syntax_end_quote1 fn_std_syntax_code_sets	standard flat \ " " 0x00010020 (ASCII)
User	fn_syntax_type fn_std_syntax_direction fn_std_syntax_escape fn_std_syntax_begin_quote1 fn_std_syntax_end_quote1 fn_std_syntax_code_sets	standard flat \ " " 0x00010020 (ASCII)
Service	fn_syntax_type fn_std_syntax_direction fn_std_syntax_separator fn_std_syntax_case_insensitive fn_std_syntax_escape fn_std_syntax_begin_quote1 fn_std_syntax_end_quote1 fn_std_syntax_code_sets	standard left_to_right / \ " " 0x00010020 (ASCII)

B.4.2.3 Context Representations

The implementation of XFN contexts in NIS+ uses NIS+ directories and tables. Some of these are existing NIS+ objects; others are newly created to represent data that do not exist in NIS+.

Organization objects map directly to NIS+ directories.

The hostname and username contexts for an organization use existing NIS+ tables as their source of host names and user names, respectively, as well as new tables to hold context data for users and hosts.

Other types of context objects such as user, host, service, site require new representations for their contexts. The storage for these types of context objects are implemented in a similar way: name to reference bindings of a context are stored using a bindings table. A bindings table contains atomic name to reference mappings. Each binding has a flag to prevent orphan contexts. In addition, a bindings table has one or more columns for storing attributes. Implementations for specific context types might differ in how they parse names and how they construct the reference or attributes to return. If the context allows next naming system pointer to be bound, the special name **_FNS_nns_** is used to bind the reference of the nns pointer. Some contexts may share a single physical bindings table (the bindings within the table are logically separated by context); for example, all contexts and subcontexts owned by a user may share the same bindings table.

To keep NIS+ objects that are used for the context implementation separate from other NIS+ objects, NIS+ objects used for context objects of an organization are kept under the **ctx_dir** subdirectory, at the same level as **org_dir**.

B.4.3 XFN References

B.4.3.1 Reference Types

The following reference type identifiers are defined for the types of XFN / NIS+ contexts. Their format is FN_ID_STRING.

Object Type	Reference Type Identifier
enterprise context	onc_fn_enterprise
file system context	onc_fn_fs
organization	onc_fn_organization
printername context	onc_fn_printername
printer context	onc_printers
site context	onc_fn_site
user name context	onc_fn_username
host name context	onc_fn_hostname
user object	onc_fn_user
host object	onc_fn_host
service context	onc_fn_service
namespace id context	onc_fn_nsid
null context	onc_fn_null
generic context	onc_fn_generic

B.4.3.2 Address Formats and Types

An XFN / NIS+ **FN_ref_t** object consists of one address; the address has type "onc_fn_nisplus" (FN_ID_STRING format). There are different subtypes of addresses, reflecting the different types of XFN / NIS+ contexts. The different addresses are differentiated for the following reasons:

- the names might have different syntax (for example, service names are slash separated left-to-right, organization names are dot separated right-to-left)
- operations (such as lookup and bind) might have different semantics.

An address of type "onc_fn_nisplus" has the following format:

context type	*repr type*	*version*	*unused*	*internal name or reference*

context type (byte) is an unsigned number indicating the type of context. *repr type* (byte) is an unsigned number with information regarding the context's representation. *version* (byte) is an unsigned number indicating the version number of the address. The last field is a variable-sized byte string that contains either an internal name of the NIS+ object that represents the context, or a serialised reference if this reference is that of a null context. An internal name is represented as an XDR encoded string. A serialised reference is an XDR encoded reference.

The *fn_ctx_handle_from_ref*( ) operation supported by XFN / NIS+ examines the context and representation type information in the address and returns a context of the appropriate subclass.

In addition, the following address type is used for Internet addresses associated with hosts: "inet_ipaddr_string" (see Section B.1).

B.4.4 XFN API Function Mapping

B.4.4.1 Context Operations

Lookup operations involve checking for the existence of corresponding NIS+ objects or looking up the binding in the appropriate bindings table.

List operations involve listing the appropriate NIS+ table or directory.

The create and destroy operations involve creation/destruction of the corresponding directory, bindings table or the set of bindings associated with the context.

The bind, unbind and rename operations involve updating the corresponding binding in the appropriate bindings table.

B.4.4.2 Attribute Operations

Syntax attributes requested through *fn_ctx_get_syntax_attrs*() are generated algorithmically on demand.

There are different levels of support for the attribute operations depending on the type of object. This is implementation-dependent.

B.4.4.3 Context Creation

Because supplemental information, such as the type of context to create and information about its representation, is required for creation of some contexts, XFN/NIS+ provides an extended context interface with the following additions:

```
FN_ref_t *fn_nis_ctx_create_context(
    FN_nis_ctx_t *ctx,
    const    FN_composite_name_t *name,
    unsigned int context_type,
    unsigned int repr_type,
    FN_status_t *status);

FN_ref_t *fn_nis_ctx_create_context_with_attrset(
    FN_nis_ctx_t *ctx,
    const    FN_composite_name_t *name,
    unsigned int context_type,
    unsigned int repr_type,
    const FN_attrset_t *attrs,
    FN_status_t *status);
```

Appendix C

Guidelines for Federating a Naming System

These guidelines are intended for naming system implementors who either want to integrate an existing naming system into the federation or develop a new XFN-conformant naming system (including any application that exports naming interfaces).

These guidelines are to be used in conjunction with the native specifications of the naming system to be federated and the implementation specification of the XFN client library.

The first section of this appendix describes some typical implementation models that an implementor can use to provide XFN service.

The remaining sections summarise information contained in Chapter 3 and Chapter 4. These contain guidelines to help integrate a naming system into the XFN federation. It is recommended that integrators publish specifications that define the behaviour of their XFN-conformant naming systems. This provides application writers that use XFN with necessary information for writing portable applications and administrators with the necessary information to integrate that naming system into the federation.

C.1 Implementation Models

These guidelines do not prescribe any particular implementation model but in order to appreciate the features of the different possible configurations of the XFN system, it might be helpful to understand the various building blocks and its required functionality.

The three diagrams Figure C-1, Figure C-2 and Figure C-3 serve as examples of the conceptual models of the different possible configurations. The dark shaded boxes shown in these diagrams are building blocks that a naming service integrator needs to provide in order to integrate the naming system with XFN. The modules depicted in the three diagrams are defined as follows:

XFN API
: The *XFN* API is the complete set of interface operations defined in this XFN specification.

XFN Framework
: The *XFN framework* is the implementation of the XFN API, including the client library and the service provider interfaces necessary for integrating native naming systems.

Context Implementation
: The *context implementation* is the naming service-specific module on the XFN client system that is required to integrate legacy naming systems with XFN. The code of the context implementation is a wrapper that maps the XFN API to the API exported by the legacy naming system. The complexity of the context implementation depends on how well the XFN API maps to the native naming service API and which XFN operations are to be supported. At a minimum, the name resolution phase of all operations must be supported.

  The techniques used to access the naming service-specific context implementations from the XFN framework may vary. For systems that support shared libraries and dynamic linking, a common approach might be that the context implementations are dynamically loadable modules.

  This approach of integrating a naming service using a context implementation module does not require any modification to the existing naming service's source code nor does it require access to the naming service's source code. All that is needed is access to the module (library) that exports the naming service-specific API. This approach is by far the easiest and fastest way of adding an existing naming system into the XFN federation.

XFN Client
: The *XFN client* is a module that implements the client protocol machines for the XFN protocols.

  Two XFN protocols are specified in Appendix A, the RPC based protocols for ONC+ systems (specified in RPCL) and for DCE environments (specified in IDL). The list of supported protocols might evolve over time.

  In addition to supporting the protocols, the XFN client might provide services typically offered by naming clients, such as caching. The extent of this support is implementation-specific.

XFN Protocol Exporter
: The *XFN protocol exporter* is the module required on systems that export one of the XFN protocols. This could be a new naming system, an existing naming system that was modified to also support XFN protocols, or a system that supports the XFN client library and also exports XFN protocols (capable of acting as surrogate client).

  The advantage for naming systems that support one of the specified protocols is that any existing XFN client that imports the protocols can be used to communicate with it. This is particularly useful for applications that need to export naming interfaces. Application

programmers do not have to duplicate the client-side implementation and they do not have to invent new naming interfaces. This provides additional benefits such as the ability to utilise caching and other mechanisms provided by the XFN client implementations, and a direct (and possibly more efficient) mapping of XFN operations to the application's naming operations.

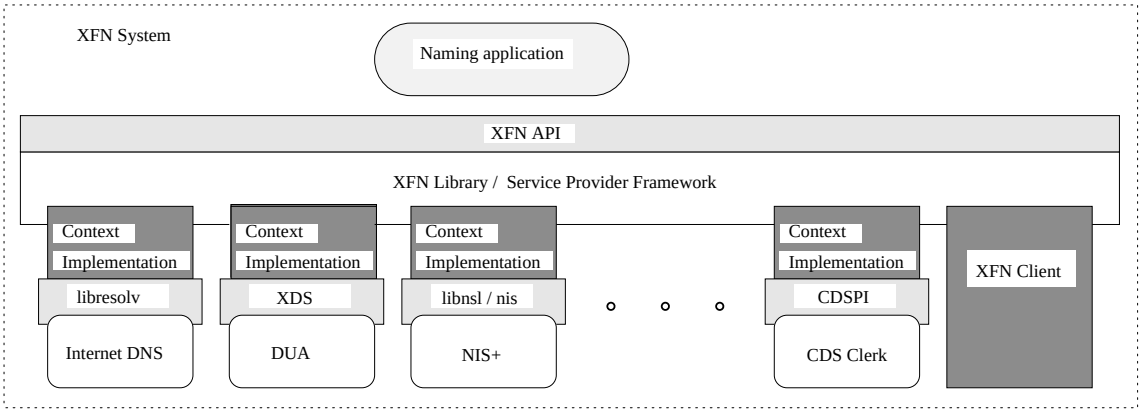

Figure C-1 XFN Configuration using Client Context Implementations

Figure C-1 shows the layering of the XFN client library on top of existing naming system clients on the same system. None of the legacy naming systems needs to be modified.

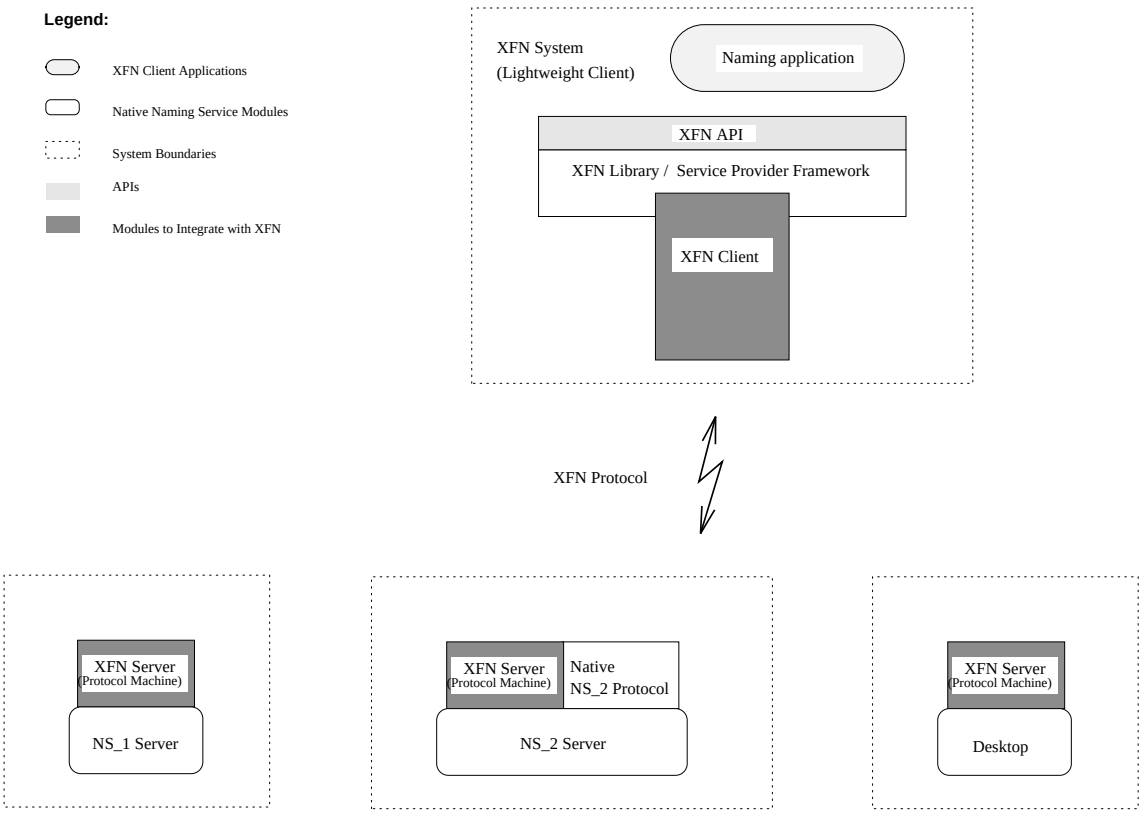

Figure C-2 Lightweight XFN Client Configuration

Figure C-2 shows multiple XFN systems that are connected via one of the specified XFN protocols. The client in this picture is a lightweight XFN client. The servers shown are name servers that directly export one of the specified XFN protocols.

The two modules shown in Figure C-3 are a lightweight XFN client and a server that acts as an intermediary. Similar to the client in Figure C-2, the client in Figure C-3 is a truly lightweight XFN client. None of the legacy naming system clients needs to be installed at that system. Depending on the client system's requirements, the XFN client can be implemented and configured to consume more or less resources, determined based on needs and availability. The XFN client might simply defer to mechanisms (such as for caching and replication) provided by the native naming system clients.

The legacy naming system clients in Figure C-3 reside on a remote system (similar to Figure C-1) that also exports at least one of the XFN protocols. This remote client can be viewed as a surrogate or proxy client that acts on behalf of the initial requestor and performs the native naming system functions.

