Federated Naming Service Programming Guide

Adobe PostScript™

Contents

Preface

The Federated Naming Service (FNS) is a set of application programming interfaces and policies that allow applications to use a common set of names and policies over different name services.

FNS is implemented on top of NIS+ and allows you to use a set of common names with desktop applications. Sun's implementation of FNS conforms to the X/Open™ federated naming (XFN) specification.

Who Should Use This Book

The primary audience of *Federated Naming Service Guide* is software developers who write distributed applications. Use of this guide assumes basic competence in programming, a working familiarity with the C programming language, and a working familiarity with the UNIX® operating system. Developers should read all four parts of this manual.

System and network administrators should look at *System Administration Guide: Naming and Directory Services (DNS, NIS, and LDAP)* for FNS setup and configuration, as well as administratin information. This manual does not cover NIS+ or the Domain Name System (DNS) except as they relate to FNS. For information on all other Solaris operating environment naming and directory services, please refer to the *System Administration Guide: Naming and Directory Services (DNS, NIS, and LDAP)*

How This Book Is Organized

Chapter 1is a high-level overview of FNS and the problems it addresses.

Chapter 2describes the client programming interfaces for X/Open Federated Naming (XFN).

Chapter 3presents three self-contained executable programs: a namespace browser, a printer client and server, and a tool to populate attributes of users.

Appendixes

Appendix Adescribes the XFN composite name string syntax and the resolution techniques for composite names.

Appendix Bgives supplemental information about composite name syntax.

Related Books

With the exception of the XFN specification, the following books do not specifically cover FNS but they provide a good background on how name services work in client-server computing:

- Raman Khanna. *Distributed Computing—Implementation and Strategy*. Prentice Hall, 1993
- Sape J. Mullender (editor) . *Distributed Systems*. ACM Press, 1990
- P. Albitz and C. Liu. *DNS and BIND*. O'Reilly, 1992
- *Managing the X.500 Client Toolkit*. SunSoft Inc., 1995
- *X/Open Preliminary Specifications, Federated Naming: The XFN Specifications*. X/Open Document #P403, ISBN: 1-85912-045-8. X/Open, July 1994

You may also want to reference the following AnswerBook® on-line documentation:

- *Solaris Reference Manual Collection*
- *Solaris Software Developer Collection*
- *System Administration Guide, Volume I*
- *System Administration Guide, Volume II*

Ordering Sun Documents

Fatbrain.com, an Internet professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Conventions

The following table describes the typographic changes used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
AaBbCc123	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
AaBbCc123	What you type, contrasted with on-screen computer output	`machine_name% `**`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type **rm** *filename*.

TABLE P–1 Typographic Conventions　　*(Continued)*

Typeface or Symbol	Meaning	Example
AaBbCc123	Book titles, new words, or terms, or words to be emphasized.	Read Chapter 6 in *User's Guide*. These are called *class* options. You must be *root* to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`

Introduction to the Federated Naming Service (FNS)

This chapter is an overview of the Federated Naming Service (FNS).

What is Federated Naming?

Federated Naming Service provides a method for hooking up, or federating, multiple naming services under a single, simple uniform interface for the basic naming and directory operations. The service supports resolution of composite names—names that span multiple naming systems—through the naming interface. Each member of a federation has autonomy in its choice of naming conventions, administrative interfaces, and its particular set of operations, other than name resolution.

In the Solaris operating environment, the FNS implementation consists of a set of enterprise-level naming services with specific policies and conventions for naming organizations, users, hosts, sites, and services, as well as support for global naming services such as DNS and X.500. More specifically, FNS has support for:

- Enterprise-level naming services: NIS+, NIS and files
- Global-level naming services: DNS, and X.500 (over LDAP or DAP). See *System Administration Guide: Naming and Directory Services (DNS, NIS, and LDAP)*for information on DNS Text records and X.500 attribute syntax for XFN references
- Application-specific namespaces: file naming, printer naming
- Generic application namespaces for other applications

What Is XFN?

XFN stands for X/Open Federated Naming. XFN is a standard that is actively supported by organizations such as Sun, IBM, Hewlett-Packard, DEC, Siemens, and OSF. The programming interfaces and policies that FNS supports are specified by XFN. An overview of XFN concepts is presented later in this chapter; Chapter 2 describes the XFN programming interface in detail.

Note – In a 64–bit XFN application, the X.500 directory service is not supported.

FNS is compliant with the *X/Open CAE Specification for Federated Naming* (July 1995). Applications that use FNS are portable across platforms because the interface exported by FNS is XFN, a public, open interface endorsed by other vendors and X/Open. X/Open Co. Ltd. is part of the Open Group, which is an international standards organization committed to defining computing standards that are endorsed and adhered to by major computer vendors.

Why FNS?

FNS is useful for the following reasons:

- A single uniform naming interface is provided to clients for accessing different naming services. As a consequence, the addition of new naming services does not require changes to applications or to existing member naming services. Furthermore, application developers need to learn and use only one naming interface.

- Names can be composed in a uniform way, and the resulting composite names can have any number of components. This allows the composite namespace to serve the needs of diverse applications.

- Coherent naming is encouraged through the use of shared contexts and shared names. This reduces duplication of effort in individual applications when supplying similar functionality.

FNS Policies

FNS provides applications with a set of policies on how namespaces are arranged and used. These policies specify:

- The namespaces for enterprise objects: organizations, hosts, users, sites, and services. (These naming services support contexts that allow other objects to be named relative to these objects.)

- The relationships between the organization, host, user, site, and service namespaces, and the names used to refer to these namespaces

- The syntax of names in these namespaces

- How to federate the enterprise namespace so that it is accessible in the global namespace

- Names and bindings present in the initial context of every process

Table 1–1 is a summary of FNS policy for arranging the enterprise namespace and Figure 1–1shows that FNS policies provides a common framework for the three levels of service: global, enterprise, and application.

TABLE 1–1 Policies for the Federated Enterprise Namespace

Namespace Identifiers	Name Service Type	Subordinate Context	Parent Context	Namespace Organization	Syntax
orgunit _orgunit	Organizational unit	Site, user, host, file system, service	Enterprise root	Hierarchical	NIS+ domain name Dot-separated, right-to-left
site _site	Site	Service, file system	Enterprise root, organizational unit	Hierarchical	Dot-separated, right-to-left
user _user	User	Service, file system	Enterprise root, organizational unit	Flat	Solaris login name
host _host	Host	Service, file system	Enterprise root, organizational unit	Flat	Solaris host name
service _service	Service	Application-specific	Enterprise root, organizational unit, site, user, host	Hierarchical	/ separated, left-to-right
fs _fs	File system	None	Enterprise root, organizational unit, site, user, host	Hierarchical	/ separated, left-to-right
printer	Printer	None	Service	Hierarchical	/ separated, left-to-right

What FNS Policies Do Not Specify

The FNS policies do not specify the specific names used within naming services. In addition, naming within the application is the responsibility of individual applications

or groups of related applications. They also do not specify the attributes to use after
the object has been named.

Global
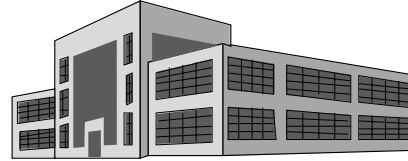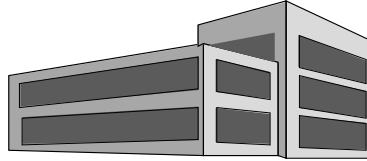
Enterprise
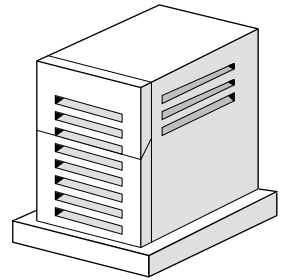
Application
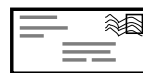
FIGURE 1–1 Different Levels of Naming Services

What FNS Enterprise Policies Arrange

The FNS enterprise policies deal with the arrangement of objects within the enterprise namespace. The policies are summarized in Table 1–1.

- Organization – Entities such as departments, centers, and divisions. Sites, hosts, users, and services can be named relative to an organization. The XFN term for organization is *organizational unit*.

- Site – Physical locations, such as buildings, machines in buildings, and conference rooms within buildings. Sites can have files and services associated with them.

- Host – Computers. Hosts can have files and services associated with them.

- User – Human users. Users can have files and services associated with them.

- Service – Services such as printers, faxes, mail, and electronic calendars.

- File – Files within a file system.
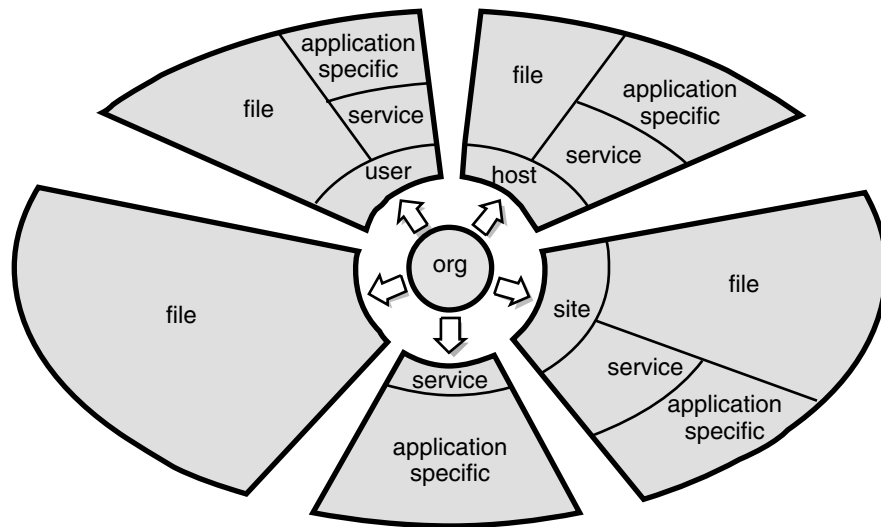
FIGURE 1–2 What FNS Policies Arrange

The namespace of an enterprise is structured around the hierarchical structure of organizational units of an enterprise. Names of sites, hosts, users, files, and services can be named relative to names of organizational units by composing the organizational unit name with the appropriate namespace identifier and object name.