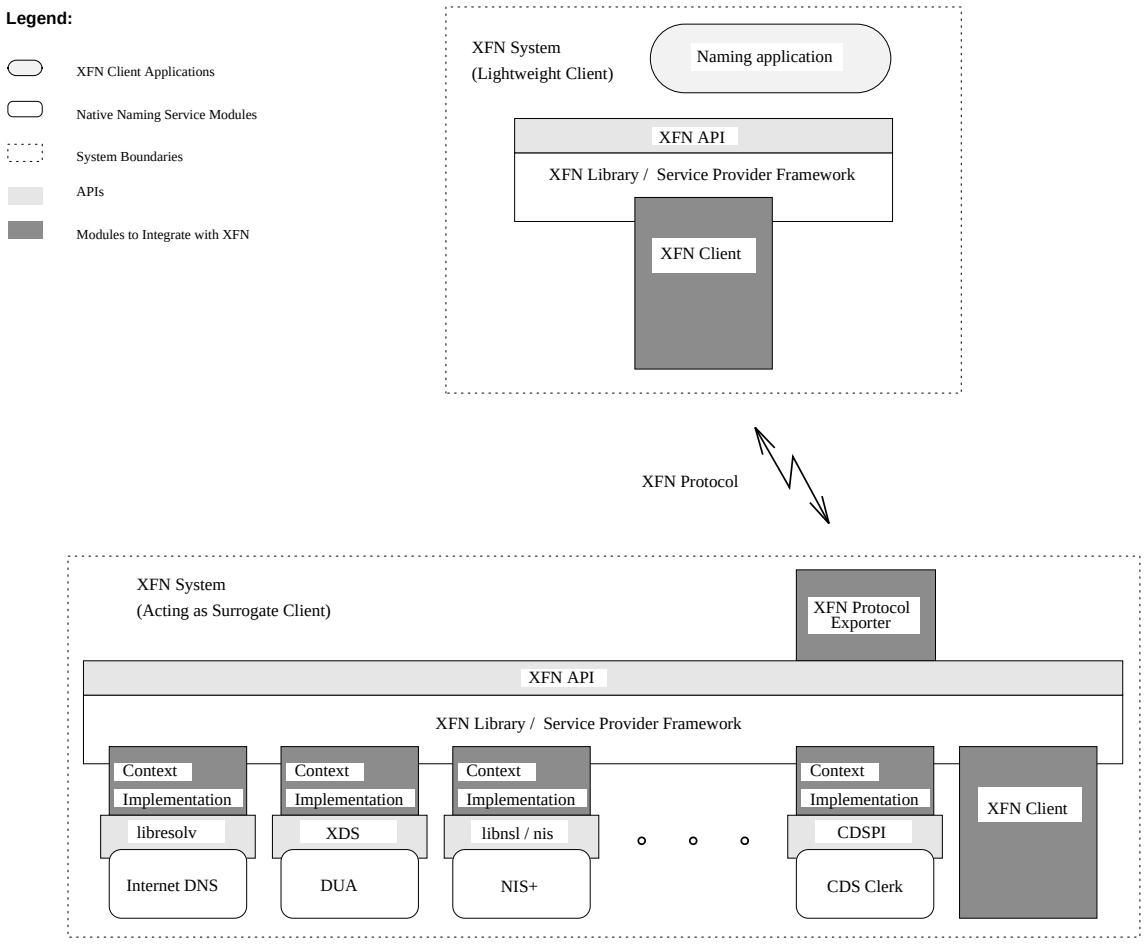

Figure C-3 XFN Configuration with Surrogate Client

Another aspect shown in Figure C-1 and Figure C-3 is the capability of the surrogate client to also import the XFN protocol (*XFN client* module). Such a configuration could serve emerging XFN servers or existing name servers that export one of the specified XFN protocols in addition to, or in replacement of, the native protocol.

Note that a *context implementation* precisely defines the set of modules that are co-located with the XFN framework to map the XFN API to the native naming service API. However, in the context of this appendix, we also use the term *context implementation* to mean the XFN mapping

code that is necessary at the server of a naming system that directly exports one of the XFN protocols (*XFN Protocol Exporter* and *XFN Server* in the diagrams).

C.2 Federating with other Naming Systems

The role and potential configuration of the naming system to be federated must be determined. The naming system might be a global naming system, and therefore should provide some means for registering enterprises and possibly other application naming systems. Or, the naming system could be an enterprise-level or application-specific naming system.

Furthermore, it must be specified whether the naming system must always be joined at the root or whether other entry points are also supported.

Any naming system that participates in the federation must define whether other naming systems can be federated as subordinate naming systems. Naming systems in the application namespaces (such as file systems) might not permit further federation.

Subordinate naming systems are federated through *next naming system* pointers (*nns* pointers). There are two flavours of XFN *nns* pointers: *junctions* and *implicit*. Both may be supported concurrently in a single naming system.

The criteria for whether a naming systems support junctions or implicit *nns* pointers, or both, are in Section 4.3 on page 63. For example, if a distinct name for naming the subordinate naming system is either not available (for example, already in use) or not desired (for example, it would not be a terminal name in that naming system), using implicit *nns* pointers is the preferred technique. An instance where junctions are better suited could be if subordinate naming systems must be selectable by explicit names. One might want to apply the rule of the thumb that implicit *nns* pointers are preferred for the global namespace (because X.500 and DNS names cannot be guaranteed to be leaf names) and most enterprise naming systems are expected to support junctions.

C.2.1 Junctions

Junctions are terminal names in a naming system that hold the reference information of subordinate naming system contexts.

The context implementation defines how references are constructed from information that is associated with the named entry. A common approach might be to use attributes for storing the appropriate reference and address information. Implementations might or might not permit the use of *bind* and *unbind* operations to create and delete references of junctions.

If junctions are supported, the context implementation must ensure the correct behaviour of XFN operations on junctions. In particular, the name resolution phase in XFN operations must follow a junction and resolve to the context that is bound by the reference of the junction.

C.2.2 Implicit Next Naming System Pointers

If the context implementation supports implicit next naming system pointers for federating naming systems, the context implementation must determine how references are managed.

The context implementation must ensure the correct behaviour of XFN operations on implicit *nns* pointers. If a name resolution is performed on a name that has a trailing XFN component separator, the implicit *nns* pointer must be followed.

C.3 Name Syntax

For naming systems participating in the naming federation it is necessary to specify the syntax of the compound name and to specify if the context supports the weak or strong separation model.

C.3.1 Weak and Strong Separation

If the component separator for atomic names of the naming system is distinct from the XFN component separator ('/') and if this XFN component separator character is not used in the compound name in unescaped and unquoted form, only the strong separation model applies.

If the atomic name separator is the same as the XFN component separator and the ordering of atomic names is **not** left-to-right, strong separation must be enforced by either quoting the compound name or escaping the atomic name separators. It must be specified which rule — quoting or escaping — applies (possibly both).

If the atomic name separator is the same as the XFN component separator and the ordering of atomic names **is** left-to-right, either strong or weak separation might be supported, or both. Please note, that in the instance where both forms of separation are supported, the context implementation must be prepared to receive in a XFN component name either a full compound name or a single atomic name. Weak separation can only be supported if at least one of the conditions that are specified in Section 4.2 on page 60 apply, namely the naming system must:

- be a terminal naming system
- if not terminal, the context must be able to do a syntax-specific discovery of naming system boundaries
- if not terminal, the context must be able to return the remaining unresolved components.

It must be specified which rule applies. If syntax-specific checking is done, the restriction for the subordinate naming systems must be clearly specified in order to avoid the use of (top level) names that conflict with the syntax rules. For instance, if the name syntax defines typed names for the atomic name component that uses the equal character ('=') as the type separator, the first name component of the subordinate naming system must not contain an unescaped equal character.

C.3.2 Syntax Attributes

In order to permit applications to use the operations on compound names, the context implementation must provide for a means of supporting the interface function for retrieving the syntax attributes (*fn_ctx_get_syntax_attrs*( )). If the "fn_syntax_type" attribute indicates the support of the XFN standard model, it can be assumed that the XFN framework implementation provides the support for interpreting the syntax attribute values. If the XFN standard model is not supported, the context implementation must provide the appropriate module for evaluating the syntax attributes and for compound name resolution. (Refer to Section 3.8 on page 50, for more information on the syntax attributes and the XFN standard model).

If the syntax attributes are fixed for a given naming system, the context implementation might not provide interfaces for modifying these attributes; in fact they might not be implemented as regular attributes of the naming system. It should be specified how these attributes are determined and whether they can be set and modified (through XFN attribute operations, for instance).

C.4 Context Operations

A conformance statement for the federated naming system must specify the level of support of the XFN context operations. The minimum requirement for XFN conformance is the support of the *fn_ctx_lookup*( ) operation and the support of the name resolution phase of the other operations.

Depending on the semantics of operations in the underlying naming system and the desired level of integration, other context operations might also be supported.

Operations that are not supported still must be provided with implementations that perform at least the name resolution phase to the target context. If the target context does not support the actual operation, the implementation returns the status [FN_E_OPERATION_NOT_SUPPORTED].

The context implementation may support operations such as *fn_ctx_list_names*( ), *fn_ctx_list_bindings*( ) and *fn_ctx_create_subcontext*( ), but the native naming system might not provide these as atomic operations. The atomicity guarantees for such operations must be specified.

There are a number of context operations whose behaviour depends on the semantics of operations in the underlying naming system. The implementation specifications must specify how the XFN operations effect the state of the underlying naming system. The XFN context operations that might incur different behaviour on different naming systems are detailed in the following subsections (For details on the operation's semantics, refer to the manual reference pages in Chapter 6.):

fn_ctx_bind( )
: Naming systems might or might not support the *exclusive* flag.

  Naming systems might or might not permit binding a name without also creating some attributes first. Also, the *fn_ctx_bind*( ) operation might only permit creation of certain types of objects in the namespace.

  If next naming system pointers are supported, implementations might not permit binding of these references through the *fn_ctx_bind*( ) operation.

fn_ctx_unbind( )
: The failure semantics of this operation is different depending on the type of object in the namespace. For example, an *fn_ctx_unbind*( ) on some contexts might not be permitted if it is not terminal (for example, if it is a directory).

fn_ctx_create_subcontext( )
: Similar to *fn_ctx_bind*( ), some naming systems might not permit creating a subcontext without also creating some attributes first.

  If naming systems support namespace partitioning or replicated servers, the *fn_ctx_create_subcontext*( ) might not provide the sufficient information to perform the creation.

fn_ctx_destroy_subcontext( )
: Similar to *fn_ctx_unbind*( ).

fn_ctx_rename( )
: The scope of the *fn_ctx_rename*( ) operation must be specified. Some naming systems might have restrictions on the semantics of rename. For example, only renames within the same context or on the same server might be permitted.

fn_ctx_lookup_link( )
: The implementation of XFN links must be specified (the information could possibly be stored in specific attributes). Also, the relationship of XFN links to any support for native soft links or aliases in the underlying naming system needs to be specified.

fn_ctx_handle_from_ref( )
: The implementation of the XFN framework in conjunction with the different context implementations determine the details of this operation. It is the responsibility of specific context implementations to define the properties of the **FN_ctx_t** object that is returned and to maintain the context handle and appropriate state information.

fn_ctx_get_ref( )
: Implementations may vary in their support for this operation in that the returned reference may contain a list of addresses that is different from the originally supplied to *fn_ctx_handle_from_ref*( ).

C.5 Attribute Operations

Similar to the context operations, a conformance statement for the federated naming system must specify the level of support of the XFN attribute operations.

The operations on attributes might be partially, fully, or not supported. Typically, partial support would contain the operations on single attributes and exclude the multi-attribute operations.

Operations that are not supported still must be provided with implementations that perform at least the name resolution phase to the target context. If the target context does not support the operation, the function returns the status [FN_E_OPERATION_NOT_SUPPORTED].

The context implementation may support operations such as *fn_attr_get_ids*( ), *fn_attr_multi_get*( ) and *fn_attr_multi_modify*( ) but the native naming system might not provide these as atomic operations. The atomicity guarantees for such operations must be specified.

If the attribute operations are at least partially supported, the implementation specification of the naming service must specify how its attribute model maps to the XFN attribute operations. Section 3.3.2 on page 27 discusses that the attribute model is not dictated by XFN. Some naming systems might associate attributes with the named objects, others might support attributes that are associated with names in contexts that are bound to objects, or naming systems support a combination of both. Some naming systems might not even distinguish between the two models.

Implementations might only support the attribute operations for certain types of attributes or attributes that are associated with certain type of entries in the namespace (for instance, directories or soft links may be excluded).

For the attribute modify operations, implementations must determine how the operation codes FN_ATTR_OP_ADD, FN_ATTR_OP_ADD_EXCLUSIVE, FN_ATTR_OP_REMOVE, FN_ATTR_OP_ADD_VALUES and FN_ATTR_OP_REMOVE_VALUES apply and what the exact semantics are.

C.5.1 Attributes and Next Naming System Pointers

In order to support next naming system pointers, the implementation specification must define how access to objects that the next naming system pointers is bound to (the contexts of the subordinate naming systems) can be disambiguated from access to attributes in the superior naming system. This disambiguation is particularly necessary for junctions where attributes are maintained in both the name of the junction in the superior naming system and the root context of the subordinate naming system.

If implicit *nns* pointers are used and one wants to access attributes that are associated with the subordinate naming system context, a trailing slash (XFN component separator) used for the resolution of the context disambiguates the access. If a trailing slash is passed in the *name* argument of an attribute operation, an implementation might support operations on attributes that are associated with the implicit *nns* pointer. (The analogy to the behaviour of junctions becomes obvious if one considers a trailing slash as a trailing empty name — this *nns* pointer is implicit).

For junctions, no trailing XFN component separators are permitted, but one typically accesses attributes associated with the subordinate naming system context if the context of the fully resolved name is passed as an argument to the attribute operation (the *name* argument is empty). In contrast, if the name of the junction is passed in the *name* argument, one might access attributes associated with the junction name (this might or might not be supported by naming system implementations).

Figure C-4 and Figure C-5 demonstrate possible associations between attributes, objects and names in a naming system. These diagrams show how one might use attributes to maintain references for next naming system pointers.

In the first example, Figure C-4, the reference of an implicit next naming system pointer is supported in the global namespace as an attribute that is associated with a context object.

If we take the name **.../umass.edu/** as an example (both with and without trailing slash), we find that there are four possible ways how this name can be represented to the attribute operations, depending on what the starting context (the object passed to the *ctx* argument) is set to:

ctx of	*name*	# in Diagram	Attribute Operation Refers To
`.../edu`	`umass`	(1)	Attributes associated with name **umass** in the context of **edu**
`.../umass.edu`	*empty*	(2)	Attribute associated with the context of *umass*
`.../umass.edu`	`"/"`	(3)	Attributes associated with anonymous *nns* pointer maintained in context **umass**
`.../umass.edu/`	*empty*	(4)	Attributes associated with root context of next naming system

The second example, Figure C-5, depicts a possible implementation for junctions. The junction's reference is maintained in an attribute (or conceivably set of attributes) that is associated with a terminal name of the enterprise naming system.

If we take the name **/.../umass.edu/science/service/spreadsheet** as an example for naming a junction, we have two ways of passing this to one of the attribute operations (note that trailing slashes are not permitted for junctions):

ctx of	*name*	# in Diagram	Attribute Operations Refers To
service	`spreadsheet`	(5)	Attributes associated with junction name **spreadsheet** in the context of **service**
service/spreadsheet	*empty*	(6)	Attributes associated with root context of next naming system

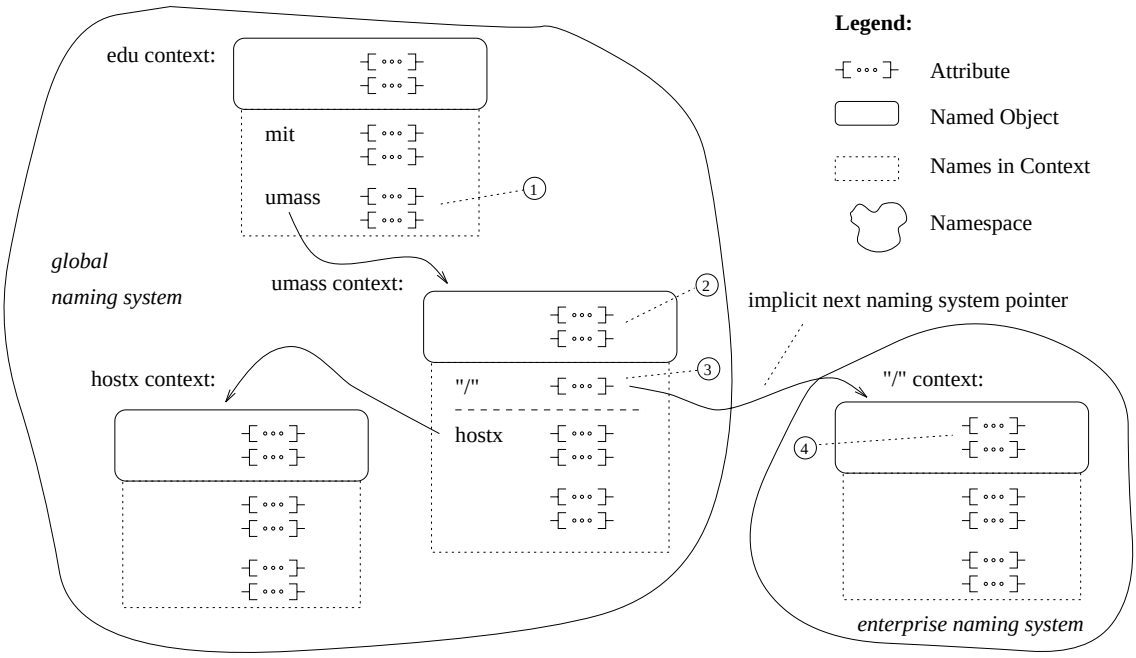

Figure C-4 Attribute Example with Implicit Next Naming System Pointer

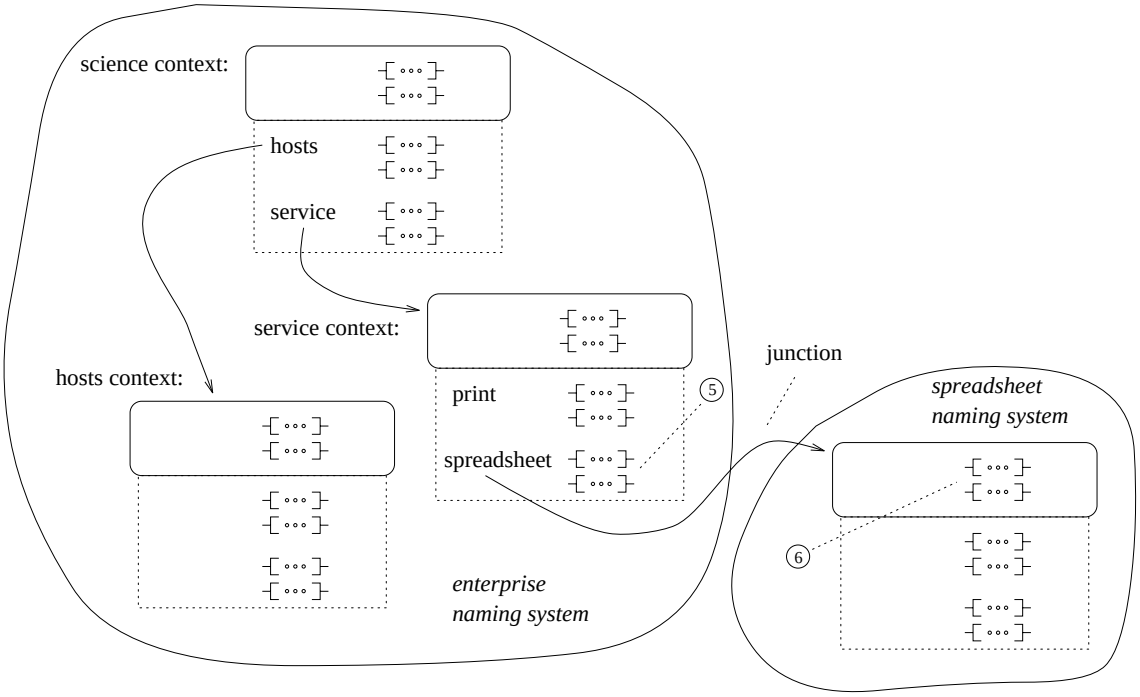

Figure C-5 Attribute Example with Junction

C.6 Reference and Address Types and its Registration

The values and encodings of reference and address types exported by the naming system need to be specified in order to support its federation by other naming systems. The implementor of the context implementation must decide which of these types are necessary to be registered with X/Open.

Generally, a single distinct reference type is specified for uniquely identifying a particular naming service. This reference type is used by the XFN framework as the discriminator for locating and dispatching to the context implementation.

XFN framework implementations may also permit default fallbacks where reference type specific implementations are not available. The address types can be used as discriminator instead. If a reference contains multiple address types, the order of selection is unspecified.

In cases where servers of a naming system directly export one of the XFN protocols, the appropriate reference types (for example, FN_DCE_SERVICE_REF) are used for selecting the client implementation. If implementations for the reference types are available, the XFN framework dispatches to the XFN client module that imports these protocols.

It might also be relevant for users and administrators of a federated naming system to know how XFN references and addresses map to its representation in the naming system. Since these might typically be stored in attributes, it is necessary for implementors of a federated naming system to define the format and encodings.

Appendix D

Policies for the Enterprise Namespace

Computing environments contain several independent naming systems that are interrelated in a navigational sense meaningful to users. Support for composite names in the base context and attribute interfaces enables applications and users to compose and use names that span these naming systems. The lack of common naming policies *within* a computing environment promotes incoherence in naming across different applications. Furthermore, the lack of common policies *across* different computing environments makes applications less portable and presents an impediment to developing distributed applications that span multiple computing environments. This appendix addresses these problems by providing policies for the enterprise namespace.

This appendix provides policies for XFN implementations that deal with enterprise naming systems. These policies are referred to as XFN-EP, for XFN enterprise policies. XFN-EP describes the arrangement of naming systems within an enterprise and how such naming systems can be used by applications. Global naming systems that already have policies for naming enterprise-level objects, such as users and hosts, are not covered under these policies. XFN-EP does not specify how these globally named objects are related to the objects named in the enterprise namespace.

The goals of XFN-EP are to promote the portability of applications and development of applications that span heterogeneous distributed computing environments. Applications and services that follow these policies benefit from these goals. Applications and services that extend these policies, or use modified versions of these policies may improve their usability with respect to certain environments, but are not guaranteed the same degree of portability.

Not all policies in XFN-EP are applicable in all environments. For example, some environments may have no notion of users. The intent of XFN-EP is to provide a minimal, yet sufficiently rich, set of policies so that applications need not invent or use ad hoc policies for specific environments. In an environment where only some of the policies in XFN-EP are meaningful, the parts of XFN-EP that are meaningful are used and those parts that are not meaningful are not used. There may be gradations of support of XFN-EP among systems. The purpose of the XFN-EP conformance statement of any particular system is to enable the application developer to decide when to use XFN-EP or some environment-specific policies.

D.1 Terminology

In addition to the terminology described in Section 5.1 on page 71, this appendix uses the following terms:

Organisational Units
An enterprise is organized into organisational units[4] such as centers, laboratories, departments, divisions, campuses, geographical sites and so on. An organisational unit is a subunit of an enterprise.

Namespace Identifier
A namespace identifier is an atomic name used to refer to the root of a namespace.

User Context
A context for naming objects related to a human user.

Username Context
A context for naming users.

Host Context
A context for naming objects related to a computer.

Hostname Context
A context for naming computers.

Organisational Unit Context
A context for naming objects related to an organisational unit.

4. The term *organisation* is used by some naming systems to refer to an enterprise (for example, X.500), and by others to mean an organisational unit. In this document, the terms *enterprise* and *organisational unit* are used to avoid ambiguity.

D.2 Policy Overview

The Enterprise Namespace

Within an enterprise, XFN-EP specifies:

- that there are *namespaces* for *organisational units, hosts, users, files* and *services*
- the *namespace identifiers* of these namespaces
- *composition policies* of how names from these namespaces should be composed, and consequently, the *relationships* among the underlying contexts.

XFN-EP does not specify:

- the syntax of names within each individual namespace
- how specific names are chosen for objects within these namespaces.

Initial Context

Each XFN client has an *Initial Context* that provides a starting point for resolving composite names. XFN-EP specifies *names* and *bindings* in the Initial Context for naming enterprise-related contexts.

D.3 The Enterprise Namespace

D.3.1 Types of Namespaces and Namespace Identifiers

Within an enterprise, XFN-EP specifies that there are namespaces for organisational units, hosts, users, files and services. There can be one or multiple namespaces of the same type within an enterprise. If there are multiple namespaces of the same type, XFN-EP does not specify the relationship between them.

XFN-EP specifies that these namespaces are referred to by the atomic names `_orgunit`, `_host`, `_user`, `_fs` and `_service`, respectively, as their canonical namespace identifiers (see Table D-1). These names are encoded using ISO 646 (same encoding as ASCII).

Canonical Namespace Identifier	Resolves to
`_orgunit`	context for naming organisational units
`_host`	context for naming hosts
`_user`	context for naming users
`_fs`	context for naming files
`_service`	context for naming services

Table D-1 XFN-EP Canonical Namespace Identifiers

A namespace identifier has a canonical representation and optionally one or more customized representations. Customized representations are implementation-dependent and defined by each context. A context implementation must accept both the canonical representation and its own customized representations of a namespace identifier and resolve them to the same reference. Because customized namespace identifiers are context implementation-dependent, they may not necessarily be portable to other contexts whereas canonical namespace identifiers are always portable to other contexts that accept namespace identifiers. For coherence, customized namespace identifiers should be consistent within an enterprise.

> For example, the canonical namespace identifier for the user namespace is `_user`, while, as a local policy in the `Wiz.COM` enterprise, a customized namespace identifier for the user namespace might be `employee`. The name `_user` can be used in any environment to mean the user namespace, whereas only contexts in the `Wiz.COM` enterprise accept the name `employee` to refer the user namespace.

The XFN component separator is used to delimit namespace identifiers.

> For example, composing the namespace identifier `_orgunit` with the organisational unit name `finance/accounts_payable` gives the composite name
> `_orgunit/finance/accounts_payable`.
>
> Composing the global name `…/Wiz.COM` with an organisational unit name `_orgunit/finance/accounts_payable` gives the composite name
> `…/Wiz.COM/_orgunit/finance/accounts_payable`.
>
> In another example, composing the name `_orgunit/finance/accounts_payable` with the name `_user/jsmith` gives the composite name
> `_orgunit/finance/accounts_payable/_user/jsmith`.

These namespaces may be supported by a federation of one or more naming systems.

If a naming system uses the same character as the XFN component separator as its atomic component separator and supports naming more than one type of these entities (organisational units, hosts, users, file systems or services), these types of entities may be represented by contexts within the same naming system.

> For example, if a naming system supports a single namespace for both organisational units and users, and uses a left-to-right, slash-separated syntax, a name for a user name might look like:
>
> `_orgunit/finance/accounts_payable/_user/jsmith`.

If a naming system supports naming more than one type of these entities but does not use the XFN component separator, the context implementation should still support the federated namespace structure as specified in this appendix.[5]

> For example, if two separate naming systems are used for organisational unit names (dot-separated, right-to-left syntax), user names (flat namespace), using similar names as the previous example, a user name might look like:
>
> `_orgunit/accounts_payable.finance/_user/jsmith`.

XFN-EP reserves for future extension the use of all atomic names beginning with the underscore character ('_') in contexts in which namespace identifiers can appear. XFN-EP does not otherwise restrict the use of these atomic names within other contexts or component naming systems. Some component naming systems, however, might have restrictions on the use of these names.

> For example, the atomic name `_service` is used as a canonical namespace identifier relative to a user name, as in `_user/jsmith/_service/calendar`, to mean the root of user `jsmith`'s service namespace. This does not preclude a system from using the name `_service` as a user name, as in `_user/_service`, because XFN-EP specifies that the context to which `_user/` is bound is for user names and not for namespace identifiers. Thus, in this example, `_service` is unambiguously interpreted as a user name.