In Figure 1–3, a user, jsmith in the engineering organization of an enterprise, is named using the name orgunit/desktop.sw.eng/user/jsmith
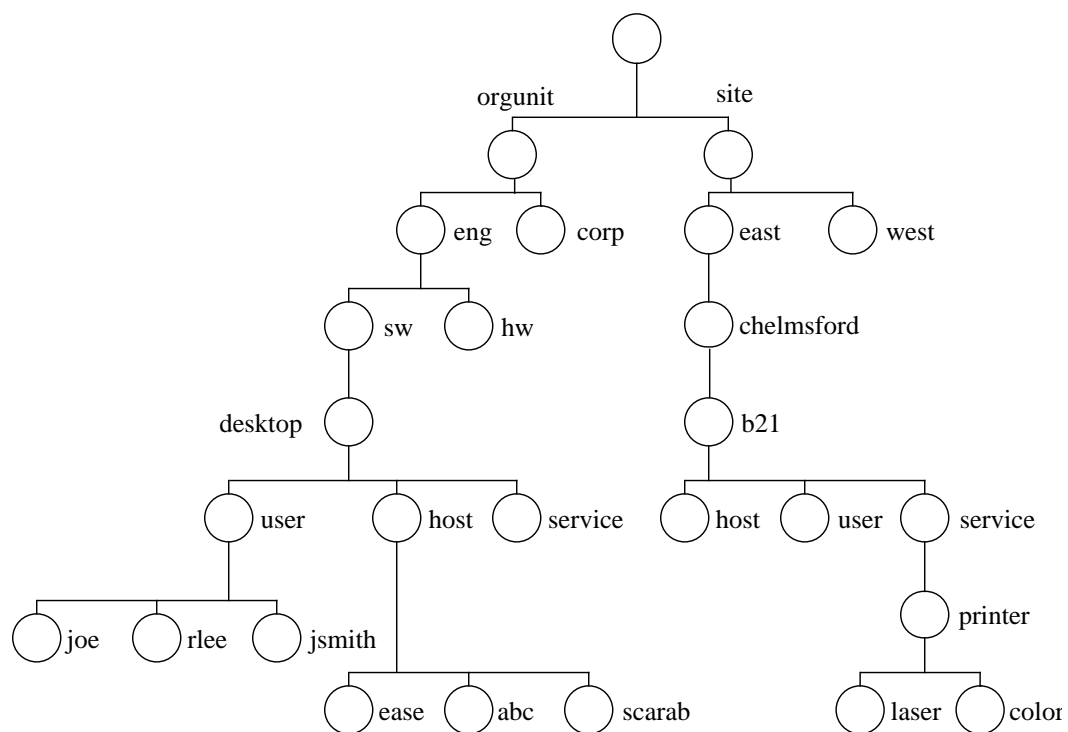
FIGURE 1–3 Example of an Enterprise Namespace

Initial Context Bindings

Resolution of a name in XFN always begins with some context. XFN defines an *initial context* as a starting point for name resolution. The initial context contains bindings that allow the client application to (eventually) name any object in the enterprise namespace. Figure 1–4 shows the same naming system as the one shown in Figure 1–3, except that the initial context bindings are shaded and shown in italics.
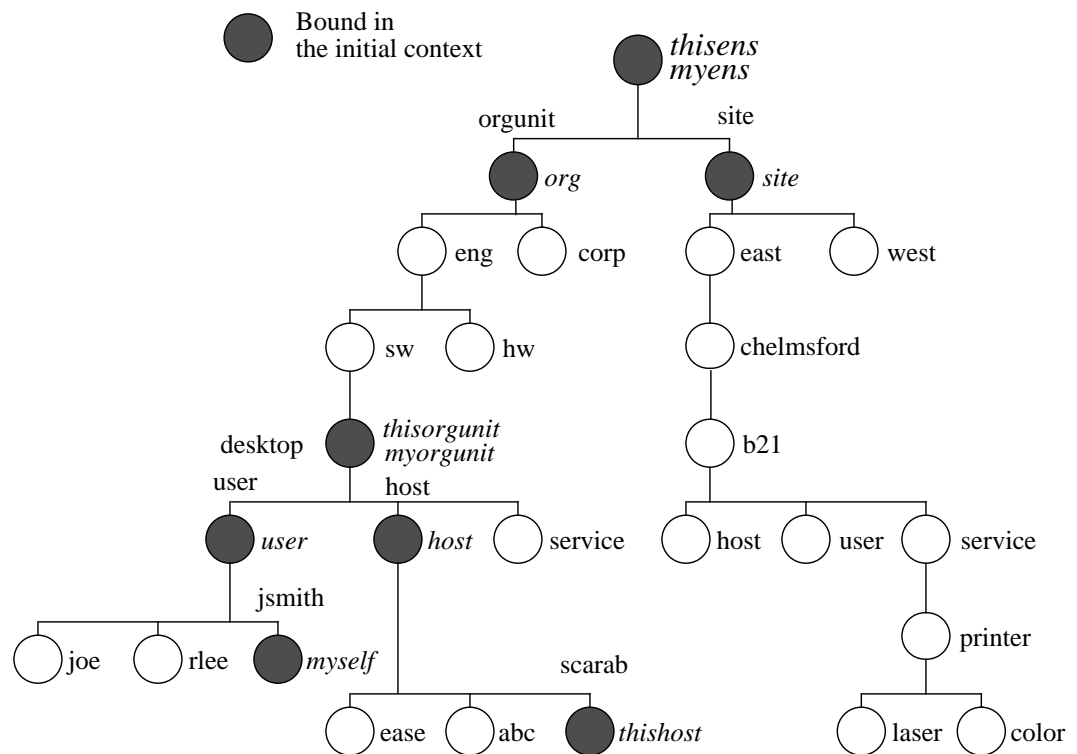
FIGURE 1–4 Example of Enterprise Bindings in the Initial Context

The initial context has a flat namespace for namespace identifiers. The bindings of these namespace identifiers are summarized in Table 1–2. The categories of bindings are:

- User-related bindings
- Host-related bindings
- "Shorthand" bindings

In Table 1–2, the user to which the bindings are related is denoted by U, and the host to which the bindings are related is denoted by H. Not all of these names need to appear in all initial contexts. For example, when a program is invoked by the superuser, none of the user-related bindings appears in the initial context.

TABLE 1–2 Initial Context Bindings for Naming Within the Enterprise

Namespace Identifier	Binding
myself _myself thisuser	U's user context
myens _myens	The enterprise root of U
myorgunit _myorgunit	U's distinguished organizational unit context. For NIS+, the distinguished organizational unit is U's NIS+ home domain. For NIS and files, it is the current domain and system, respectively.
thishost _thishost	H's host context
thisens _thisens	The enterprise root of H
thisorgunit _thisorgunit	H's distinguished organizational unit context. For NIS+, the distinguished organizational unit is H's NIS+ home domain. For NIS and files, it is the current domain and system, respectively.
user _user	The context in which users in the same organizational unit as H are named
host _host	The context in which hosts in the same organizational unit as H are named
org orgunit _orgunit	The root context of the organizational unit namespace in H's enterprise. For NIS+, this corresponds to the NIS+ root domain. For NIS and files, it is the current domain and system, respectively.
site _site	The root context of the site namespace at the top organizational unit if the site namespace has been configured
... /...	Global context for resolving DNS or X.500 names
_dns	Global context for resolving DNS names
_x500	Global context for resolving X.500 names

Examples of Composite Names

This section shows examples of names that follow FNS policies. The specific choices of organization names, site names, user names, host names, file names, and service names (such as "calendar" and "printer") are illustrative only; these names are not specified by FNS policy.

Composing Names Relative to Organizations

The naming systems to be found under an organization are: `user`, `host`, `service`, `fs`, and `site`.

`org/csl.parc/site/videoconference.northwing`
   names a conference room `videoconference` located in the north wing of the site associated with the organization `csl.parc`.

`org/csl.parc/user/mjones`
   names a user `mjones` in the organization `csl.parc`.

`org/csl.parc/host/inmail`
   names a machine `inmail` belonging to the organization `csl.parc`.

`org/csl.parc/fs/pub/blue-and-whites/CSL92-124`
   names a file `pub/blue-and-whites/CSL92-124` belonging to the organization `csl.parc`.

`org/csl.parc/service/calendar`
   names the `calendar` service for the organization `csl.parc`. This service might manage the meeting schedules for the organization.

Composing Names Relative to Users

The naming systems associated with users are `service` and `fs`.

`user/jsmith/service/calendar`
   names the `calendar` service of the user `jsmith`.

`user/jsmith/fs/bin/games/riddles`
   names the file `bin/games/riddles` under the home directory of the user `jsmith`.

Composing Names Relative to Hosts

The naming systems associated with hosts are `service` and `fs`.

`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 file system exported by the machine `mailhop`.

Composing Names Relative to Sites

The naming systems associated with sites are `service` and `fs`.

```
site/B5.MountainView/service/printer/speedy
```
names a printer `speedy` in the `B5.MountainView` site.

```
site/B5.MountainView/fs/usr/dist
```
names a file directory `usr/dist` available in the `B5.MountainView` site.

XFN Overview

The following gives an overview of the main concepts in XFN and they are used in defining a federated naming system.

XFN References

An XFN name is bound to a reference, which is the information on how to reach an object. It contains a list of addresses, which identify communication endpoints on how to reach the object. Multiple addresses identify multiple communication endpoints for a single conceptual object or service. For example, a list of addresses might be required because the object is distributed or because the object can be accessed through more than one communication mechanism.

XFN Contexts

An XFN context is an object that exports the XFN base context programming interface. A context contains a list of atomic names bound to references, as shown in Figure 1–5. An atomic name can have zero or more attributes. Contexts are at the heart of the lookup and binding operations, described extensively in Chapter 2.
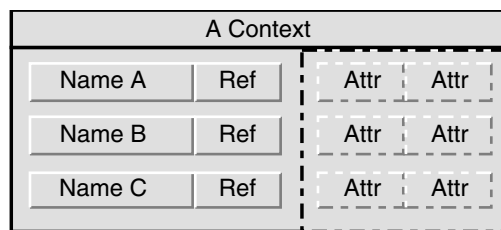
FIGURE 1–5 An XFN Context

XFN Attributes

In addition to references, there can be zero or more attributes associated with each named object, as shown in Figure 1–5. Each attribute has a unique attribute identifier, an attribute syntax, and a set of zero or more distinct attribute values.

XFN defines operations for examining and modifying the values of attributes associated, as well as searching for objects using their associated attributes.

XFN Compound Names

An XFN compound name is a sequence of one or more atomic 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. Objects in the subcontext are named using a compound name. Compound names are resolved by looking up each successive atomic name in each successive context.

A familiar analogy for UNIX users is the file naming model, where directories are analogous to contexts, and path names serve as compound names. Furthermore, contexts can be arranged in a "tree" structure, just as directories are, with the compound names forming a hierarchical namespace.

- UNIX example: `usr/local/bin`. UNIX atomic names are ordered from left to right and are delimited by slash (/) characters. The name `usr` is bound to a context in which `local` is bound. The name `local` is bound to a context in which `bin` is bound.

- DNS example: `sales.Wiz.COM`. DNS atomic names are ordered from right to left, and are delimited by dot (.) characters. The domain name `COM` is bound to a context in which `Wiz` is bound. `Wiz` is bound to a context in which `sales` is bound.

- X.500 example: `c=us/o=wiz/ou=sales`. An X.500 atomic name contains an attribute type and an attribute value. Atomic names are known as *relative*

distinguished names in X.500. In this string representation, X.500 atomic names are ordered from left to right, and are delimited by slash (/) characters. An attribute type is separated from an attribute value by an equal sign (=) character. Abbreviations are defined for commonly used attribute types (for example, "c" represents country name). The country name US is bound to a context in which wiz is bound. The organization name wiz is bound to a context in which the organizational unit name sales is bound.

Note – In a 64–bit XFN application, the X.500 directory service is not supported.

Figure 1–6shows an example of a hierarchical naming system with compound names.
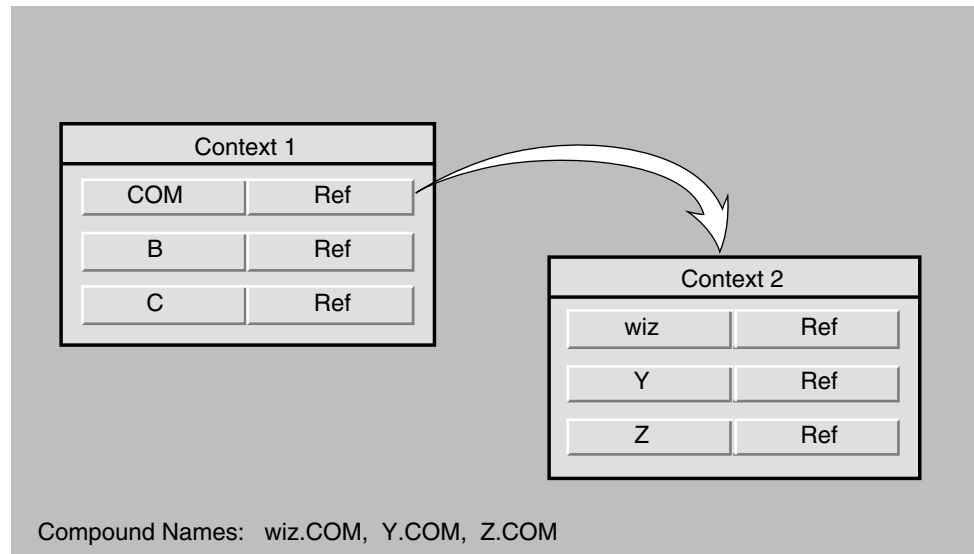
FIGURE 1–6 Hierarchical Naming System With Compound Names

XFN Composite Names

An XFN 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 name from the namespace of a single naming system. Composite name resolution is the process of resolving a name that spans multiple naming systems. Appendix A, and Appendix B, supply more detail about composite names.

Components are slash-separated (/) and ordered from left to right, according to XFN composite name syntax. For example, the composite name

```
sales.Wiz.COM/usr/local/bin
```

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

Figure 1–7 shows an example of a federated naming system with composite names.
The position of the name within a context has no inherent significance in this
illustration.

XFN Links

An XFN link is a special form of reference that is bound to an atomic name in a
context. Instead of an address, a link contains a composite name. Many naming
systems support a native notion of link that can be used within the naming system
itself. XFN does not specify whether there is any relationship between such native
links and XFN links.

"XFN Links" on page 89describes links in detail.
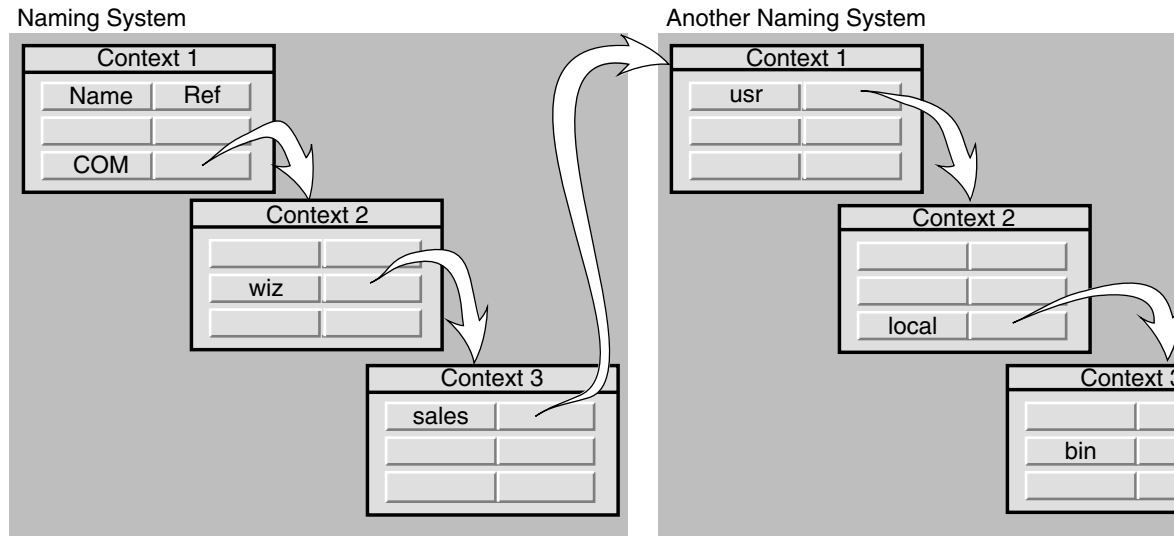
Composite name: sales.wiz.com, usr/local/bin

FIGURE 1–7 Federated Naming System With Composite Names

XFN Initial Context

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

The policies described in*System Administration Guide: Naming and Directory Services (DNS, NIS, and LDAP)* specify a set of names that the client can expect to find in this context and the semantics of their bindings. This provides the initial pathway to other XFN contexts.

API Usage Model

Many clients of the XFN interface are only interested in lookups. Their usage of the interface amounts to:

- Obtaining the initial context
- Looking up one or more names relative to the initial context

After the client obtains a desired reference from the lookup operation, it constructs a client-side representation of the object from the reference. This need not be code within the application layer but can be code inside the service 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 receiving 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 FNS policies described earlier further encourage a bind/lookup model for how services and clients can rendezvous through the use of the naming service.

FNS and Applications

Applications that are aware of FNS can expect the namespace to be arranged according to the FNS policies, and applications that bind names in the FNS namespace are expected to follow these policies.

Applications use FNS in the following ways:

1. *Applications can use FNS through existing interfaces*. A significant proportion of FNS use is through existing application programming interfaces. For example, consider a UNIX application that obtains a file name that it later supplies to the UNIX `open()` function. With FNS support for resolution of file names, the application need not be aware that the strings it deals with are composite names rather than the traditional local path names. Many applications can thereby support the use of composite names without modification.

2. *Applications can be direct clients of the XFN interface and policies*. Application-level utilities, such as the file system, the printing service, and the desktop tools (calendar manager, file manager) are examples of clients that use the XFN interface directly.

3. *Systems can export the XFN interface*. Naming systems, such as DNS and X.500, and naming systems embedded in other services, like the file system and printing service, in combination with XFN, are examples of naming systems that export the XFN interface.

This book addresses the needs of applications that use the XFN interface. Some examples of these applications are given in the next chapter.

Application Interaction with XFN

The way that client applications interact with XFN to access different naming systems is illustrated in a series of figures.Figure 1–8 shows an application that uses the XFN API and library.
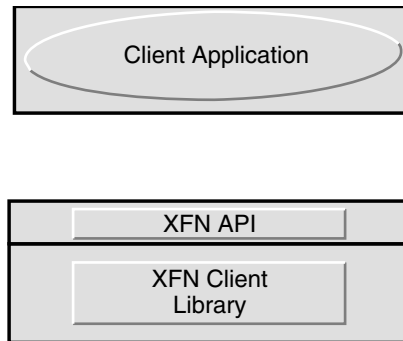
FIGURE 1–8 Client Application Interaction With XFN

Figure 1–9 shows the details beneath the API. A naming service that is federated is accessed through the XFN client library and a *context shared object module*. This module translates the XFN calls into naming service–specific calls.
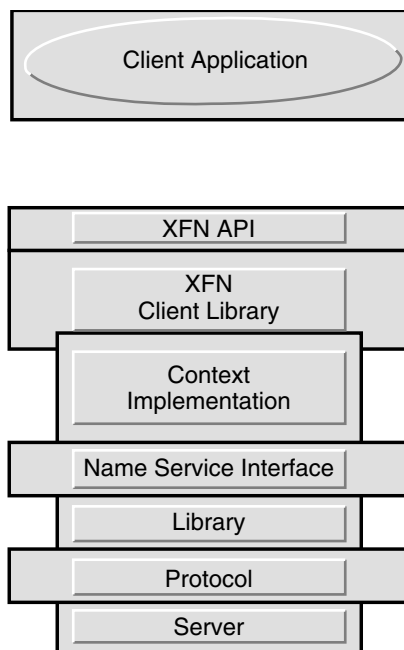
FIGURE 1–9 Details Beneath XFN API

X.500, DNS, and NIS+ are the naming services that have been federated in the example shown in Figure 1–10.

As resolution of a composite name proceeds, it can cause these different modules to be linked in, depending on the types of contexts referenced in the name.

Composite names

Client Application

XFN API

XFN Client Library

XDS API	Resolver API	NIS+ API
DUA	libresolv	libnsl/nis
X.500 DAP	RFC 1035	NIS+ protocol
DSA	DNS server	NIS+ server

Context shared object
modules for specific
name services
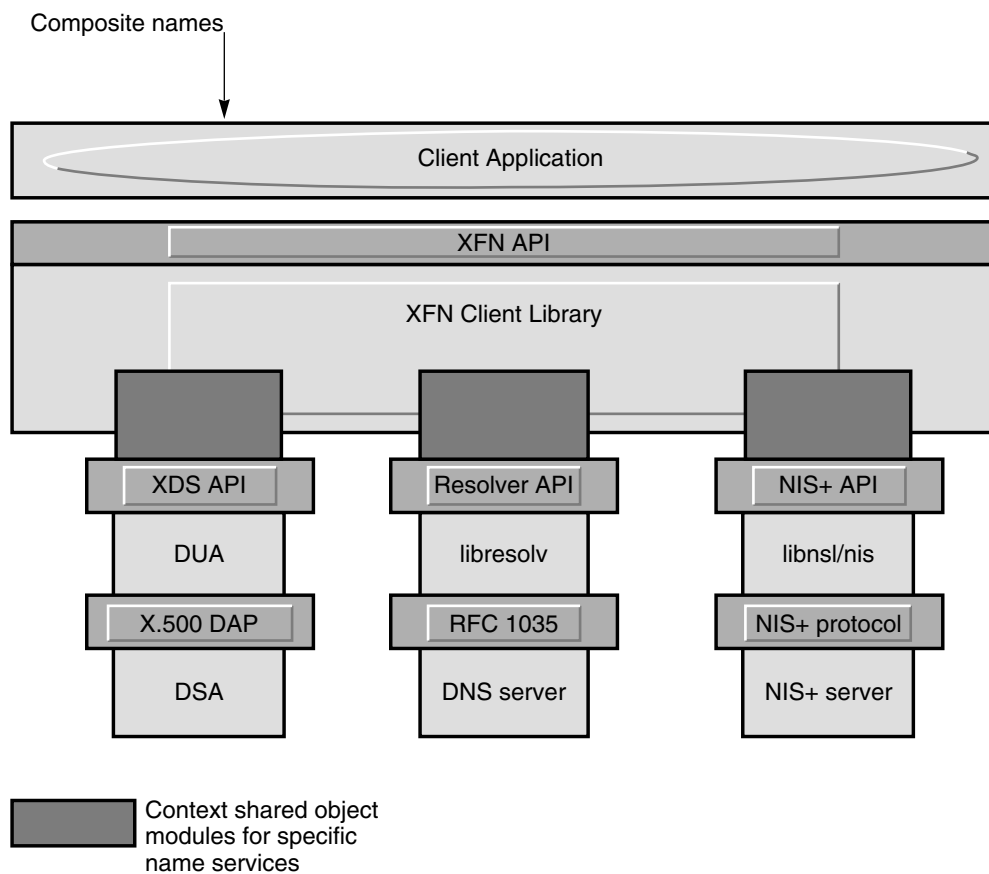
FIGURE 1–10 XFN Implementation Examples

Interfaces for Writing XFN Applications

This chapter describes the client programming interfaces for XFN. Additional information on the XFN interfaces is available in the man pages.