D.3.2 Structure of the Enterprise Namespace

XFN-EP defines the structure of the enterprise namespace. The goal of this structure is to allow easy and uniform composition of names. This structure uses two main rules:

1. Objects with narrower scopes are named relative to objects with wider scopes.
2. Namespace identifiers are used to denote the transition from one namespace to the next.[6]

Table D-2 contains a summary of XFN-EP for arranging the enterprise namespace. Figure D-1 on page 281 shows an example of a namespace layout that uses XFN-EP.

XFN-EP specifies that the root context of an enterprise is a context for namespace identifiers. The namespace identifiers that may appear in this context are `_orgunit`, `_host`, `_user`, `_fs` and `_service`. They are bound to the contexts for naming organisational units, hosts, users, files and services, respectively.

---

5. For efficiency, the context implementation for such a naming system may support some syntactic mapping of composite names to names native to the naming system as part of the resolution process.
6. These namespaces could be implemented by the same or different naming systems.

Context Type	Subordinate Context	Parent Context	Arrangement of Contexts of Same Type
organisational unit	service user host file system	enterprise root	hierarchical
user	service file system	enterprise root organisational unit	not specified
host	service file system	enterprise root organisational unit	not specified
service	not specified	enterprise root organisational unit user host	not specified
file system	not specified	enterprise root organisational unit user host	not specified

Table D-2 Policies for Arranging the Enterprise Namespace

The namespace of an enterprise is structured around the hierarchical structure of organisational units of an enterprise. Names of hosts, users, files and services may be named relative to names of organisational units by composing the organisational unit name with the appropriate namespace identifier and object name. In a similar fashion (with the use of appropriate namespace identifiers), names of files and services may also be named relative to names of users or hosts.

> For example, the user `jsmith` in the Florida campus of an enterprise is named using the name `_orgunit/east/florida/_user/jsmith` (organisational unit names in this example have left-to-right, slash-separated name syntax). Note the use of the namespace identifier `_user` to denote the transition from the organisational unit namespace into the user namespace.

XFN-EP does not define the containment semantics among objects named in the hierarchical enterprise namespace. For example, XFN-EP does not specify the relationship between the users named relative to one organisational unit to those of another, possibly subordinate, organisational unit. Furthermore, the same type of object in different parts of the enterprise namespace need not be supported by the same naming system. Each organisational unit may maintain its own user and host namespace, possibly using different naming system technologies.

D.3.3 Policies for Naming Organisational Units

The organisational unit namespace provides a namespace for naming subunits of an enterprise. An example is shown in Figure D-1 on page 281.

XFN-EP specifies that the string `_orgunit` is the canonical namespace identifier for the organisational unit namespace. An organisational unit is named by composing `_orgunit`, or any of its customized representations, with its organisational unit name using the XFN component separator.

XFN-EP specifies that organisational units are named using compound names, where each atomic part names an organisational unit within a larger unit. XFN-EP does not specify the

syntax of organisational unit names.

> For example, using ONC/NIS+, the name `accounts_payable.finance` names an organisational unit `accounts_payable` within a larger one named `finance`. Similarly, another organisational unit is named in DCE/CDS using the name `east/chelmsford`. Composing these names with `_orgunit`, the resulting composite names for these organisational units are `_orgunit/accounts_payable.finance` and `_orgunit/east/chelmsford`, respectively.

XFN-EP specifies that organisational units may be named relative to the root context of an enterprise.

XFN-EP specifies that an organisational unit name is bound to an organisational unit context. An organisational unit context provides a context in which namespace identifiers can be bound. XFN-EP specifies that names of users, hosts, services and file systems, as well as organisational subunits, can be named relative to organisational unit names. When an organisational subunit is named, the atomic name separator of the compound name syntax is used to compose the subunit's name. When a user, host, service or file system is named relative to an organisational unit name, the XFN component separator is used.

> For example, a subunit `accounts_payable` of the organisational unit `finance` is named in NIS+ using the name `_orgunit/accounts_payable.finance` whereas a user in the `finance` organisational unit is named using the name `_orgunit/finance/_user/jsmith`.

XFN-EP does not specify the syntax of organisational unit names. XFN-EP also does not specify how organisational units are defined or assigned names.

> For example, one enterprise may choose to define their organisational units based on their corporate structure; another enterprise may choose to define their organisational units based on their geographical dispositions; yet another enterprise may choose to use a combination of geographic and corporate information.

D.3.4 Policies for Naming Users

The user namespace provides a namespace for naming human users in a computing environment.

XFN-EP specifies that the atomic name `_user` is the namespace identifier for the user namespace. A user is named by composing the namespace identifier for the user namespace, `_user`, or any of its customized representations, with the user name using the XFN component separator.

> For example, given an organisational unit name `_orgunit/finance`, the name `_orgunit/finance/_user` names the context for naming users.

XFN-EP does not specify the valid arrangement of names within the user namespace — it can be flat or hierarchical. For example, a hierarchical user naming system might reserve the top level of the user namespace for names of users; lower level names might further delineate different identity aspects of a user, such as his roles and capabilities. If user names have a hierarchical syntax, lower level names are composed using the atomic name separator of the user namespace syntax; objects in XFN-EP namespaces subordinate to a user name are named using the XFN component separator and the appropriate namespace identifier.

XFN-EP does not specify the syntax of user names or how the names are assigned.

XFN-EP specifies that user names can be named relative to an organisational unit or the root context of an enterprise.

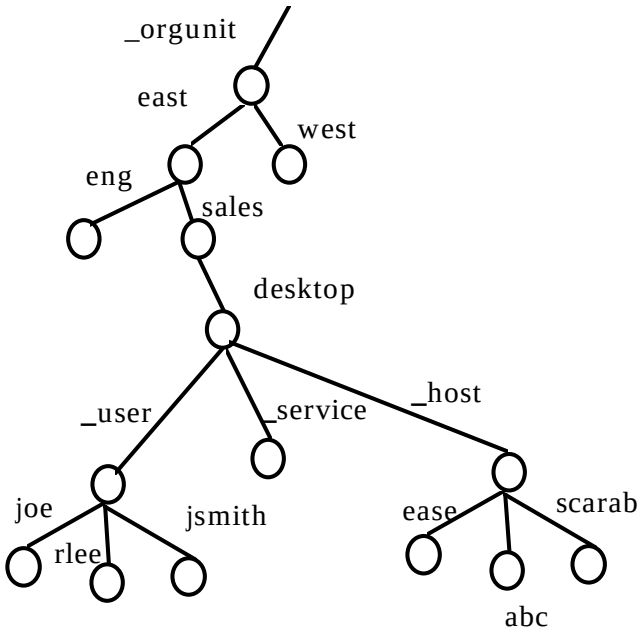

Figure D-1 Example of an Enterprise Namespace

XFN-EP specifies that a user name is bound to a user context. A user context provides a context in which namespace identifiers can be bound. XFN-EP specifies that names of files and services can be named relative to user names (using the XFN component separator and the appropriate namespace identifiers).

The user namespaces of different organisational units may be served by different naming systems, possibly using different naming technologies. XFN-EP does not specify the relationship between user namespaces.

D.3.5 Policies for Naming Hosts

The host namespace provides a namespace for naming computers.

XFN-EP specifies that the atomic name `_host` is the namespace identifier for the host namespace. A host is named by composing `_host`, or any of its customized representations, with its host name using the XFN component separator.

> For example, given an organisational unit name `_orgunit/finance`, the name `_orgunit/finance/_host` names the context for naming hosts.

XFN-EP does not specify the valid arrangement of names within the host namespace — it can be flat or hierarchical. For example, a hierarchical host naming system might reserve the top level

of the host namespace for machine names; lower level names might further delineate different identity aspects of a machine, such as its interfaces and capabilities. If host names have a hierarchical syntax, lower level names are composed using the atomic name separator of the host namespace syntax; objects in XFN-EP namespaces subordinate to a host name are named using the XFN component separator and the appropriate namespace identifier.

XFN-EP does not specify the syntax of host names or how the names are assigned.

XFN-EP specifies that hosts can be named relative to an organisational unit or the root context of an enterprise.

> For example, if …/`c=us/o=Wiz/_orgunit/accounts_payable.finance` names an organisational unit, then the name
> …/`c=us/o=Wiz/_orgunit/accounts_payable.finance/_host/grande`
> names a host `grande` within that organisational unit.

XFN-EP specifies that a host name is bound to a host context. A host context provides a context in which namespace identifiers can be bound. XFN-EP specifies that names of files and services can be named relative to host names (using the XFN component separator and the appropriate namespace identifiers).

The host namespaces of different organisational units may be served by different naming systems, possibly using different naming technologies. XFN-EP does not specify the relationship between host namespaces.

D.3.6 Policies for Naming Services

The service namespace provides a namespace for services used by or associated with objects within an enterprise. Examples of such services might be electronic calendars, faxes, mail and printing. XFN-EP does not specify which specific services use this namespace and whether it is utilized by object-based systems or procedural.

XFN-EP specifies that the atomic name `_service` is the namespace identifier for the service namespace. A service is named by composing `_service`, or any of its customized representations, with its service name using the XFN component separator.

XFN-EP does not specify the valid arrangement of names within the service namespace — it can be flat or hierarchical. If service names have a hierarchical syntax, lower level names are composed using the atomic name separator of the service namespace syntax; objects in XFN-EP namespaces subordinate to a service name are composed using the XFN component separator as a delimiter.

XFN-EP does not specify the syntax of service names or how service names are chosen. These are determined by service providers that share the service namespace.

A service can be named relative to an enterprise, an organisational unit, a user or a host.

> For example, if `_orgunit/accounts_payable.finance` names an organisational unit, then `_orgunit/accounts_payable.finance/_service/calendar` names its calendar service.

The service namespaces of different organisational units, users or hosts may be served by different naming systems, possibly using different naming technologies. XFN-EP does not specify the relationship between service namespaces.

XFN-EP does not specify what types of objects (for example, resources), if any, are named relative to a service object. Also, the XFN-EP does not specify the relationship between services, resources, and other types of objects controlled by object systems. Implementations determine the organisation of this namespace and the possible utilization of resource discovery

mechanisms. In some instances, the service namespace may be structured according to operations performed on objects, in others it might be data driven.

D.3.7 Policies for Naming Files

A *file naming system* (or simply *file system*) provides a namespace for naming files.

XFN-EP specifies that the atomic name `_fs` is the namespace identifier for the file namespace. A file is named by composing `_fs`, or any of its customized representations, with its file name using the XFN component separator.

> For example, if `_user/jsmith` names a user `jsmith`, the name `_user/jsmith/_fs/highlights93.mif` names her file `highlights93.mif`.

XFN-EP does not specify the file name syntax. The file name syntax depends on the file system being federated.

> For example, the name `_host/deedum/_fs/C:doc\strategy94.txt` is an example of a composite file name when the DOS file system is federated. The name `_user/jsmith/_fs/doc/orgchart.ps` is an example of a composite file name when a Unix file system is federated.

XFN-EP specifies that files can be named relative to an enterprise, an organisational unit, a user or a host.

XFN-EP does not specify what types of objects, if any, are named relative to a file.

The reference bound to `_fs` need not be one physical file system. It may be made up of multiple file systems, possibly of different types, composed into a single, virtual file system.

D.4 Bindings for the Enterprise in the Initial Context

In addition to the global policies specified by XFN (see Section 5.3 on page 73), XFN-EP specifies additional properties to be associated with the Initial Context.

XFN-EP specifies that *fn_ctx_handle_from_initial* ( ) is invoked on behalf of a user and host pair:

1. There is a user associated with the process when *fn_ctx_handle_from_initial* ( ) is invoked. In the following discussion this user is denoted by *U*. The association of user to process may change during the life of a process but does not affect the context handle originally returned by *fn_ctx_handle_from_initial* ( ).
2. The process is running on a host when *fn_ctx_handle_from_initial* ( ) is invoked. In the following discussion this host is denoted by *H*. The process to host association may change during the life of a process and may be interpreted differently on systems that support process migration or distributed processing. However, this does not affect the context handle originally returned by *fn_ctx_handle_from_initial* ( ).

XFN-EP specifies the following atomic names as namespace identifiers in the Initial Context: `_thishost`, `_thisorgunit`, `_thisens`, `_myself`, `_myorgunit`, `_myens`, `_orgunit`, `_user` and `_host`.[7] These names are encoded using ISO 646 (same encoding as ASCII). The bindings for these names are summarized in Table D-3. They have the same properties as the namespace identifiers described in Section D.3.1 on page 278.

Not all of these names need to appear in all Initial Contexts. For example, some machines or environments do not have the notion of a user, in which case, all atomic names relating to *U* will not appear in the Initial Context of those clients. Implementations are free to add names and bindings to the Initial Context but any such additions are not part of XFN-EP.

D.4.1 Host-related Bindings

The namespace identifiers `_thishost`, `_thisens` and `_thisorgunit` name contexts related to the machine the process is on when *fn_ctx_handle_from_initial* ( ) is invoked.

XFN-EP specifies that the namespace identifier `_thishost` resolves to the host context of *H*.

> For example, if the process is on the host `gofer`, `_thishost` resolves to the host context of `gofer` and the name `_thishost/_service/display` refers to the display service of `gofer`.

XFN-EP assumes that there is an association of a host to an enterprise. XFN-EP specifies that the namespace identifier `_thisens` resolves to the root context of the enterprise to which *H* belongs.

> For example, `_thisens/_service` resolves to the root of the service naming system in the enterprise of *H*.

XFN-EP assumes that there is an association of a host to an organisational unit of an enterprise. A host may be associated with multiple organisational units, but there must be one that is distinguished. XFN-EP does not specify how the determination of a host's affiliation with its distinguished organisational unit is made. The namespace identifier `_thisorgunit` resolves to the context of *H*'s distinguished organisational unit.

---

7. In addition to these bindings, the Initial Context also contains bindings for resolving global names: `...`, `_dns` and `_x500` (see Section 5.3 on page 73).