XFN Interface Overview

The XFN client interface consists of the base context interface, the base attribute interface, the extended attribute interface, and some supporting interfaces.

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

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

The extended attribute interface provides operations to search and create objects in the namespace with attributes.

The supporting interfaces contain:

- Operations on the status object and status codes used in the context and attribute operations
- Abstract data types that represent objects passed to and returned from the context and attribute operations, such as composite names, references, and attributes
- A standard model and operations for parsing compound names whose syntax is specific to a naming system. These are of interest primarily to service implementers
- Operations for manipulating objects that are used to specify the criteria of extended search operations

"API Usage Model" on page 25summarizes how an application typically uses the programming interface.

Interface Conventions

The XFN interface is presented in ISO standard C, which is equivalent to ANSI standard C. The symbols defined by the interface are prefixed by fn or FN, for *federated naming*.

- The FN_ prefix is used for both data types and predefined constants.

  In addition, data types have a _t suffix, such as FN_ref_t. Predefined constants appear in all–uppercase characters, such as FN_ID_STRING.

- The fn_ prefix is used for function names. Names of functions in the base context interface have the prefix fn_ctx_, such as fn_ctx_lookup. Names of functions in the base attribute interface have the prefix fn_attr_, such as fn_attr_get.

Usage

Include the XFN header file in your code.

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

Include the XFN library when you compile.

```
cc -o program_name file1.c file2.c -lxfn
```

Abstract Data Types

Except for FN_attrvalue_t and FN_identifier_t, the types defined in the interface hide their actual data representation from the client. The client performs every operation on an object of one of these types through a well-defined interface for that data type.

When the client accesses these objects, the client refers to the objects solely through a *handle* to an object. Operations are provided to create objects of each type and to destroy them. The creation operation returns a handle to the new object. The destroy operation releases all resources associated with the object.

The only information about this handle revealed to the client is that it is a pointer type. The client cannot assume what this handle points to. In particular, the handle might not point directly to the memory containing the object's actual state.

The value 0 is defined for all pointer types. The functions that return handles in the interface return the value 0 as an indication of failure. The values 0 and NULL are equivalent.

Memory–Management Policies

The following memory–management policies are used for all client interfaces described in this chapter:

- When a function returns a non-`const` pointer to an object, the client "owns" the object. The client can alter the object and is responsible for freeing the space allocated to it when the object is no longer required.

- When a function returns a `const` pointer to an object, the service "owns" the object. The client must neither modify the object in any way, nor free the space allocated to it. If the client needs to control a copy, it must make one for itself.

- When a function takes a non-`const` parameter that is passed by reference, the service "borrows" the object during the function's execution. It can modify the object during this period, but it does not retain any reference to the object beyond this period.

- When a function takes a `const` parameter that is passed by reference, the service reads but does not modify the object. The service does not keep any reference to the object beyond the period of the function's execution.

The Base Context Interface

This section describes the operations in the base context interface. The interfaces to the objects used in the operations are described in "Parameters Used in the Interface" on page 55.

- "Names in Context Operations" on page 34
- "Requirements for Supporting the Context Operations" on page 34
- "Status Objects" on page 35

- "Getting Context Handles" on page 35
- "Lookup and List Contexts" on page 36
- "Updating Bindings" on page 38
- "Managing Contexts" on page 39

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 might 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:

- Doing a lookup on that portion of the name that represents the target context
- 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*, which are defined in "XFN Links" on page 89.

Requirements for Supporting the Context Operations

The lookup operation `fn_ctx_lookup()` must be supported by all contexts. When contexts do not support other operations, they can return an `FN_E_OPERATION_NOT_SUPPORTED` status code (see codes in Table 2–3).

XFN contexts are required to support the resolution phase of every operation in the base context and attribute interface 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, not all contexts need to allow binding and listing names. However, all contexts must fully support the resolution phase of these operations.

Composite names are 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:

- Determining which set of leading components it must resolve
- Resolving that portion to a reference

- Returning a status object containing this reference and the portion of the name unresolved

Composite name resolution is further discussed in "Composite Name Resolution" on page 87.

Status Objects

In each context operation, the caller supplies an `FN_status_t` parameter. The called function sets this status object as described in "Status Objects and Status Codes" on page 51. All status objects are handled in this way for each operation in the base context interface (this is not restated in the individual operation descriptions).

Getting Context Handles

All operations on a context require a context handle. There are several ways of obtaining a context handle. If you have a reference, you can use it to construct a context handle. Otherwise, to get a handle to the initial context, you must call `fn_ctx_handle_from_initial()`.

```
fn_ctx_handle_from_initial
fn_ctx_handle_from_ref
```

Construct Handle to Initial Context

This operation returns a handle to the callers initial context. On successful return, the context handle points to a context containing the bindings described in "Initial Context Bindings" on page 17.

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

`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 can 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. In some, the authoritative source is a single "master" servers, while in others, the authoritative source is a quorum of servers.

Construct Context Handle From Reference

This operation returns a handle to an `FN_ctx_t` object given a reference, *ref*, for that context.

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

`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 this context handle can 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. In some, the authoritative source is a single "master" server, while in others, the authoritative source is a quorum of servers.

Lookup and List Contexts

```
fn_ctx_lookup
fn_ctx_list_names
fn_namelist_next
fn_namelist_destroy
fn_ctx_list_bindings
fn_bindinglist_next
fn_bindlist_destroy
fn_ctx_lookup_link
```

Lookup

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

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

List Names

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

```
FN_nameslist_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);
```

The call to `fn_ctx_list_names()` initiates the enumeration process for the target context. It returns an `FN_nameslist_t` object that you can use for the enumeration.

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 marker. Successive calls to `fn_namelist_next()` using *nl* return successive names 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 call can be invoked at any time to terminate the enumeration.

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

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

List Bindings

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

```
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);
```

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.

Lookup Link

This operation returns the XFN link bound to *name*. The terminal atomic part of *name* must be bound to an XFN link.

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

The normal `fn_ctx_lookup()` operation follows all XFN links encountered, including any that are 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 last link is not followed, and the operation returns the link.

Updating Bindings

```
fn_ctx_bind
fn_ctx_unbind
fn_ctx_rename
```

Bindings can be added, overwritten, removed, or renamed.

Bind

This operation binds the supplied reference *ref* to the supplied composite name *name*, taken 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.

```
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);
```

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.

The value of *ref* cannot be NULL. If you want to reserve a name using the `fn_ctx_bind()` operation, bind a reference containing no address. This reference can be naming service-specific or it can be the conventional NULL reference.

Unbind

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

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

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()` operations are idempotent.

Rename

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

```
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);
```

If `exclusive` is zero, this operation overwrites any old binding of *newname*. If `exclusive` is nonzero, 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. 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 an atomic name.

Managing Contexts

```
fn_ctx_create_subcontext
fn_ctx_destroy_subcontext
fn_ctx_get_ref
fn_ctx_get_syntax_attrs
fn_ctx_handle_destroy
```

```
fn_ctx_equivalent_name
```

Contexts can be created, destroyed,and referenced.

Create Subcontext

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

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

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 exports the context interface and is created in the same naming system as the target context. XFN does not specify any further properties of the new subcontext. Other properties of the subcontext are determined by the target context and its naming system.

Destroy Subcontext

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

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

As with 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 part of *name*.

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

Get Reference to Context

This operation returns a reference to the supplied context object.

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

Get Syntax Attributes of Context

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

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

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 might not be associated with the context; they might be associated with the reference of the context, rather than with the context itself (see "Relationship to Naming Operations" on page 43).

Destroy Context Handler

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.

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

Construct an Equivalent Name (Preliminary Specification)

Given the name of an object *name* relative to the context *ctx,* the 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. For example, for user jsmith, the names "myself/service/calendar" is equivalent to "user/jsmith/service/calendar" when resolved relative to the Initial Context.

```
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);
```

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

Base Attribute Interface

This section describes the operations in the base attribute interface. The interfaces to the objects used in operations in this interface are described in "Parameters Used in the Interface" on page 55.

- "XFN Attribute Model" on page 42
- "Relationship to Naming Operations" on page 43
- "Status Objects" on page 43
- "Single-Attribute Operations" on page 43
- "Multiple-Attribute Operations" on page 46

XFN Attribute Model

In the XFN attribute model, a set of zero or more attributes can be 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 attribute values are encoded.

The operations in the base attribute interface can be used to examine and modify the settings of attributes associated with existing named objects. These objects can be contexts or other types of objects. The attribute operations do not create names or remove names in contexts.

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

Typically, attributes of an object are manipulated through operations that operate on a single attribute, such as reading or updating a single attribute. Moreover, the client is typically expected to be 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 name service. To address these requirements, two kinds of attribute operations are defined:

- Single-attribute operations
- Multiple-value and multiple-attribute operations

Relationship to Naming Operations

An XFN attribute operation might not be equivalently expressed as 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 some attribute models associate attributes with a named object in the context in which the object is named. In others, an object's attributes are stored in the object itself. XFN accommodates both these models.

Note – Invoking an attribute operation using the target context and the terminal atomic name accesses either the attributes that are associated with the terminal name or the object named by the terminal name—this is dependent upon the underlying attribute model. This document uses the term *attributes associated with a named object* to refer to all of these cases.

XFN does not provide any guarantee about the validity of the relationship between the attributes and the reference associated with a given name. Some naming systems might 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 the attribute operations when manipulating references. Applications should avoid using specific knowledge about how an XFN context implementation over a particular naming system constructs references.

Status Objects

In each attribute operation, the caller supplies an `FN_status_t` parameter. The called function sets this status object as described in "Status Objects and Status Codes" on page 51. All status objects are handled in this manner for each operation in the base attribute interface; this will not be restated in the individual operation descriptions.

Single-Attribute Operations

```
fn_attr_get
fn_attr_modify
fn_attr_get_values
fn_valuelist_next
fn_valuelist_destroy
```

Each of these operations takes as arguments a context and composite name relative to this context and manipulates the attributes associated with the named object. Each operation sets a status object to describe the status of the operation.

Get Attribute

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

```
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);
```

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

Modify Attribute

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.

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

TABLE 2–1 XFN Attribute-Modification Operations

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 can 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 if an attribute with this identifier already exists. The set of values can be empty if the target naming system permits.

TABLE 2–1 XFN Attribute-Modification Operations	*(Continued)*
Operation Code	**Meaning**
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 given set). Create the attribute if it does not already exist. The set of values can be empty if the target naming system permits.
FN_ATTR_OP_REMOVE ()	Remove the attribute with the given attribute identifier and all 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_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 given set). 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 removes the attribute as well.

Get Attribute Values

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

```
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);
```

This set of operations is used to obtain the set of values of a single attribute, identified by *attribute_id*, associated with *name*, relative to *ctx*. If *name* is empty, the attribute associated with *ctx* is 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.

The operation `fn_valuelist_destroy()` frees the resources associated with the enumeration. This operation can be invoked at any time to terminate the enumeration.

Multiple-Attribute Operations

```
fn_attr_get_ids
fn_attr_multiget
fn_multigetlist_next
fn_multigetlist_destroy
fn_attr_multi_modify
```

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

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

Get Attribute Identifiers

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

```
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);
```

Get Multiple Attributes

Get one or more attributes associated with the object named *name* relative to the context *ctx*. If *name* is empty, the attributes associated with *ctx* are returned.

```
FN_multigetlist_t *fn_attr_multiget(
    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);
```

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 identifiers are relevant for this
operation. Each attribute (identifier, syntax, and 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 the
enumeration. This operation can be invoked at any time to terminate the enumeration.

Implementations are not required to return all attributes requested by *attr_ids*. Some
might choose to return only the attributes found successfully; such implementations
might not necessarily return identifiers for attributes that could not be read.

Modify Multiple Attributes

Modify the attributes associated with the object named *name*, relative to *ctx*.

```
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);
```

In the *mods* parameter, the caller specifies a sequence of modifications that are to be
performed in order on the attributes. Each modification in the sequence specifies a
modification operation code (shown in Table 2–1) 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 point 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 a failure and a NULL pointer is returned in *unexecuted_mods*, that means no modifications were executed.

Extended Attribute Interface (Preliminary Specification)

```
fn_attr_search
fn_searchlist_next
fn_searchlist_destroy
fn_attr_ext_search
fn_ext_searchlist_next
fn_ext_searchlist_destroy
```

The XFN extended attribute interface consists of operations that perform searching and creation of objects in the namespace with attributes. The operations in this interface are considered "preliminary," in that they are not yet standard and might change in the next revision of the specification.

Attribute Search Interface

The search interface contains several operations: a basic search operation, which performs associative lookup in a single context, and an extended search operation that allows the search criteria to be specified using an expression . It also allows the scope of the search to encompass a wider scope than only a single context.

Basic Search

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*. Using 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.

```
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);
```

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.

Extended Search

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.

```
FN_ext_searchlist_t *fn_attr_ext_search(
    FN_ctx_t *ctx,
    const FN_composite_name_t *name,,
    const FN_search_control_t *control
    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);
```

These options are:

1. The scope of the search. This can be any of the following:

   ▪ Search the named object

- sSearch the context named by name
- Search the entire subtree rooted at the context named by name
- Ssearch 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 teriminal 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.

Object Creation with Attributes

```
fn_attr_bind
fn_attr_create_subcontext
```

At times 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 that the versions in the extended attribute interface allow an extra parameter for specifying attributes to be associated with the new binding.

Bind with Attributes

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 part of *name* to the supplied reference in the target context. The target context must already exist.

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

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.

Create Subcontext with Attributes

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 *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.

```
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);
```

Status Objects and Status Codes

The result statuses of operations in the context interface and the attribute interface are encapsulated in `FN_status_t` objects. The `FN_status_t` object contains 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. If the error occurred while resolving an XFN link, the status object contains additional information about that error.

The status object contains several items of information as shown in Table 2–2.

TABLE 2–2 Status Object

Information Type	Description
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 might have been successful beyond this point, but the error can not be pinpointed further.
Resolved reference	The reference to which the resolved name is bound.
Remaining name	The remaining unresolved portion of the name.
Diagnostic message	Any diagnostic message returned by the context implementation.
Link status code	If an error occurs while resolving an XFN link, the primary status code has the value FN_E_LINK_ERROR, and this 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 resolved portion of the name in the XFN link.
Link diagnostic message	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 codes in Table 2–3are determined by XFN.

TABLE 2–3 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.

TABLE 2–3 Status Codes   *(Continued)*

Code	Meaning
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 interfaces.
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 might be because the naming system is busy or is not providing service. In some implementations this might not be distinguished from a communication failure.
FN_E_ILLEGAL_NAME	The name supplied to the operation was not a well-formed composite name, or one of the component names was not well formed according to the syntax of the naming systems 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 (on memory, file descriptors, communication ports, stable media space, for example) 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, because an update operation occurred during the enumeration, or for some other reason.

TABLE 2–3 Status Codes   *(Continued)*

Code	Meaning
FN_E_INVALID_SYNTAX_ATTRS	The syntax attributes supplied are invalid or insufficient to fully specify the syntax.
FN_E_LINK_ERROR	An error occurred while resolving an XFN link encountered during resolution of the supplied name.
FN_E_LINK_LOOP_LIMIT	A nonterminating loop (cycle) in the resolution is suspected. This arises due to XFN links encountered during the resolution of a supplied 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 (and thus a cycle is suspected).
FN_E_MALFORMED_LINK	A malformed link reference was encountered. For `fn_ctx_lookup_link()`, the name supplied resolved to a reference that was not a link.
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 does 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.

TABLE 2–3 Status Codes *(Continued)*

Code	Meaning
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.

Parameters Used in the Interface

- "Composite Names" on page 55
- "References and Addresses" on page 56
- "Identifiers" on page 56
- "Strings" on page 57
- "Attributes and Attribute Values" on page 57
- "Attribute Sets" on page 57
- "Attribute-Modification Lists" on page 57

This section gives an overview of the types of parameters that are passed and returned by operations in the base context and attribute interfaces. Manipulation of these objects using their corresponding interfaces 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 "The Base Context Interface" on page 33and "Base Attribute Interface" on page 42.

Composite Names

A composite name is represented by an object of type FN_composite_name_t. 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 "Syntax" on page 83for a description of composite name syntax and structure.) Operations are provided to iterate over this sequence, modify it, and compare two composite names.

References and Addresses

A 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, but XFN does not dictate its precise use.

Each address in a reference 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 can arise for various reasons; for example, because the object offers 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 on the type of the reference. However, this interpretation is intended to occur below the application layer.

Most application developers should not be required to manipulate the contents of either address or reference objects themselves. These interfaces are generally used within service libraries.

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 `FN_identifier_t` type is used to represent an identifier. It consists of an unsigned integer, 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, as shown in Table 2–4.

TABLE 2–4 XFN Identifier Formats

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 (ISBN 1-872630-95-2).

TABLE 2–4 XFN Identifier Formats	*(Continued)*
Identifier Format	**Description**
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 (ISO 8824).

Strings

The FN_string_t type represents character strings in the XFN interface. It provides a layer of insulation from specific string representations. The FN_string_t operations contain operations for string comparison, substring searches, and manipulation. The FN_string_t type supports multiple code sets. In Solaris 2.5, FNS supports ISO 646.

Attributes and Attribute Values

An attribute is represented by the FN_attribute_t type, and contains:

- An attribute identifier (of type FN_identifier_t)
- A syntax (of type FN_identifier_t)
- A set of distinct values (each value is a sequence of octets of type FN_attrvalue_t)

Various operations allow the construction, destruction, and manipulation of an attribute.

Attribute Sets

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

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

Attribute-Modification Lists

Use an attribute-modification list to specify multiple modification operations to be performed on the attributes associated with a single named object.

An attribute-modification list is represented by the FN_attrmodlist_t type. It consists of an ordered list of attribute-modification specifiers. Each specifier contains

an operation and an attribute object. The attribute's identifier indicates the attribute that is to be operated upon. How the attribute's values are used depends on the operation.

The operation specifier is one of the values described in Table 2–1. The operations should be done in the order in which they appear in the list.

Parameters Used in Extended Search (Preliminary Specification)

The types of objects used to specify the scope and details of an extended search operation:

- the search control operations (`FN_search_control_t`)
- the search filter expression (`FN_search_filter_t`)

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. The default is `FN_SEARCH_ONE_CONTEXT`.

TABLE 2–5 Different Scopes for Searching

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.

- Follow links during search. This determines whether links encountered during the search will be followed. 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.

The default is to 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.

  The default is to return all named objects found.

- Return reference. This determines whether the reference of the object is returned.

  The default is to not return the reference.

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

  The default is to not return any attributes.

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.

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 is listed in Table 2–6.

TABLE 2–6 Substitute Tokens in Search Filter Expressions

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

Substitute Tokens in Search Filter Expressions

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

Table 2–7 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.

TABLE 2–7 Relational Operators in Search Filter Expressions

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. Table 2–8contains examples of wildcarded strings and their meaning.

TABLE 2–8 Examples of Wildcarded Strings

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

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.

The extended operations shown in Table 2–9are currently defined:

TABLE 2–9 Extended Operations

`'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.

TABLE 2–9 Extended Operations   *(Continued)*

`'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. `FN_E_OPERATION_NOT_SUPPORTED` is returned when `fn_attr_ext_search()` is not supported.

Table 2–10 contains examples of filter expressions that contain extended operations.

TABLE 2–10 Extended Operations in Search Filter Expressions

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`.

Parsing Compound Names

- "Syntax Attributes" on page 64
- "XFN Standard Syntax Model" on page 64

Because most applications treat names as opaque data, most clients of the XFN interface do not need to parse compound names from specific naming systems. When an application (such as a browser) needs to do this, however, it can use the `FN_compound_name_t` object.

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 can 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 is set to an implementation-specific string and different or additional syntax attributes are in the set.

Syntax attributes of a context can be generated automatically by a context, in response to `fn_ctx_get_syntax_attrs()`, or can be created and updated using the attribute operations. This is implementation dependent.

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. This model is expressed in terms of syntax properties of the naming convention and it uses XFN attributes to describe properties of the syntax.

Unless otherwise qualified, the syntax attributes described in this section 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.

In the XFN standard syntax model, these 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 mark, or escape string is escaped if preceded immediately (on the left) by the escape string.

3. A non-escaped begin-quote that precedes a component must be matched by a non-escaped end-quote at the end of the component. Quotes embedded in nonquoted 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 (nonescaped) begin-quote and the end-quote, it is ignored.

6. When the separator is escaped, it is not treated as a separator. 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 that precedes a character other than an escape string, a begin-quote or an end-quote is consumed (in other words, escaping a non-meta characater 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.