Namespace Identifier	Binding
`_thishost`	*H*'s host context.
`_thisens`	the root context of the enterprise of *H*.
`_thisorgunit`	*H*'s distinguished organisational unit context.
`_myself`	*U*'s user context.
`_myens`	the root context of the enterprise of *U*.
`_myorgunit`	*U*'s distinguished organisational unit context.
`_user`	the distinguished context in which users are named.
`_host`	the distinguished context in which hosts are named.
`_orgunit`	the distinguished context in which organisational units are named.

Table D-3 Enterprise-related Bindings in the Initial Context

For example, if *H* is in the organisational unit `accounts_payable.finance`, `_thisorgunit` resolves to the organisational unit context for `accounts_payable.finance` and `_thisorgunit/_service/fax` refers to the fax service of `accounts_payable.finance`.

D.4.2 User-related Bindings

The namespace identifiers `_myself`, `_myens` and `_myorgunit` name contexts related to the user associated with the process when *fn_ctx_handle_from_initial* ( ) is invoked.

XFN-EP specifies that the namespace identifier `_myself` resolves to the user context of *U*.

For example, if *U* is `jsmith`, `_myself` resolves to `jsmith`'s user context, and `_myself/_fs/.cshrc` names the file `.cshrc` of `jsmith`.

XFN-EP assumes that each user is affiliated with an enterprise. XFN-EP specifies that the namespace identifier `_myens` resolves to the root context of the enterprise to which *U* belongs.

For example, `_myens/_orgunit` resolves to the root of the organisational unit naming system in the enterprise of *U*.

XFN-EP assumes that each user is affiliated with an organisational unit of an enterprise. A user may be affiliated with multiple organisational units but there must be one that is distinguished, perhaps by its position in the organisational unit namespace or by the user's role in the organisational unit. XFN-EP does not specify how the determination of a user's affiliation with its distinguished organisational unit is made. XFN-EP specifies that the namespace identifier `_myorgunit` resolves to the context of *U*'s distinguished organisational unit.

For example, if *U* is in the organisational unit `accounts_payable.finance`, `_myorgunit` resolves to the organisational unit context for `accounts_payable.finance`, and `_myorgunit/_service/calendar` resolves to the calendar service of `accounts_payable.finance`.

D.4.3 Shorthand Bindings

The namespace identifiers `_orgunit`, `_user` and `_host` are the other bindings in the Initial Context. The exact bindings of these namespace identifiers are not specified by XFN-EP but are instead determined by the implementation and possibly customized by specific installations if allowed by the implementation. They are provided as shorthands for contexts for naming organisational units, hosts and users expected to be most frequently referenced in a particular environment. Applications or end-users needing a more precise association can use names starting with the user-related or host-related namespace identifiers described above.

For example, assume `_orgunit` is bound to the root of the organisational unit namespace of *U*, then the name `_orgunit/accounts_payable.finance` names an organisational unit `accounts_payable.finance` in the enterprise of *U*, and is equivalent to the name `_myens/_orgunit/accounts_payable.finance`.

For example, if `_host` is bound to the hostname context of *H*'s distinguished organisational unit, then using `_host` allows other hosts in the same organisational unit as *H* to be named from this context, and the names `_host` and `_thisorgunit/_host` resolve to the same context.

For example, if `_user` is bound to the username context of *U*'s distinguished organisational unit, `_user/jsmith` names the user `jsmith` at the same organisational unit as *U*, and the names `_user` and `_myorgunit/_user` resolve to the same context.

The meanings of `_orgunit`, `_user` and `_host` can vary among configurations. However, once the end-user is familiar with the configuration of a particular installation, she can use these namespace identifiers in her day-to-day operations. When she uses a system from another environment, with possibly a different and not-so-familiar configuration, she can use the longer forms to name precisely what she intends (for example, `_thisorgunit/_user/` instead of `_user/`). These namespace identifiers can also be used by applications that do not care about the precise definition but are instead interested in naming, for example, users in the distinguished context for users in any environment.

D.4.4 Relationships and Usage of Bindings

A typical situation is one in which many of the host-related and user-related bindings are the same.

`_thisens` and `_myens` are often bound to the same reference. They will be different only if the user using the system is from a foreign enterprise.

`_thisorgunit` and `_myorgunit` will be bound to the same reference if both *H* and *U* have the same distinguished organisational unit. For example, in a situation in which host-to-organisational unit assignments are based on the organisational units of machine owners (users), `_thisorgunit` and `_myorgunit` would often be bound to the same reference; they would be different only when *U* uses a machine from another organisational unit.

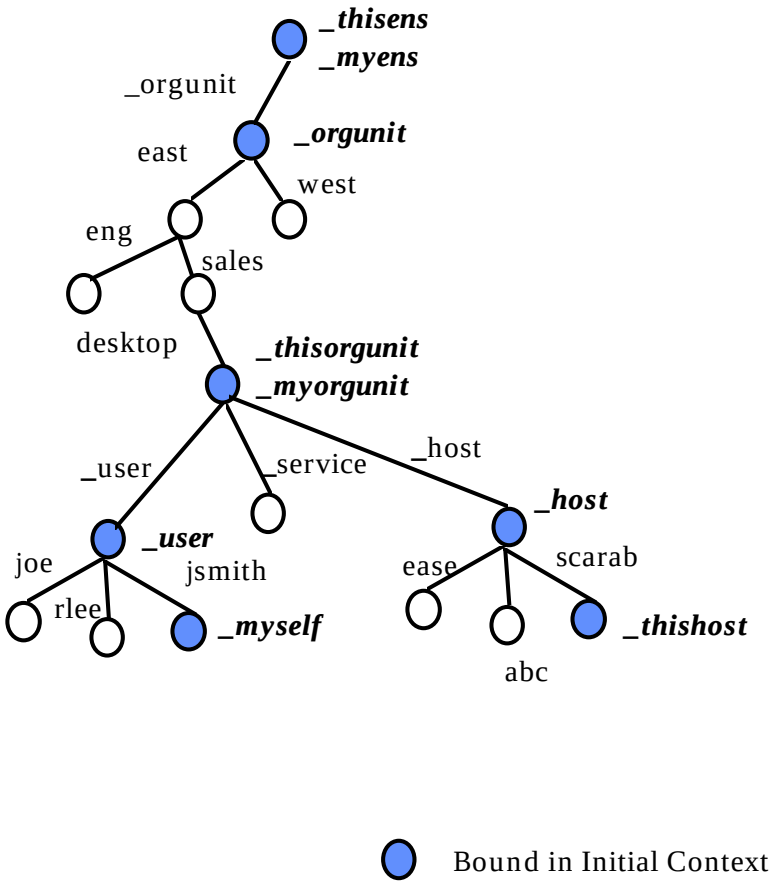

Figure D-2 Example of Enterprise Bindings in the Initial Context

Figure D-2 is an example of an Initial Context, in which *H* and *U* share many common affiliations. *U* is user `jsmith`, who is using machine `scarab`, *H*. Both `jsmith` and `scarab` belong to the same enterprise, and both have `east/sales/desktop` as their distinguished organisational unit. `_orgunit` is associated with *H*; `_user` and `_host` are associated with *H*'s distinguished organisational unit.

With the advent of mobile systems and globally accessible enterprises, situations in which many of the host-related and user-related bindings are different will become increasingly common. Therefore, instead of having a single set of bindings, XFN-EP defines two sets, one for the host and one for the user.

Figure D-3 contains another example of an Initial Context, in which *H* and *U* are in different enterprises. *U* is user `mjones`, of the `finance` organisational unit in some enterprise.

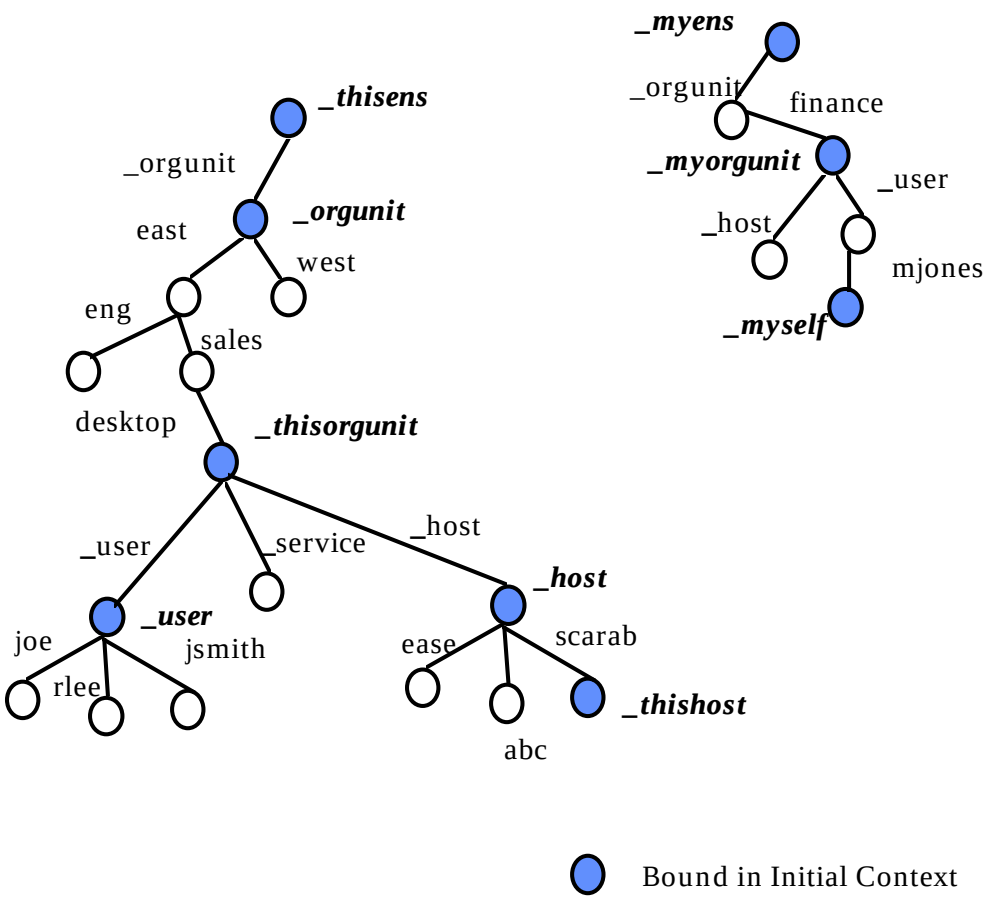

Figure D-3 Example of Bindings when *U* and *H* are in Different Enterprises

`mjones` is using machine `scarab`, *H*, in another enterprise. `scarab`'s distinguished organisational unit is `east/sales/desktop`. As in Figure D-2, `_orgunit` is associated with *H*; `_user` and `_host` are associated with *H*'s distinguished organisational unit.

When an enterprise is bound in a global namespace, the application and end-user have the additional capability of naming objects using global names.

> In Figure D-3, suppose user `mjones` is in the `c=jp/o=utokyo` enterprise, and the machine `scarab` is in the `c=us/o=Wiz` enterprise. Then, instead of using the name
>
> `_thisens/_orgunit/east/sales/desktop/_user/jsmith`
>
> to refer to a user in `scarab`'s enterprise, the user `mjones` could have used the name
>
> `…/c=us/o=Wiz/_orgunit/east/sales/desktop/_user/jsmith`.

Furthermore, if the user `mjones` wants to give out a name that can be used globally, he might give out the name

`…/c=jp/o=utokyo/_orgunit/finance/_user/mjones/_fs/memo.txt`

for a file that he usually accesses using the name `_myself/_fs/memo.txt`.

D.5 Examples of Composite Names

In the following examples, note that the specific choices of organisational unit names, user names, host names, file names and service names, and their syntaxes are just illustrative. They are not specified by XFN or XFN-EP.

D.5.1 Composing Names Starting with Global Names

Here are examples of names that begin with global components.

.../c=jp/o=utokyo/`_orgunit/grads.ai.cs/_user/mtakuda/_service/calendar`
names the calendar service of a user `mtakuda` in the organisational unit `grads.ai.cs` at the University of Tokyo in Japan.

.../c=us/o=Wiz/`_fs/export\usoft\products.txt`
names the file `export\usoft\products.txt` from the enterprise `Wiz`.

.../c=us/o=Wiz/`_service/fax`
names the fax service of the enterprise `Wiz`.

.../Wiz.com/`_orgunit/accounts_payable.finance/_service/fax`
names a fax service in the `accounts_payable.finance` organisational unit of the enterprise named by the DNS name `Wiz.com`.

.../Wiz.com/`_orgunit/b1.palo_alto.west.us/_service/fax`
names a fax service in Building 1 at the Palo Alto campus of Wiz.

D.5.2 Composing Names Starting with the Enterprise Root

The types of objects that may be named relative to the enterprise root are user, host, service, organisational unit, and file. Here are some examples of names that begin the enterprise root.

_thisens/`_orgunit/engineering/servers/multimedia`
names an organisational unit `engineering/servers/multimedia` in *H*'s enterprise.

_myens/`_user/hdiffie`
names the user `hdiffie` in *U*'s enterprise.

_myens/`_service/teletax`
names the teletax service of *U*'s enterprise.

D.5.3 Composing Names Starting with Organisational Units

The types of objects that may be named relative to an organisational unit name are: user, host, service and file. Here are some examples of names that begin with organisational unit names (either explicitly via `_orgunit`, or implicitly via `_thisorgunit` or `_myorgunit`), and name objects relative to organisational unit names when resolved in the Initial Context.

_orgunit/finance/`_user/mjones`
names a user `mjones` in the organisational unit `finance`.

_orgunit/finance/`_host/inmail`
names a machine `inmail` belonging to the organisational unit finance.

_orgunit/accounts_payable.finance/`_fs/pub/blue-and-whites/FY92-124`
names a file `pub/blue-and-whites/FY92-124` belonging to the organisational unit `accounts_payable.finance`.

_orgunit/accounts_payable.finance/_service/calendar
names the calendar service of the organisational unit accounts_payable.finance. This might manage the meeting schedules of the organisational unit.

_thisorgunit/_user/cmead
names the user cmead in *H*'s organisational unit.

_myorgunit/_fs/pub/project_plans/widget.ps
names the file pub/project_plans/widget.ps exported by *U*'s organisational unit's file system.

D.5.4 Composing Names Starting with Users

The types of objects that may be named relative to a user name are services and files. Here are some examples of names that begin with user names (explicitly via _user or implicitly via _myself), and name objects relative to users when resolved in the Initial Context.