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. 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.

Table 2–11 lists all the XFN standard syntax model attributes.

TABLE 2–11 XFN Syntax Attributes

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_synt ax_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 (that is, not hierarchical, with all names atomic).
fn_std_syntax_separator	Its value is the separator string for this name syntax. This attribute is required unless the fn_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 present, it indicates that names differing 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.

TABLE 2–11 XFN Syntax Attributes *(Continued)*

Attribute Identifier	Attribute Value
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 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_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 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_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 present, its value identifies the code sets of the string representation for this syntax. Its value consists of a structure containing an array of code sets supported by the context; the first member of the array is the preferred code set of the context. The values for the code sets are defined in the X/Open code set registry currently defined in DCE RFC 40.1. If this attribute is not present, or if the value is empty, the default code set is ISO 646 (same encoding as ASCII).

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_syuntax_type" with the value "standard."

XFN Programming Examples

This chapter presents self-contained executable programs for the following programs:

- A namespace browser
- A printer client and server
- A tool to populate attributes of users.

Namespace Browser Example

Figure 3–1illustrates the XFN APIs that are used by the browser application.
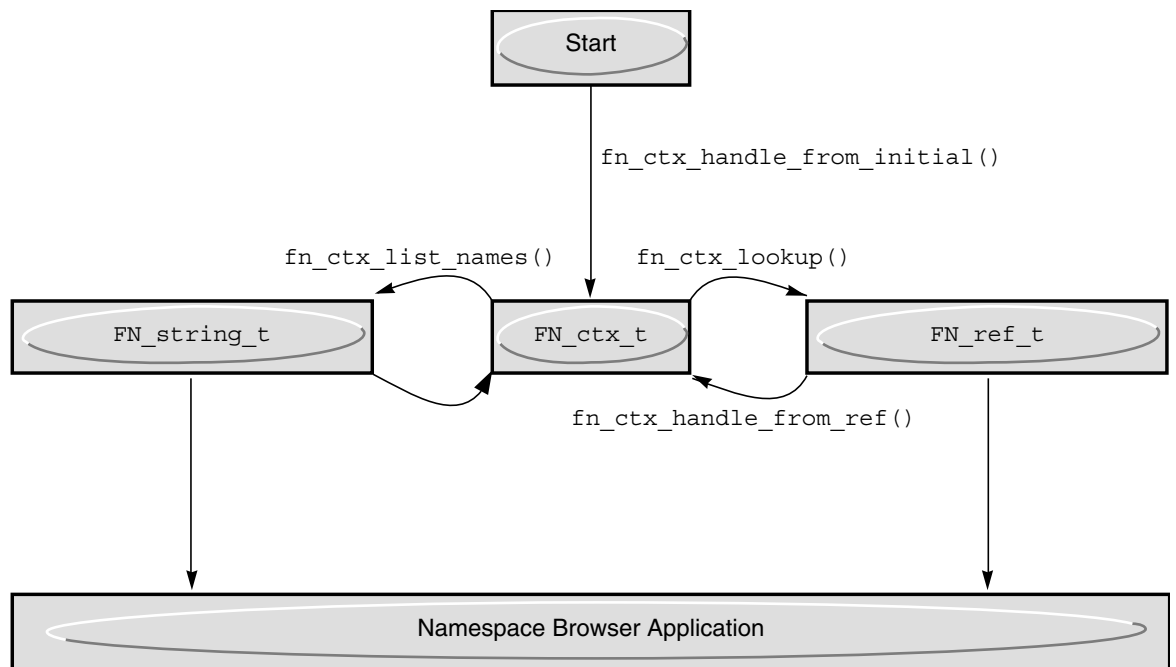
FIGURE 3–1 Diagram of `fnbrowse` Program

The first example is a browser that lists all names that it finds in the namespace. When the program is invoked, the browser is set at the initial context or the composite name given on the command line.

See "Commands" on page 76 and "Sample Output" on page 76.

EXAMPLE 3–1 `fnbrowse` Source

```
/*
 * fnbrowse.c -- FNS namespace browser.
 *
 * To keep this example program relatively short,
 * limited error checking is done.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <xfn/xfn.h>


#define LINELEN 128    /* maximum length of input line */

typedef enum {CMD_DOWN, CMD_UP, CMD_LIST, CMD_SHOW, CMD_QUIT} command;

static FN_status_t *status;
```

EXAMPLE 3–1 fnbrowse Source   *(Continued)*

```
static unsigned     int auth = 0; /* non-authoritative */


/* Lookup a context named relative to the initial context. */
FN_ctx_t *lookup(const FN_composite_name_t *name);

/* Set the browser's focus to the given context. */
void browse(FN_ctx_t *ctx);

/* Set the browser's focus to a subcontext of the given context. */
void cmd_down(FN_ctx_t *ctx, const FN_composite_name_t *child);

/* Print the names bound within a context. */
void cmd_list(FN_ctx_t *ctx);

/*
 * Print a description of the reference bound to "child" in the
 * given context or, if "child" is the empty string, the reference
 * of the context itself.
 */
void cmd_show(FN_ctx_t *ctx, const FN_composite_name_t *child);

/*
 * Read and parse the next command typed by the user.  If the
 * command has an argument, set *argp to point to the argument.
 */
command read_command(FN_string_t **argp);

/* Print an error message, and the description associated
 * with "status".
 */
void error(const char *msg);


int
main(int argc, char *argv[])
{
        unsigned char    *target;

        switch (argc) {
    case 1:
        target = (unsigned char *)"";
        break;
    case 2:
        target = (unsigned char *)argv[1];
        break;
    default:
        fprintf(stderr, "Usage: %s [<composite_name>]\n",
            argv[0]);
        return (1);
    }
```

EXAMPLE 3–1 fnbrowse Source *(Continued)*

```
    status = fn_status_create();

    browse(lookup(fn_composite_name_from_str(target)));
    return (0);
}


FN_ctx_t *
lookup(const FN_composite_name_t *name)
{
    FN_ctx_t    *ctx;
    FN_ref_t    *ref;

    ctx = fn_ctx_handle_from_initial(auth, status);
    if (ctx == NULL) {
       error("Could not construct initial context");
       exit(1);
    }
    if (fn_composite_name_is_empty(name)) {
        return (ctx);
    }
    ref = fn_ctx_lookup(ctx, name, status);
    fn_ctx_handle_destroy(ctx);
    if (ref == NULL) {
       error("Lookup failed");
       exit(1);
    }
    ctx = fn_ctx_handle_from_ref(ref, auth, status);
    fn_ref_destroy(ref);
    if (ctx == NULL) {
       error("Could not construct context handle");
       exit(1);
    }
    return (ctx);
}


void
browse(FN_ctx_t *ctx)
{
  FN_string_t         *arg;
  FN_composite_name_t     *child;

  while (1) {
     switch (read_command(&arg)) {
     case CMD_DOWN:
        child = fn_composite_name_from_string(arg);
        fn_string_destroy(arg);
             cmd_down(ctx, child);
        fn_composite_name_destroy(child);
        break;
     case CMD_UP:
```

EXAMPLE 3–1 fnbrowse Source   *(Continued)*

```
        return;
    case CMD_LIST:
        cmd_list(ctx);
        break;
    case CMD_SHOW:
        child = fn_composite_name_from_string(arg);
        fn_string_destroy(arg);
        cmd_show(ctx, child);
        fn_composite_name_destroy(child);
        break;
    case CMD_QUIT:
        exit(0);
    }
  }
}


void
cmd_down(FN_ctx_t *ctx, const FN_composite_name_t *child)
{
  FN_ref_t     *ref;
  FN_ctx_t     *subctx;

  ref = fn_ctx_lookup(ctx, child, status);
  if (ref == NULL) {
      error("Lookup failed");
      return;
  }
  subctx = fn_ctx_handle_from_ref(ref, auth, status);
  fn_ref_destroy(ref);
      if (subctx == NULL) {
      error("Could not construct context handle");
      return;
  }
  browse(subctx);
  fn_ctx_handle_destroy(subctx);
}


void
cmd_list(FN_ctx_t *ctx)
{
  FN_string_t         *empty_string = fn_string_create();
  FN_composite_name_t    *empty_name;
  FN_namelist_t        *children;
  FN_string_t         *child;
  unsigned intstatcode;
  int   has_children = 0;

  empty_name = fn_composite_name_from_string(empty_string);
  fn_string_destroy(empty_string);
```

EXAMPLE 3–1 fnbrowse Source   *(Continued)*

```
  children = fn_ctx_list_names(ctx, empty_name, status);
  fn_composite_name_destroy(empty_name);

  if (children == NULL) {
      error("Could not list names");
      return;
    }
  while ((child = fn_namelist_next(children, status))
    != NULL) {
    has_children = 1;
    printf("%s  ", fn_string_str(child, &statcode));
    fn_string_destroy(child);
  }
  if (has_children) {
      printf("\n");
  }
  fn_namelist_destroy(children);
}


void
cmd_show(FN_ctx_t *ctx, const FN_composite_name_t *child)
{
    FN_string_t    *desc;
    FN_ref_t    *ref;
    unsigned int    statcode;

    ref = fn_ctx_lookup(ctx, child, status);
    if (ref == NULL) {
        error("Lookup failed");
        return;
    }

    desc = fn_ref_description(ref, 2, NULL);
    fn_ref_destroy(ref);
    if (desc != NULL) {
        printf("%s", fn_string_str(desc, &statcode));
        fn_string_destroy(desc);
    } else {
        printf("[No description]\n");
        }
}


command
read_command(FN_string_t **argp)
{
    char    buf[LINELEN + 1];
    char *cmd;
    char *child;

    while (printf("\n> "), fflush(stdout), gets(buf) != NULL) {
```

EXAMPLE 3–1 fnbrowse Source   *(Continued)*

```
        cmd = strtok(buf, " \t");
        if (cmd == NULL) {
             continue;
              }
        if (strcmp(cmd, "down") == 0) {
            child = strtok(NULL, " \t");
            if (child != NULL) {
                *argp =
                    fn_string_from_str((unsigned char *)child);
                return (CMD_DOWN);
            }
        }
        if (strcmp(cmd, "up") == 0) {
            return (CMD_UP);
        }
        if (strcmp(cmd, "list") == 0) {
            return (CMD_LIST);
        }
        if (strcmp(cmd, "show") == 0) {
            child = strtok(NULL, " \t");
            *argp = (child != NULL)
                ? fn_string_from_str((unsigned char *)child)
                : fn_string_create();
            return (CMD_SHOW);
        }
        if (strcmp(cmd, "quit") == 0) {
            return (CMD_QUIT);
             }
        fprintf(stderr, "Valid commands are:  "
            "down <child>, up, list, show [<child>], quit\n");
    }
    return (CMD_QUIT);     /* EOF */
}


void
error(const char *msg)
{
    FN_string_t    *reason;
    unsigned int    statcode;

    fprintf(stderr, "%s", msg);
    reason = fn_status_description(status, 0, NULL);
    if (reason != NULL) {
        fprintf(stderr, ": %s",
            (const char *)fn_string_str(reason, &statcode));
        fn_string_destroy(reason);
    }
    fprintf(stderr, "\n");
}
```

Compiling and Executing Browser Example

To compile Example 3–1, type:

```
% cc -o fnbrowse fnbrowse.c -lxfn
```

To browse the namespace starting from the initial context, the program is invoked as:

```
% fnbrowse
```

Or to browse a composite name and its descendents, type:

```
% fnbrowse composite_name
```

Commands

The commands supported by the fnbrowse program are summarized in Table 3–1.

TABLE 3–1 Namespace Browser Commands

Command	Usage
down *child*	Sets the browser at the subcontext of the child
up	Sets the browser at one level higher than the current context
list	Lists the names bound within the current context
show	Prints the reference of the current context
show *child*	Prints the reference of the current context's child
quit	Exits the browser

Sample Output

Sample output for navigating the entire namespace is displayed here.

Note the following:

- The first list command shows the initial context bindings.
- The fnbrowse program lists all names it finds in the namespace, including names with underscores. These names are explained in "Initial Context Bindings" on page 17.
- The three dots (. . .) represent the global namespace.

```
% fnbrowse
> list
_myorgunit  ...  _myself  thishost  myself  _orgunit  _host
```

```
_thisens  myens  thisens  org  orgunit  thisuser  _thishost
myorgunit  _user  thisorgunit  host  _thisorgunit  _myens  user
```

Navigating the namespace is accomplished with the up and down commands. In the
following output, the down command brings the focus of the browser to the enterprise
root of the namespace, thisens (can also be myens). The show command displays
information about the reference and address type for thisens.

```
> down thisens
> show
Reference type: onc_fn_enterprise
Address type: on_fn_nisplus
  length: 20
  context type: enterprise root
  representation: normal
  version: 0
  internal name: eng.wiz.com
> up
> down thisorgunit
```

Continuing with the example, this list command shows the contexts for
thisorgunit.

```
> list
service  _fs  _host  _service  _site  site  _user  host  fs  user

> down usr
Lookup failed: Name Not Found: 'usr'

> down service
> list
printer

> down printer
```

The list command shows the printer names that are bound in the printer context.
The show command displays the reference for the child, colorful.

```
> list
celeste  _default  color  colorful  quartz  nuttree  puffin

> show colorful
printer
Reference type: onc_printers
Address type: onc_printers_bsdaddr
  length: 12
  data: 0x00 0x00 0x00 0x08 0x62 0x6c 0x61 0x63 0x6b 0x63
....blackc 0x61 0x74 at
> down colorful
Could not construct context handle: No Supported Address
> quit
%
```

Printer Programming Example

Printer client and server software can take advantage of FNS to advertise and to browse the printers available with respect to organizations, sites, users, and hosts. The APIs used by the server and the client are XFN APIs, thereby ensuring that the application is portable across the different naming services used for storing printer bindings.

The programming example in this section shows how printer clients and servers obtain and store printer bindings. Users can then make use of the FNS commands, `fnlist` and `fnlookup`, to browse the `printer` context.

For example, use `fnlist` to look at the user `printer` context for `jsmith`:

```
% fnlist user/jsmith/service/printer
celeste
lp
_default
myprinter
```

Similarly, you can look at the organization's printers:

```
% fnlist org/wiz.com/service/printer
sales_printer
mktg_printer
eng_printer
```

Alternatively, you can type:

```
% fnlist thisorgunit/service/printer
```

You can look at the printers at a specific site, for example, the printers in the `MTV` site:

```
% fnlist thisorgunit/site/MTV/service/printer
b1_printer
b2_printer
```

Client

The scenario for Example 3–2 is a user who would like to print to a printer named `colorful` in his organization's context, `thisorgunit/service/printer/colorful`. The example printer client illustrates how the bindings for a specific printer are obtained.

The variable *printer_binding* contains the reference (the binding information) of the named printer. Using the binding information, the printer client can connect to the server and send the printer request. Note that the `fn_ctx_lookup()` function can be

replaced by `fn_ctx_list_name()` or `fn_ctx_list_bindings()` to list all the names and their bindings.

EXAMPLE 3–2 Print Client source

```c
#include <stdio.h>
#include <xfn/xfn.h>
#include <string.h>
#include <stdlib.h>

/*
 * Routine to obtain the address of a specific printe.
 * This routine takes the printer name and the address type
 * as the input arguments and returns the address
 * of the requested printer.
 */

char *
get_address_of_printer(const char *printer_name,
        const char *address_type)
{
   FN_composite_name_t *printer_name_comp;
   FN_status_t *status = fn_status_create();
   FN_ctx_t *initial_context;
   FN_ref_t *printer_ref;
   const FN_identifier_t *addr_id;
   const FN_ref_addr_t *address;
   char *addr_data; /* Return value */

   void *ip;
   size_t address_type_len, addr_len;

   /* Convert the printer name to a composite name */
   printer_name_comp = fn_composite_name_from_string(
      (const unsigned char *)printer_name);

   /* Get the initial context */
   initial_context = fn_ctx_handle_from_initial(0, status);

   /* Check status for any error messages */
   if (!fn_status_is_success(status)){
    fprintf(stderr,
         "Unable to obtain the initial context\n");
    return (0);
    }
  /* Perform a lookup for the printer name */
  printer_ref = fn_ctx_lookup(initial_context,
     printer_name_comp, status);

  /* Check status for any error messages */
  if (!fn_status_is_success(status)){
    fprintf(stderr, "Lookup failed on: %s\n",
        printer_name);
    return (0);
```

EXAMPLE 3–2 Print Client source   *(Continued)*

```
  }

  fn_ctx_handle_destroy(initial_context);
  fn_composite_name_destroy(printer_name_comp);
  address_type_len = strlen(address_type);

  /* Obtain the requested address from the address type */
  for (address = fn_ref_first(printer_ref, &ip);
     address != NULL;
     address = fn_ref_next(printer_ref, &ip)) {
         addr_id = fn_ref_addr_type(address);
         if (addr_id->length == address_type_len &&
            strncmp(address_type,
                (char *)addr_id->contents,
                address_type_len) == 0)
                break;
  }
  if (address == NULL)
      return (0);
  addr_len = fn_ref_addr_length(address);
  addr_data = (char *)(malloc(addr_len + 1));
  strncpy(addr_data,(char*)(fn_ref_addr_data(address)),
          addr_len);
  addr_data[addr_len] = '\0';
  fn_ref_destroy(printer_ref);
  return (addr_data);
}
```

Calling the Printer Client Function

The following code could be used to call the get_address_of_printer() routine shown above.

```
char* printer_server;
printer_server = get_address_of_printer

    "thisorgunit/service/printer/colorful",

    "onc_bsdaddr");
```

Server

Using the XFN APIs, print servers can advertise their services. Example 3–3 illustrates a host, labpc, that would like to advertise the binding for the color printer colorful. The FNS name for this printer is thisorgunit/service/printer/colorful.

The main tasks are to obtain the composite name for the printer name, to generate the binding (reference) for the printer, and to bind the name and references to the FNS namespace.

EXAMPLE 3–3 Print Server Source

```
#include <stdio.h>
#include <xfn/xfn.h>
#include <string.h>
/*
 * Routine to export the printer binding to the FNS name space.
 * This routine takes the printer name along with its
 *  reference type, address type, and address. Returns the status.
 */
int
export_printer_to_fns(const char *printer_name,
                const char *reference_type,
                const char *address_type,
                const char *address_data)
{
  int return_status;
  FN_composite_name_t *printer_name_comp;
  FN_identifier_t ref_id, addr_id;
  FN_status_t *status;
  FN_ref_t *printer_ref;
  FN_ref_addr_t *address;
  FN_ctx_t *initial_context;

  /* Obtain the initial context */
  status = fn_status_create();
  initial_context = fn_ctx_handle_from_initial(0, status);
  /* Check status for any error messages */
  if ((return_status = fn_status_code(status)) != FN_SUCCESS) {
      fprintf(stderr, "Unable to obtain the initial context\n");
      return (return_status);
  }
  /* Construct the composite name for the printer name */
  printer_name_comp = fn_composite_name_from_string(
      (unsigned char *)printer_name);

  /* Construct the printer address */
  addr_id.format = FN_ID_STRING;
  addr_id.length = strlen(address_type);
  addr_id.contents = (void *) address_type;
  address = fn_ref_addr_create(&addr_id,
      strlen(address_data), (const void *) address_data);

  /* Construct the printer reference */
  ref_id.format = FN_ID_STRING;
      ref_id.length = strlen(reference_type);
      ref_id.contents = (void *) reference_type;
      printer_ref = fn_ref_create(&ref_id);

      /* Add the printer address to the printer reference */
```

EXAMPLE 3–3 Print Server Source   *(Continued)*

```
    fn_ref_append_addr(printer_ref, address);

    /* Bind the reference to the context */
    fn_ctx_bind(initial_context, printer_name_comp, printer_ref, 0,
        status);
    /* Check the error status and return */
return_status = fn_status_code(status);

fn_composite_name_destroy(printer_name_comp);
fn_ref_addr_destroy(address);
fn_ref_destroy(printer_ref);
fn_status_destroy(status);
fn_ctx_handle_destroy(initial_context);
return (return_status);
}
```

Calling the Printer Server Function

The following code could be used to call the export_printer_to_fns() routine shown above.

```
export_printer_to_fns
    "thisorgunit/service/printer/colorful",
        "onc_printers",
        "onc_bsdaddr",
        "labpc");
```

XFN Composite Names

This appendix describes XFN composite names in detail.

Syntax

The standard string form for XFN composite names is the concatenation of the components of a composite name from left to right, with the XFN component separator character (/) separating each component. Components can be quoted using either double-quote (" ") or single-quote (' ') pairs. You can use a backslash character (\) to excape the XFN component separator or quote characters if the intention is for these characters not to behave as separators or quotes. Note that quotation marks and escape characters are interpreted as such only when they appear in places that need quotes or escapes. For example, a quote appearing in an unquoted component is not interpreted as a quote.

XFN defines an abstract data type, FN_composite_name_t, for representing the structural form of a composite name. XFN also defines the syntax of how component

83

string names are composed into an XFN composite name and the corresponding rules for converting an XFN composite name to its structural form from its string form, and vice versa. The XFN client interface includes operations that perform these conversions.

Table A–1 contains some examples of how the string form of XFN composite names are decomposed into components according to the syntax of XFN composite names. See also "Composite Name Encoding" on page 89 for more information.

TABLE A–1 String and Structural Forms of XFN 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/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

String form	Components in FN_composite_name_t
`a/\"b`	`a, "b`

Composite Name and Naming System Boundaries

There might not be a one-to-one correspondence between component separators and naming system boundaries if a composite name contains names from naming systems that use the same character as the XFN component separator to separate their atomic names. Consequently, a component of a composite name might represent an atomic name from a hierarchical naming system that uses the XFN component separator or a compound name. *Strong separation* and *weak separation* refer to how a context considers the XFN component separator as a naming system boundary.