_user/jsmith/_service/calendar
names the calendar service of the user jsmith.

_user/jsmith/_fs/bin/games/riddles
names the file bin/games/riddles of the user jsmith.

_user/rjones/_fs/games\crash.exe
names the file games\crash.exe of user rjones.

_myself/_service/printer/default
names the default printer service of *U*.

D.5.5 Composing Names Starting with Hosts

The types of objects that may be named relative to a host name are services and files. Here are some examples of names that begin with host names (explicitly via _host or implicitly via _thishost), and name objects relative to hosts when resolved in the Initial Context.

_host/mailhop/_service/mailbox
names the mailbox service associated with the machine mailhop.

_host/mailhop/_fs/pub/saf/archives.91
names the directory pub/saf/archives.91 found under the root directory of the machine mailhop.

_host/labpc/_fs/D:\udir\jsmith\games\mario.exe
names a file \udir\jsmith\games\mario.exe on drive D found on the machine labpc.

_thishost/_fs/D:\udir\jsmith\games\mario.exe
names a file \udir\jsmith\games\mario.exe on drive D found on *H*.

_thishost/_service/printer/default
names the default printer service of *H*.

Appendix E

Integrating File Services

A file service is an important service in distributed computing platforms today. It is an example of a service that incorporates a naming system for naming files.

Typically, a file namespace is hierarchically structured. Names representing non-leaf nodes identify directories. Leaf nodes generally identify files. A directory can be viewed as inheriting a naming context in which other directories and files are bound.

The main motivation for federating a file naming system is to enable clients of the file service to use composite names to access files through the file interface.

E.1 Using the XFN Interface for POSIX.1 File Systems

In most systems, XFN naming operations will not be exposed directly to the client of the file service. Systems typically provide a file service interface. The implementation of operations which accept file names as parameters needs to be modified to accept composite names and use the XFN interface. An example of such an operation is the *open*( ) system call in POSIX.1, which returns a descriptor of the file specified by the given name. The following outlines how this can be done in POSIX.1.

In POSIX.1, when a pathname (name of file or directory) that begins with a slash ('/') is specified as a parameter to a file operation, it is resolved with respect to the root directory of the local file system. Otherwise, it is resolved with respect to the current working directory, which is a context (directory) that is associated with the process.

Operations in the file interface are implemented as system calls. One or more file systems can be ''mounted'' at different points in the file namespace and are distinguished by the prefix of the pathname supplied to the operations.

Because all pathnames are interpreted with respect to the root directory or current working directory, the first question is how to incorporate an XFN composite name in a pathname. One solution is to effectively ''mount'' the XFN Initial Context in the root directory, thereby providing access to other federated file services. This requires reserving a directory name in the root directory of the file system. XFN recommends the name `xfn`. Thus, any pathname with a prefix `/xfn` is an XFN composite name.

The process that is mounted at `/xfn` is the name resolver and thus a client of the XFN interface. In ONC+, this can be the Automounter. In DCE, this can be the DFS cache-manager. Upon receiving a request for name resolution, the name resolver uses the *fn_ctx_lookup*( ) operation to obtain an XFN reference (**FN_ref_t**). The XFN reference is then used to derive the information for accessing the named file. The structure of the reference depends on the distributed file service.

Appendix F

Techniques for Extending XFN

The XFN interface is designed to be extensible.

Except for the types **FN_attrvalue_t** and **FN_identifier_t**, all other data types defined in the interface are defined so as to hide their actual data representation from the client. The actual representations of these types are not defined by XFN but by XFN implementations.

The methods for extending the set of operations on any of the abstract types, including those on the context, are similar. The following discussion focuses on extending the context interface. The same techniques can be applied to any of the other abstract data types.

F.1 Extending the C Context Interface

In the C context interface, the client performs operations on a context through an interface that XFN defines, and refers to the context through a pointer to the object.

There are two approaches for extending the C context interface:

1. Make the **FN_ctx_t** object extensible.

   Operations are added that operate on objects of type **FN_ctx_t**. For example, one might add an operation *fn_ctx_list_with_filter*( ), which is similar to *fn_ctx_list_names*( ) except it accepts a filter for selecting which names to return.

   ```
   FN_filtered_namelist_t *fn_ctx_list_with_filter(
       FN_ctx_t *ctx,
       const FN_filter_t *filter,
       FN_status_t *status);
   ```

   Support for some of these extended operations may require access to additional data related to the context. For example, if the extended context is a user context, the extended operations may require access to data related to the user. The implementation that supplies the **FN_ctx_t** object determines whether such support is possible. Using an extended operation on a non-extended **FN_ctx_t** object is undefined and if at all possible, should be reported to the client.

2. Declare new types of context objects.

   Both context and extended operations are supported through newly defined operations that take the new type as parameter. One of these operations must be a function analogous to *fn_ctx_handle_from_ref*( ), which, given a reference, returns the handle to an object of the new context type. For example, **FN_user_ctx_t** is a new type of context, and the following are examples of declarations for a context operation and an extended operation.

   ```
   FN_ref_t *fn_user_ctx_lookup(
       FN_user_ctx_t *uctx,
       const FN_composite_name_t *name,
       FN_status_t *status);

   uid_t fn_ctx_get_uid(
       FN_user_ctx_t *ctx,
       FN_status_t *status);
   ```

   The extended type, **FN_user_ctx_t** in the example, would encapsulate the object that it is extending, **FN_ctx_t** in the example. For example, *fn_user_ctx_lookup*( ) might be implemented by extracting an **FN_ctx_t** pointer from the given *uctx*, and passing the **FN_ctx_t** pointer to *fn_ctx_lookup*( ).

In both approaches, an extended operation cannot be invoked with a composite name argument where the intent is to perform composite name resolution to the target context and then perform the extended operation. This is not possible because the intermediate contexts may not necessarily be extended in the same way. The steps for invoking an extended operation are:

1. Obtain a reference to the target extended context.
2. Use *fn_ctx_handle_from_ref*( ) or an equivalent of this for the extended context object to create a handle to the extended context object.

3. Invoke the extended operation using the handle.

The following example illustrates how a client could access an extended operation *fn_user_ctx_get_uid*( ), using the second approach of extension described above.

```
FN_string_t *input_username;
FN_composite_name_t *uname;
FN_ref_t *eref;
FN_status_t *status = fn_status_create();

/* 1.  Obtain reference to extended context */
uname = fn_composite_name_from_string(input_username);
eref = fn_ctx_lookup(init_ctx, uname, status);
/* check status */

if (eref) {
    /* 2.  Obtain handle to extended context object */
    FN_user_ctx_t *uctx = fn_user_ctx_handle_from_ref(eref, status);
    /* check status */

    /* 3.  Perform extended operation on object */
    if (uctx) {
        uid_t thisuid;
        thisuid = fn_user_ctx_get_uid(uctx, status);
        /* check status */
    }
    ...
}
```

Appendix G

Registry of Types, Identifiers and Code Sets

This appendix contains a registry of identifiers and codes used by XFN implementations. These include:

- reference types
- address types and address formats
- attribute syntaxes and identifiers
- code sets used in character strings
- extended operations for search filter expression.

When an implementation requires the use of an item (such as a reference type or address type) and an item with the same semantics is already defined in the registry, the item in the registry should be used. If the item has different semantics than the items already in the registry, a new item should be defined. If the scope of use of the item is expected to be external to the environment that defines it, the item should be added to the registry.

The registry in this Appendix will be maintained by X/Open as a part of maintaining this document.

G.1 Reference Types

G.1.1 XFN Standard References

Reference Type Description	Identifier Format	Identifier Value
XFN link reference	FN_ID_STRING	fn_link_ref
XFN null reference	FN_ID_STRING	fn_null_ref

An XFN link reference consists of a single address, with address type *fn_link_addr*.

An XFN null reference contains no addresses.

G.1.2 Naming Service-dependent References

Reference Type Description	Identifier Format	Identifier Value
FN_DCE_CDS_REF	FN_ID_ISO_OID_STRING	1.3.22.1.6.1.1
FN_DCE_RPC_SERVER_REF	FN_ID_ISO_OID_STRING	1.3.22.1.6.1.2
FN_DCE_XFN_SERVER_REF	FN_ID_ISO_OID_STRING	1.3.22.1.6.1.3
FN_DCE_GROUP_REF	FN_ID_ISO_OID_STRING	1.3.22.1.6.1.4
FN_DCE_DFS_REF	FN_ID_ISO_OID_STRING	1.3.22.1.6.1.5
FN_DCE_SEC_REF	FN_ID_ISO_OID_STRING	1.3.22.1.6.1.6
FN_CPIC_PGM_INST_REF	FN_ID_ISO_OID_STRING	1.3.18.0.2.6.7
ONC Enterprise Context	FN_ID_STRING	onc_fn_enterprise
ONC File System Context	FN_ID_STRING	onc_fn_fs
ONC Organization Context	FN_ID_STRING	onc_fn_organization
ONC Printername Context	FN_ID_STRING	onc_fn_printername
ONC Printer Context	FN_ID_STRING	onc_printers
ONC Site Context	FN_ID_STRING	onc_fn_site
ONC Username Context	FN_ID_STRING	onc_fn_username
ONC Hostname Context	FN_ID_STRING	onc_fn_hostname
ONC User Context	FN_ID_STRING	onc_fn_user
ONC Host Context	FN_ID_STRING	onc_fn_host
ONC Service Context	FN_ID_STRING	onc_fn_service
ONC Namespace Id Context	FN_ID_STRING	onc_fn_nsid
ONC Null Context	FN_ID_STRING	onc_fn_null
ONC Generic Context	FN_ID_STRING	onc_fn_generic
Internet domain	FN_ID_STRING	inet_domain
Internet host	FN_ID_STRING	inet_host
X.500 Object	FN_ID_STRING	x500

G.2 Address Types and Address Formats

G.2.1 XFN Standard Addresses

Address Type Description	Identifier Format	Identifier Value
XFN link address	FN_ID_STRING	fn_link_addr

An XFN link address contains the string form of the composite name (that returned by *fn_string_from_composite_name*( ) when applied to an **FN_composite_name_t** object.)

G.2.2 Naming Service-dependent Addresses

Address Type Description	Identifier Format	Identifier Value
FN_DCE_RPC_SERVER_ADDR	FN_ID_ISO_OID_STRING	1.3.22.1.6.2.1
FN_DCE_GROUP_MEMBER_ADDR	FN_ID_ISO_OID_STRING	1.3.22.1.6.2.2
FN_CPIC_PGM_INST_ADDR	FN_ID_ISO_OID_STRING	1.3.18.0.2.4.14
ONC FN / NIS+ Address	FN_ID_STRING	onc_fn_nisplus
ONC FN / NIS+ Root Address	FN_ID_STRING	onc_fn_nisplus_root
ONC FN / NIS (YP) Address	FN_ID_STRING	onc_fn_nis
ONC FN Service Address	FN_ID_STRING	onc_fn_generic
ONC RPC Address	FN_ID_STRING	onc_rpc
IP Address	FN_ID_STRING	inet_ipaddr_string
X.500 Address	FN_ID_STRING	x500
OSI Presentation Address	FN_ID_STRING	osi_paddr
Internet Domain	FN_ID_STRING	inet_domain

G.3 Attribute Identifiers and Attribute Syntaxes

G.3.1 Attribute Identifiers

Attribute Identifier Description	Identifier Format	Identifier Value
syntax model type	FN_ID_STRING	fn_syntax_type
syntax direction	FN_ID_STRING	fn_std_syntax_direction
atomic component separator	FN_ID_STRING	fn_std_syntax_separator
escape character(s)	FN_ID_STRING	fn_std_syntax_escape
begin-quote from first quote set	FN_ID_STRING	fn_std_syntax_begin_quote1
end-quote from first quote set	FN_ID_STRING	fn_std_syntax_end_quote1
begin-quote from second quote set	FN_ID_STRING	fn_std_syntax_begin_quote2
end-quote from second quote set	FN_ID_STRING	fn_std_syntax_end_quote2
attribute-value-assertion_separator	FN_ID_STRING	fn_std_syntax_ava_separator
typed-value_separator	FN_ID_STRING	fn_std_syntax_typeval_separator
locales supported	FN_ID_STRING	fn_std_syntax_locales
FN_CPIC_PGM_PFID	FN_ID_ \ ISO_OID_STRING	1.3.18.0.2.4.13

G.3.2 Attribute Syntaxes

Attribute Syntax Description	Identifier Format	Identifier Value
ASCII string	FN_ID_STRING	fn_attr_syntax_ascii
Locale array	FN_ID_STRING	fn_attr_syntax_locale_array

An attribute value with syntax **fn_attr_syntax_ascii** contains a linear sequence of ASCII characters.

fn_attr_syntax_locale_array is defined by the following structures:

```
struct {
    unsigned long   code_set;
    unsigned long   lang_terr;
} fn_attr_syntax_locale_info_t;

struct {
    size_t          num_locales;
    fn_attr_syntax_locale_info_t
                    *locales;     /* pointer to array of locales */
} fn_attr_syntax_locales_t;
```

G.4 Code Sets

XFN uses the code sets as defined by the DCE RFC 40.1.

The default code set is ISO 646 (code set ID 0x00010020).

G.5 Extended Operations for Search Filter Expression

The following three extended operations are currently defined:

'name'(<Wildcarded String>)
: The identifier for this operation is 'name' ({FN_ID_STRING}). The argument to this operation is a wildcarded string. The operation returns TRUE if the name of the object matches the supplied wildcarded string.

'reftype'(%i)
: The identifier for this operation is 'reftype' ({FN_ID_STRING}). The argument to this operation is an identifier. The operation returns TRUE if the reference type of the object is equal to the supplied identifier.

'addrtype'(%i)
: The identifier for this operation is 'addrtype' ({FN_ID_STRING}). The argument to this operation is an identifier. The operation returns TRUE if any of the address types in the reference of the object is equal to the supplied identifier.

Appendix H

Headers

This chapter defines the header file used in the XFN client interface. The **<xfn/xfn.h>** header contains the XFN interface declarations for:

1. the XFN base context interface
2. the XFN extended attribute interface
3. the XFN base attribute interface
4. status object and status codes used by operations in these three interfaces
5. abstract data types passed as parameters to, and returned as values from, operations in these three interfaces
6. the interface for the XFN standard syntax model for parsing compound names.

H.1 Synopsis

The XFN client interface can be referenced using the following directive:

```
#include <xfn/xfn.h>
```

H.2 Structures

The **<xfn/xfn.h>** header declares the following structures:

FN_identifier_t

```
struct {
    unsigned int    format;
    size_t          length;
    void           *contents;
} FN_identifier_t;
```

FN_attrvalue_t

```
struct {
    size_t    length;
    void     *contents;
} FN_attrvalue_t;
```

H.3 Enumeration Types

The **<xfn/xfn.h>** header defines the following enumeration types:

String Index Operation Types

```
enum {
    FN_STRING_INDEX_NONE = -1,
    FN_STRING_INDEX_FIRST = 0,
    FN_STRING_INDEX_LAST = INT_MAX
};
```

Identifier Types

```
enum {
    FN_ID_STRING,
    FN_ID_DCE_UUID,
    FN_ID_ISO_OID_STRING,
    /* others...*/
};
```

Status Codes

```
enum {
    FN_SUCCESS = 1,
    FN_E_LINK_ERROR,
    FN_E_CONFIGURATION_ERROR,
    FN_E_NAME_NOT_FOUND,
    FN_E_NOT_A_CONTEXT,
    FN_E_LINK_LOOP_LIMIT,
    FN_E_MALFORMED_LINK,
    FN_E_ILLEGAL_NAME,
    FN_E_CTX_NO_PERMISSION,
    FN_E_NAME_IN_USE,
    FN_E_OPERATION_NOT_SUPPORTED,
    FN_E_COMMUNICATION_FAILURE,
    FN_E_CTX_UNAVAILABLE,
    FN_E_NO_SUPPORTED_ADDRESS,
    FN_E_MALFORMED_REFERENCE,
    FN_E_AUTHENTICATION_FAILURE,
    FN_E_INSUFFICIENT_RESOURCES,
    FN_E_CTX_NOT_EMPTY,
    FN_E_NO_SUCH_ATTRIBUTE,
    FN_E_INVALID_ATTR_IDENTIFIER,
    FN_E_INVALID_ATTR_VALUE,
    FN_E_TOO_MANY_ATTR_VALUES,
    FN_E_ATTR_VALUE_REQUIRED,
    FN_E_ATTR_NO_PERMISSION,
    FN_E_PARTIAL_RESULT,
    FN_E_INVALID_ENUM_HANDLE,
    FN_E_SYNTAX_NOT_SUPPORTED,
    FN_E_INVALID_SYNTAX_ATTRS,
    FN_E_INCOMPATIBLE_CODE_SETS
```

```
    FN_E_CONTINUE,
    FN_E_UNSPECIFIED_ERROR,
    FN_E_NO_EQUIVALENT_NAME,
    FN_E_ATTR_IN_USE,
    FN_E_INCOMPATIBLE_LOCALES,
    FN_E_SEARCH_INVALID_FILTER,
    FN_E_SEARCH_INVALID_OP,
    FN_E_SEARCH_INVALID_OPTION
};
```

Attribute Modification Types

```
enum {
    FN_ATTR_OP_ADD = 1,
    FN_ATTR_OP_ADD_EXCLUSIVE,
    FN_ATTR_OP_REMOVE,
    FN_ATTR_OP_ADD_VALUES,
    FN_ATTR_OP_REMOVE_VALUES
};
```

H.4 Data Types

The **<xfn/xfn.h>** header defines the following data types through **typedef**:

FN_identifier_t

```
typedef struct {
    unsigned int format;
    size_t length;
    void *contents;
} FN_identifier_t;
```

FN_attrvalue_t

```
typedef struct {
    size_t length;
    void *contents;
} FN_attrvalue_t;
```

Other data types used in the XFN client interface are defined as abstract data types. They contain no explicit data type definitions and can only be manipulated through operations on the type.

H.5 Functions

The **<xfn/xfn.h>** header declares the following as functions.

H.5.1 Operations on FN_string_t

```
extern FN_string_t *fn_string_create(void);
extern void fn_string_destroy(FN_string_t *);

extern FN_string_t *fn_string_from_str(const unsigned char *str);
extern FN_string_t *fn_string_from_str_n(
    const unsigned char *str, size_t storlen);
extern const unsigned char *fn_string_str(
    const FN_string_t *, unsigned int *status);

extern FN_string_t *fn_string_from_contents(
    unsigned long code_set,
    unsigned long lang_terr,
    size_t charcount,
    size_t bytecount,
    const void *contents,
    unsigned int *status);

unsigned long fn_string_code_set(
    const FN_string_t *str);

unsigned long fn_string_lang_terr(
    const FN_string_t *str);

extern size_t fn_string_charcount(const FN_string_t *);
extern size_t fn_string_bytecount(const FN_string_t *);
extern const void *fn_string_contents(const FN_string_t *);

extern FN_string_t *fn_string_copy(const FN_string_t *);
extern FN_string_t *fn_string_assign(
    FN_string_t *dst,
    const FN_string_t *src);
extern FN_string_t *fn_string_from_strings(
    unsigned int *status,
    const FN_string_t *s1,
    const FN_string_t *s2,
    ...);
extern FN_string_t *fn_string_from_substring(
    const FN_string_t *,
    int first,
    int last);

extern int fn_string_is_empty(const FN_string_t *);
extern int fn_string_compare(
    const FN_string_t *s1,
    const FN_string_t *s2,
    unsigned int string_case,
    unsigned int *status);
```

```
extern int fn_string_compare_substring(
    const FN_string_t *s1,
    int first,
    int last,
    const FN_string_t *s2,
    unsigned int string_case,
    unsigned int *status);
extern int fn_string_next_substring(
    const FN_string_t *str,
    const FN_string_t *sub,
    int index,
    unsigned int string_case,
    unsigned int *status);
extern int fn_string_prev_substring(
    const FN_string_t *str,
    const FN_string_t *sub,
    int index,
    unsigned int string_case,
    unsigned int *status);
```

H.5.2 Operations on FN_composite_name_t

```
extern FN_composite_name_t *fn_composite_name_create(void);
extern void fn_composite_name_destroy(FN_composite_name_t *);
extern FN_composite_name_t *fn_composite_name_from_string(
    const FN_string_t *);
extern FN_composite_name_t *fn_composite_name_from_str(
    const unsigned char *cstr);
extern FN_string_t *fn_string_from_composite_name(
    const FN_composite_name_t *,
    unsigned int *status);

extern FN_composite_name_t *fn_composite_name_copy(
    const FN_composite_name_t *);
extern FN_composite_name_t *fn_composite_name_assign(
    FN_composite_name_t *dst,
    const FN_composite_name_t *src);
extern int fn_composite_name_is_empty(const FN_composite_name_t *);
extern unsigned int fn_composite_name_count(
    const FN_composite_name_t *);

extern const FN_string_t *fn_composite_name_first(
    const FN_composite_name_t *,
    void **iter_pos);
extern const FN_string_t *fn_composite_name_next(
    const FN_composite_name_t *,
    void **iter_pos);
extern const FN_string_t *fn_composite_name_prev(
    const FN_composite_name_t *,
    void **iter_pos);
extern const FN_string_t *fn_composite_name_last(
    const FN_composite_name_t *,
    void **iter_pos);
```

```
extern FN_composite_name_t *fn_composite_name_prefix(
    const FN_composite_name_t *,
    const void *iter_pos);
extern FN_composite_name_t *fn_composite_name_suffix(
    const FN_composite_name_t *,
    const void *iter_pos);

extern int fn_composite_name_is_equal(
    const FN_composite_name_t *n1,
    const FN_composite_name_t *n2,
    unsigned int *status);
extern int fn_composite_name_is_prefix(
    const FN_composite_name_t *,
    const FN_composite_name_t *prefix,
    void **iter_pos,
    unsigned int *status);
extern int fn_composite_name_is_suffix(
    const FN_composite_name_t *,
    const FN_composite_name_t *suffix,
    void **iter_pos,
    unsigned int *status);

extern int fn_composite_name_prepend_comp(
    FN_composite_name_t *,
    const FN_string_t *);
extern int fn_composite_name_append_comp(
    FN_composite_name_t *,
    const FN_string_t *);
extern int fn_composite_name_insert_comp(
    FN_composite_name_t *,
    void **iter_pos,
    const FN_string_t *);
extern int fn_composite_name_delete_comp(
    FN_composite_name_t *,
    void **iter_pos);

extern int fn_composite_name_prepend_name(
    FN_composite_name_t *,
    const FN_composite_name_t *);
extern int fn_composite_name_append_name(
    FN_composite_name_t *,
    const FN_composite_name_t *);

extern int fn_composite_name_insert_name(
    FN_composite_name_t *,
    void **iter_pos,
    const FN_composite_name_t *);
```

H.5.3 Operations on FN_ref_addr_t

```
extern FN_ref_addr_t *fn_ref_addr_create(
    const FN_identifier_t *type,
    size_t len,
    const void *data);
extern void fn_ref_addr_destroy(FN_ref_addr_t *);

extern FN_ref_addr_t *fn_ref_addr_copy(const FN_ref_addr_t *);
extern FN_ref_addr_t *fn_ref_addr_assign(
    FN_ref_addr_t *dst,
    const FN_ref_addr_t *src);

extern const FN_identifier_t *fn_ref_addr_type(const FN_ref_addr_t *);
extern size_t fn_ref_addr_length(const FN_ref_addr_t *);
extern const void *fn_ref_addr_data(const FN_ref_addr_t *);

extern FN_string_t *fn_ref_addr_description(
    const FN_ref_addr_t *,
    unsigned int detail,
    unsigned int *more_detail);
```

H.5.4 Operations on FN_ref_t

```
extern FN_ref_t *fn_ref_create(const FN_identifier_t *ref_type);
extern void fn_ref_destroy(FN_ref_t *);
extern FN_ref_t *fn_ref_copy(const FN_ref_t *);
extern FN_ref_t *fn_ref_assign(FN_ref_t *dst, const FN_ref_t *src);

extern const FN_identifier_t *fn_ref_type(const FN_ref_t *);
extern unsigned int fn_ref_addrcount(const FN_ref_t *);

extern const FN_ref_addr_t *fn_ref_first(
    const FN_ref_t *,
    void **iter_pos);
extern const FN_ref_addr_t *fn_ref_next(
    const FN_ref_t *,
    void **iter_pos);
extern int fn_ref_prepend_addr(FN_ref_t *, const FN_ref_addr_t *);
extern int fn_ref_append_addr(FN_ref_t *, const FN_ref_addr_t *);
extern int fn_ref_insert_addr(
    FN_ref_t *,
    void **iter_pos,
    const FN_ref_addr_t *);
extern int fn_ref_delete_addr(FN_ref_t *, void **iter_pos);
extern int fn_ref_delete_all(FN_ref_t *);

extern FN_ref_t *fn_ref_create_link(
    const FN_composite_name_t *link_name);
extern int fn_ref_is_link(const FN_ref_t *);
extern FN_composite_name_t *fn_ref_link_name(const FN_ref_t *link_ref);

extern FN_string_t *fn_ref_description(
```

```
    const FN_ref_t *,
    unsigned int detail,
    unsigned int *more_detail);
```

H.5.5 Operations on FN_attribute_t

```
extern FN_attribute_t *fn_attribute_create(
    const FN_identifier_t *attr_id,
    const FN_identifier_t *attr_syntax);
extern void fn_attribute_destroy(FN_attribute_t *);

extern FN_attribute_t *fn_attribute_copy(const FN_attribute_t *);
extern FN_attribute_t *fn_attribute_assign(
    FN_attribute_t *dst,
    const FN_attribute_t *src);

extern const FN_identifier_t *fn_attribute_identifier(
    const FN_attribute_t *);
extern const FN_identifier_t *fn_attribute_syntax(
    const FN_attribute_t *);
extern unsigned int fn_attribute_valuecount(const FN_attribute_t *);

extern const FN_attrvalue_t *fn_attribute_first(
    const FN_attribute_t *,
    void **iter_pos);
extern const FN_attrvalue_t *fn_attribute_next(
    const FN_attribute_t *,
    void **iter_pos);

extern int fn_attribute_add(
    FN_attribute_t *,
    const FN_attrvalue_t *,
    unsigned int exclusive);
extern int fn_attribute_remove(
    FN_attribute_t *,
    const FN_attrvalue_t *);
```

H.5.6 Operations on FN_attrset_t

```
extern FN_attrset_t *fn_attrset_create(void);
extern void fn_attrset_destroy(FN_attrset_t *);

extern FN_attrset_t *fn_attrset_copy(const FN_attrset_t *);
extern FN_attrset_t *fn_attrset_assign(
    FN_attrset_t *dst,
    const FN_attrset_t *src);

extern const FN_attribute_t *fn_attrset_get(
    const FN_attrset_t *,
    const FN_identifier_t *attr);
extern unsigned int fn_attrset_count(const FN_attrset_t *);

extern const FN_attribute_t *fn_attrset_first(
```

```
    const FN_attrset_t *,
    void **iter_pos);
extern const FN_attribute_t *fn_attrset_next(
    const FN_attrset_t *,
    void **iter_pos);

extern int fn_attrset_add(
    FN_attrset_t *,
    const FN_attribute_t *attr,
    unsigned int exclusive);
extern int fn_attrset_remove(
    FN_attrset_t *,
    const FN_identifier_t *attr_id);
```

H.5.7 Operations on FN_attrmodlist_t

```
extern FN_attrmodlist_t *fn_attrmodlist_create(void);
extern void fn_attrmodlist_destroy(FN_attrmodlist_t *);

extern FN_attrmodlist_t *fn_attrmodlist_copy(const FN_attrmodlist_t *);
extern FN_attrmodlist_t *fn_attrmodlist_assign(
    FN_attrmodlist_t *dst,
    const FN_attrmodlist_t *src);

extern unsigned int fn_attrmodlist_count(const FN_attrmodlist_t *);

extern const FN_attribute_t *fn_attrmodlist_first(
    const FN_attrmodlist_t *,
    void **iter_pos,
    unsigned int *first_mod_op);
extern const FN_attribute_t *fn_attrmodlist_next(
    const FN_attrmodlist_t *,
    void **iter_pos,
    unsigned int *mod_op);

extern int fn_attrmodlist_add(
    FN_attrmodlist_t *,
    unsigned int mod_op,
    const FN_attribute_t *mod_args);
```

H.5.8 Operations on FN_status_t

```
extern FN_status_t *fn_status_create(void);
extern void fn_status_destroy(FN_status_t *);
extern FN_status_t *fn_status_copy(const FN_status_t *);
extern FN_status_t *fn_status_assign(
    FN_status_t *dst,
    const FN_status_t *src);
extern unsigned int fn_status_code(const FN_status_t *);
extern const FN_composite_name_t *fn_status_remaining_name(
    const FN_status_t *);
extern const FN_composite_name_t *fn_status_resolved_name(
    const FN_status_t *);
```

```
extern const FN_ref_t *fn_status_resolved_ref(const FN_status_t *);
extern const FN_string_t* fn_status_diagnostic_message(
    const FN_status_t *);

extern unsigned int fn_status_link_code(const FN_status_t *);
extern const FN_composite_name_t *fn_status_link_remaining_name(
    const FN_status_t *);
extern const FN_composite_name_t *fn_status_link_resolved_name(
    const FN_status_t *);
extern const FN_ref_t *fn_status_link_resolved_ref(
    const FN_status_t *);
extern const FN_string_t* fn_status_link_diagnostic_message(
    const FN_status_t *);

extern int fn_status_is_success(const FN_status_t *);
extern int fn_status_set_success(FN_status_t *);
extern int fn_status_set(
    FN_status_t *,
    unsigned int code,
    const FN_ref_t *resolved_ref,
    const FN_composite_name_t *resolved_name,
    const FN_composite_name_t *remaining_name);
extern int fn_status_set_code(FN_status_t *, unsigned int code);
extern int fn_status_set_remaining_name(
    FN_status_t *,
    const FN_composite_name_t *);
extern int fn_status_set_resolved_name(
    FN_status_t *,
    const FN_composite_name_t *);
extern int fn_status_set_resolved_ref(FN_status_t *, const FN_ref_t *);
extern int fn_status_set_diagnostic_message(
    FN_status_t *,
    const FN_string_t *);

extern int fn_status_set_link_code(FN_status_t *, unsigned int code);
extern int fn_status_set_link_remaining_name(
    FN_status_t *,
    const FN_composite_name_t *);
extern int fn_status_set_link_resolved_name(
    FN_status_t *,
    const FN_composite_name_t *);
extern int fn_status_set_link_resolved_ref(
    FN_status_t *, const FN_ref_t *);
extern int fn_status_set_link_diagnostic_message(
    FN_status_t *,
    const FN_string_t *);

extern int fn_status_append_resolved_name(
    FN_status_t *,
    const FN_composite_name_t *);
extern int fn_status_append_remaining_name(
    FN_status_t *,
```

```
    const FN_composite_name_t *);

extern int fn_status_advance_by_name(
    FN_status_t *,
    const FN_composite_name_t *prefix,
    const FN_ref_t *resolved_ref);
extern FN_string_t *fn_status_description(
    const FN_status_t *,
    unsigned int detail,
    unsigned int *more_detail);
```

H.5.9 Operations on FN_search_control_t

```
extern FN_search_control_t *fn_search_control_create(
    unsigned int scope,
    unsigned int follow_links,
    unsigned int max_names,
    unsigned int return_ref,
    const FN_attrset_t *return_attr_ids,
    unsigned int *status);
extern void fn_search_control_destroy(
    FN_search_control_t *scontrol);
extern FN_search_control_t *fn_search_control_copy(
    const FN_search_control_t *scontrol);
extern FN_search_control_t *fn_search_control_assign(
    FN_search_control_t *dst,
    const FN_search_control_t *src);
extern unsigned int fn_search_control_scope(
    const FN_search_control_t *scontrol);
extern unsigned int fn_search_control_follow_links(
    const FN_search_control_t *scontrol);
extern unsigned int fn_search_control_max_names(
    const FN_search_control_t *scontrol);
extern unsigned int fn_search_control_return_ref(
    const FN_search_control_t *scontrol);
extern const FN_attrset_t *fn_search_control_return_attr_ids(
    const FN_search_control_t *scontrol);
```

H.5.10 Operations on FN_search_filter_t

```
extern FN_search_filter_t *fn_search_filter_create(
    unsigned int *status,
    const unsigned char *estr,
    ... );
extern void fn_search_filter_destroy(FN_search_filter_t *sfilter);
extern FN_search_filter_t *fn_search_filter_copy(
    const FN_search_filter_t *sfilter);
extern FN_search_filter_t *fn_search_filter_assign(
    FN_search_filter_t *dst,
    const FN_search_filter_t *src);
extern const unsigned char *fn_search_filter_expression(
    const FN_search_filter_t *sfilter);
extern const void **fn_search_filter_arguments(
```

```
    const FN_search_filter_t *sfilter,
    size_t *number_of_arguments);
```

H.5.11 Context Operations on FN_ctx_t

```
extern FN_ctx_t *fn_ctx_handle_from_initial(
    unsigned int authoritativeness,
    FN_status_t *status);
extern FN_ref_t *fn_ctx_lookup(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    FN_status_t *status);

extern FN_namelist_t *fn_ctx_list_names(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    FN_status_t *status);
extern FN_string_t *fn_namelist_next(
    FN_namelist_t *nl,
    FN_status_t *status);
extern void fn_namelist_destroy(
    FN_namelist_t *nl);

extern FN_bindinglist_t *fn_ctx_list_bindings(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    FN_status_t *status);
extern FN_string_t *fn_bindinglist_next(
    FN_bindinglist_t *bl,
    FN_ref_t **ref,
    FN_status_t *status);
extern void fn_bindinglist_destroy(
    FN_bindinglist_t *bl);

extern int fn_ctx_bind(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_ref_t *ref,
    unsigned int exclusive,
    FN_status_t *status);
extern int fn_ctx_unbind(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    FN_status_t *status);

extern FN_ref_t *fn_ctx_create_subcontext(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    FN_status_t *status);
extern int fn_ctx_destroy_subcontext(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    FN_status_t *status);
```

```
extern int fn_ctx_rename(
    FN_ctx_t *ctx,
    const FN_composite_name_t *oldname,
    const FN_composite_name_t *newname,
    unsigned int exclusive,
    FN_status_t *status);
extern FN_ref_t *fn_ctx_lookup_link(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    FN_status_t *status);
FN_composite_name_t *fn_ctx_equivalent_name(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_string_t *leading_name,
    FN_status_t *status);
extern FN_ref_t *fn_ctx_get_ref(
    const FN_ctx_t *ctx, FN_status_t *status);

extern FN_ctx_t *fn_ctx_handle_from_ref(const FN_ref_t *ref,
    unsigned int authoritativeness,
    FN_status_t *status);
extern void fn_ctx_handle_destroy(FN_ctx_t *ctx);
```

H.5.12 Attribute Operations on FN_ctx_t

```
extern FN_attribute_t *fn_attr_get(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_identifier_t *attr_id,
    unsigned int follow_link,
    FN_status_t *status);
extern int fn_attr_modify(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    unsigned int mod_op,
    const FN_attribute_t *attr,
    unsigned int follow_link,
    FN_status_t *status);

extern FN_valuelist_t *fn_attr_get_values(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_identifier_t *attr_id,
    unsigned int follow_link,
    FN_status_t *status);
extern FN_attrvalue_t *fn_valuelist_next(
    FN_valuelist_t *vl,
    FN_identifier_t **attr_syntax,
    FN_status_t *status);
extern void fn_valuelist_destroy(
    FN_valuelist_t *vl);

extern FN_attrset_t *fn_attr_get_ids(
```

```
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    unsigned int follow_link,
    FN_status_t *status);

extern FN_multigetlist_t *fn_attr_multi_get(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrset_t *attr_ids,
    unsigned int follow_link,
    FN_status_t *status);
extern FN_attribute_t *fn_multigetlist_next(
    FN_multigetlist_t *ml,
    FN_status_t *status);
extern void fn_multigetlist_destroy(
    FN_multigetlist_t *ml);

extern int fn_attr_multi_modify(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrmodlist_t *mods,
    unsigned int follow_link,
    FN_attrmodlist_t **unexecuted_mods,
    FN_status_t *status);

extern FN_attrset_t *fn_ctx_get_syntax_attrs(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    FN_status_t *status);
```

H.5.13 Extended Attribute Operations on FN_ctx_t

```
extern int fn_attr_bind(
    FN_ctx_t *ctx,
    const    FN_composite_name_t *name,
    const    FN_ref_t *ref,
    const    FN_attrset_t *attrs,
    unsigned int exclusive,
    FN_status_t *status);

extern FN_ref_t *fn_attr_create_subcontext(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrset_t *attrs,
    FN_status_t *status);

extern FN_searchlist_t *fn_attr_search(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_attrset_t *match_attrs,
    unsigned int return_ref,
    const FN_attrset_t *return_attr_ids,
    FN_status_t *status);
```

```
extern FN_string_t *fn_searchlist_next(
    FN_searchlist_t *sl,
    FN_ref_t **returned_ref,
    FN_attrset_t **returned_attrs,
    FN_status_t *status);
extern void fn_searchlist_destroy(
    FN_searchlist_t *sl);

extern FN_ext_searchlist_t *fn_attr_ext_search(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,
    const FN_search_control_t *control,
    const FN_search_filter_t *filter,
    FN_status_t *status);
extern FN_composite_name_t *fn_ext_searchlist_next(
    FN_ext_searchlist_t *esl,
    FN_ref_t **returned_ref,
    FN_attrset_t **returned_attrs,
    FN_status_t *status);
extern void fn_ext_searchlist_destroy(
    FN_ext_searchlist_t *esl);
```

H.5.14 Operations on FN_compound_name_t

```
extern FN_compound_name_t *fn_compound_name_from_syntax_attrs(
    const FN_attrset_t *,
    const FN_string_t *name,
    FN_status_t *);
extern FN_attrset_t *fn_compound_name_get_syntax_attrs(
    const FN_compound_name_t *);
extern void fn_compound_name_destroy(FN_compound_name_t *);

extern FN_string_t *fn_string_from_compound_name(
    const FN_compound_name_t *);
extern FN_compound_name_t *fn_compound_name_copy(
    const FN_compound_name_t *);
extern FN_compound_name_t *fn_compound_name_assign(
    FN_compound_name_t *dst,
    const FN_compound_name_t *src);
extern unsigned int fn_compound_name_count(const FN_compound_name_t *);

extern const FN_string_t *fn_compound_name_first(
    const FN_compound_name_t *,
    void **iter_pos);
extern const FN_string_t *fn_compound_name_next(
    const FN_compound_name_t*,
    void **iter_pos);
extern const FN_string_t *fn_compound_name_prev(
    const FN_compound_name_t *,
    void **iter_pos);
extern const FN_string_t *fn_compound_name_last(
    const FN_compound_name_t *,
    void **iter_pos);
```

```
extern FN_compound_name_t *fn_compound_name_prefix(
    const FN_compound_name_t *,
    const void *iter_pos);
extern FN_compound_name_t *fn_compound_name_suffix(
    const FN_compound_name_t *,
    const void *iter_pos);

extern int fn_compound_name_is_empty(const FN_compound_name_t *);

extern int fn_compound_name_is_equal(
    const FN_compound_name_t *,
    const FN_compound_name_t *,
    unsigned int *status);
extern int fn_compound_name_is_prefix(
    const FN_compound_name_t *,
    const FN_compound_name_t *prefix,
    void **iter_pos,
    unsigned int *status);
extern int fn_compound_name_is_suffix(
    const FN_compound_name_t *,
    const FN_compound_name_t *suffix,
    void **iter_pos,
    unsigned int *status);

extern int fn_compound_name_prepend_comp(
    FN_compound_name_t *,
    const FN_string_t *,
    unsigned int *status);
extern int fn_compound_name_append_comp(
    FN_compound_name_t *,
    const FN_string_t *,
    unsigned int *status);

extern int fn_compound_name_insert_comp(
    FN_compound_name_t *,
    void **iter_pos,
    const FN_string_t *,
    unsigned int *status);
extern int fn_compound_name_delete_comp(
    FN_compound_name_t *,
    void **iter_pos);

extern int fn_compound_name_delete_all(FN_compound_name_t *);
```

Glossary

address
An unambiguous name, label or number which identifies the location of a particular entity or service. See also presentation address.

API
Application Programming Interface.

argument
Information which is passed to a function or operation and which specifies the details of the processing to be performed.

atom
An atom is a unique ID corresponding to a string name. Atoms are used to identify properties, types and selections.

BER
Basic Encoding Rules.

cache
Either a copy of an entry stored in other DSA(s) through bilateral agreement, or a locally and dynamically stored copy of an entry resulting from a request (a cache copy).

CDS
DCE Cell Directory Service: a naming system in the enterprise namespace.

composite name
A name that spans multiple naming systems.

Directory
A collection of open systems which cooperate to hold a logical database of information about a set of objects in the real world.

DNS
Domain Name Service. A name and address lookup protocol used by the Internet.

federated naming
An aggregation of autonomous naming systems that cooperate to support name resolution of composite names through a standard interface.

Function
A programming language construct, modelled after the mathematical concept. A function encapsulates some behaviour. It is given some arguments as input, performs some processing, and returns some results. Also known as procedures, subprograms or subroutines. See **Operation**.

internationalisation
The provision within a computer program of the capability of making itself adaptable to the requirements of different native languages, local customs and coded character sets.

name
A construct that singles out a particular (directory) object from all other objects. A name must be unambiguous (that is, denote just one object), however, it need not be unique (that is, be the only name which unambiguously denotes the object). Network Information System, in Solaris.

nns
Next Naming System: subordinate nasming systems are federated through *nns* pointers.

Operation
Processing performed within the directory to provide a service, such as a read operation. It is given some arguments as input, performs some processing, and returns some results. An application process invokes an operation by calling an interface **Function**.

service
Software that implements the interface.

Index