Strong Separation

An XFN context that treats the XFN component separator as a naming system boundary supports strong separation. An XFN component separator that appears *within* a component to be resolved by the context must be escaped or quoted.

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, and does not use the XFN component separator as its atomic separator, supports strong separation. Examples of naming systems that support strong separation are DNS and NIS+, both of which have right-to-left dot-separated names. The following are examples of names with DNS and NIS+ components, respectively.

```
.../wiz.com/orgunit/ppt
orgunit/accountspayable.finance/user/jsmith
```

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. 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 might not necessarily signify a naming system boundary.

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.

CDS names and X.500 names are examples of names that use the XFN component separator as their atomic name separator. X.500 supports weak separation using a syntactic method (by scanning for typed names) while CDS supports weak separation by determining the naming system boundary dynamically.

The following example shows a composite name with an X.500 component.

```
.../c=us/o=wiz.com/orgunit/ppt
```

Note – An XFN context that supports weak separation using only syntax-specific discovery of its naming system boundary might not always be federated with arbitrary subordinate naming systems. If the subordinate naming system has a naming syntax that is indistinguishable from that of the superior naming system, the superior naming system is not able to identify the naming system boundary.

Naming systems that use the same character as the XFN component separator as their atomic separator, and which cannot support weak separation because it cannot use a syntactic or dynamic method to determine the naming system boundary, 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.

Composite Name Resolution

Composite name resolution combines resolution in each component naming system and resolution across federated naming system boundaries. There are several techniques for resolving an XFN composite name in the underlying federation of naming systems.

This section describes two implementation techniques for composite name resolution across a naming system boundary. One technique uses an *explicit* next naming system pointer (NNSP) to resolve across a naming system boundary, while the other uses an *implicit* NNSP.

An NNSP is the XFN reference of an XFN context in which composite name components from subordinate naming systems are to be resolved. NNSPs are entities that "tie" naming systems together into a federated system. NNSPs can be bound to names, in which case they are *explicit* NNSPs or *junctions*. NNSPs can also be nameless, in which case they are *implicit* NNSPs.

Explicit NNSPs: Junctions

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

Composite name resolution involving junctions proceeds as follows, depending on whether the context supports strong or weak separation.

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.

A context that supports weak separation and 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 resolved by the junction. Determination of whether a component is a junction can be done statically, using a syntactic policy, or dynamically during resolution.

Implicit NNSPs

When a context does not want to use part of its namespace for junctions, it uses implicit NNSPs for federating subordinate naming systems. An implicit NNSP is named using the XFN component separator. For example, the name `wiz.com/` names the implicit NNSP of `wiz.com`. Each context can have one implicit NNSP.

Composite name resolution involving implicit NNSPs proceeds as follows, depending on whether the context supports strong or weak separation.

A context that supports strong separation and resolves composite names using an implicit NNSP consumes the first component of the composite name supplied to it. Any remaining components are resolved in the context pointed to by the implicit NNSP of the first component.

A context that supports weak separation and implicit NNSPs 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 NNSP. This can be achieved using a static, syntactic policy or a dynamic, resolution-based policy.

With the syntactic policy, a context syntactically discovers where the boundary between its naming system and the subordinate naming system lies. This can impose certain restrictions on the syntax of subordinate naming systems. Subordinate naming systems must not permit, as valid top-level names, that are syntactically indistinguishable from names allowed in the 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 sign (=). The superior naming system might impose as a policy that subordinate naming systems must not have top-level names with an equal sign 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 NNSP 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 might 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 NNSP 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 NNSP. In this case, the binding in the last context takes precedence. This way of supporting weak separation requires the context to have the capability of returning remaining unresolved parts of a given name.

Coexistence of Explicit and Implicit NNSPs

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

XFN Links

An XFN 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 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 can 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 nonterminating resolutions, implementations can define limits on the number of XFN links that can may be resolved in any single operation invoked by the caller.

Composite Name Encoding

All XFN implementations are required to support the ISO 646 portable representation (same encoding as ASCII) for XFN composite names. All other representations are optional.

All characters of the string form of an XFN composite name use a single encoding. Characters with different encodings cannot exist in the same name string. 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. Strings with code sets that are determined by the implementation to be compatible are converted without loss of information into a single representation, which is also determined by the implementation. When an implementation discovers that a composite name has components with incompatible code sets, it returns the error code `FN_E_INCOMPATIBLE_CODE_SETS`.

Backus-Naur Form (BNF)

The following defines the standard string form of XFN composite names in Backus-Naur Form (BNF). All the characters of the string representation of one name must uniformly use the same encoding and locale information. The notations used are as shown in Table A–2:

TABLE A–2 Backus-Naur Notation

Symbol	Meaning
::=	Is defined to be
|	Alternatively
<text>	Nonterminal 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 shown in Table A–3:

TABLE A–3 XFN Composite Name Syntax Using BNF

XFN Composite Name	BNF Syntax
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>::=	`/`

XFN Composite Name	BNF Syntax
<Quote1>::=	"
<Quote2> ::=	'
<MetaChar> ::=	<EscapeChar> | <ComponentSep>

<SimpleChar> ::=	`// any character from <CharSet> with <MetaChar>, <Quote1>,`
	`// and <Quote2> excluded. An<EscapeChar> <MetaChar>, or`
	`// <EscapeChar> <Quote1>, or <EscapeChar> <Quote2> is equivalent to`
	`// <SimpleChar>.`

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

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

Decomposing the Composite Name String

The function `fn_composite_name_from_string()` returns an XFN composite name in its structural form, `FN_composite_name_t`, given the composite name's string representation. The syntax rules used by `fn_composite_name_from_string()` are as follows:

- An XFN 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:

   a. The first component is the string preceding the first occurrence of a component separator.

   b. Empty components are processed as follows:

      i. A leading component separator (the composite name begins with a component separator) means a leading null component.

      ii. A trailing component separator (the composite name ends with a component separator) means a trailing null component.

   c. Two consecutive component separators mean a null component.

   d. The name string that immediately follows the last component separator of the composite name is the final component.

4. A component string is evaluated from left to right and converted into its standard form according to the following rules:

   a. 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.

   b. 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 treated just as any other *<SimpleChar>*.

   c. An unmatched begin quote (missing or misplaced end quote) fails with an FN_E_ILLEGAL_NAME status.

   d. Quotes are considered to be escaped in quoted strings if a matching quote character is preceded immediately by the unescaped *<EscapeChar>*.

   e. 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 nonmatching quote characters enclosed in quoted strings are treated just as any other *<SimpleChar>*.

   f. Any of the defined metacharacters (*<ComponentSep>* and *<EscapeChar>*) 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>*).

   g. *<Quote1>* and *<Quote2>* are considered to be escaped in an unquoted component if and only if *<EscapeChar>* is preceded by a component separator (that is, sequences *<ComponentSep> <EscapeChar> <Quote1>* or *<ComponentSep> <EscapeChar> <Quote2>*). Other occurrences of *<Quote1>* and *<Quote2>* in an unquoted component are treated just as any other *<SimpleChar>*.

h. Any occurrence of escaped *<MetaChar>*, escaped *<Quote1>*, or escaped *<Quote2>* in unquoted components is substituted by the corresponding unescaped character.

i. No substitution is done for *<EscapeChar> <SimpleChar>*. *<EscapeChar>* *<SimpleChar>* maps to *<EscapeChar> <SimpleChar>*.

Composing the Composite Name String

The function `fn_string_from_composite_name()` returns the string representation of an XFN composite name given its structural form (`FN_composite_name_t`). The following are the rules used by `fn_string_from_composite_name()`.

1. The components are added to the composite name string in left to right order (that is, rightmost is the tail).

2. Successive components are separated by the component separator (*<ComponentSep>*).

3. Empty components are handled in the following way:

   a. A leading empty component is represented by a leading *<ComponentSep>*.

   b. A trailing empty component is represented by a trailing *<ComponentSep>*.

   c. 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>*.

XFN Composite Names Syntax

This appendix provides supplemental information about XFN composite name syntax.

XFN Composite Name Encoding

All XFN implementations are required to support the ISO 646 portable representation (same encoding as ASCII) for XFN composite names. All other representations are optional.

All characters of the string form of an XFN composite name use a single encoding. There cannot be characters with different encodings in the same name string. 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. Strings with code sets that are determined by the implementation to be compatible are converted without loss of information into a single representation, which is also determined by the implementation. When an implementation discovers that a composite name has components with incompatible code sets, it returns the error code FN_E_INCOMPATIBLE_CODE_SETS.

XFN Backus-Naur Form (BNF)

The following defines the standard string form of XFN composite names in Backus-Naur Form (BNF). All the characters of the string representation of one name must uniformly use the same encoding and locale information. The notations used are shown in Table B–1:

TABLE B–1 Backus-Naur Notation

Symbol	Meaning
::=	Is defined to be
|	Alternatively
\<text\>	Nonterminal 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 shown in Table B–2:

TABLE B–2 XFN Composite Name Syntax Using BNF

XFN Composite Name	BNF Syntax
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\>::=	/

TABLE B–2 XFN Composite Name Syntax Using BNF   *(Continued)*

XFN Composite Name	BNF Syntax
<Quote1>::=	"
<Quote2> ::=	'
<MetaChar> ::=	<EscapeChar> | <ComponentSep>

<SimpleChar> ::=	```// any character from <CharSet> with <MetaChar>, <Quote1>,```
	```// and <Quote2> excluded. An<EscapeChar> <MetaChar>, or```
	```// <EscapeChar> <Quote1>, or <EscapeChar> <Quote2> is equivalent to```
	```// <SimpleChar>.```

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

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

XFN Decomposing the Composite Name String

The function `fn_composite_name_from_string()` returns an XFN composite name in its structural form, `FN_composite_name_t`, given the composite name's string representation. The syntax rules used by `fn_composite_name_from_string()` are as follows:

- An XFN 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:

   a. The first component is the string preceding the first occurrence of a component separator.

   b. Empty components are processed as follows:

      i. A leading component separator (the composite name begins with a component separator) means a leading null component.

      ii. A trailing component separator (the composite name ends with a component separator) means a trailing null component.

   c. Two consecutive component separators mean a null component.

   d. The name string that immediately follows the last component separator of the composite name is the final component.

4. A component string is evaluated from left to right and converted into its standard form, according to the following rules:

   a. 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.

   b. 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 treated just as any other *<SimpleChar>*.

   c. An unmatched begin quote (missing or misplaced end quote) fails with an FN_E_ILLEGAL_NAME status.

   d. Quotes are considered to be escaped in quoted strings if a matching quote character is preceded immediately by the unescaped *<EscapeChar>*.

   e. 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 nonmatching quote characters enclosed in quoted strings are treated just as any other *<SimpleChar>*.

   f. Any of the defined metacharacters (*<ComponentSep>* and *<EscapeChar >*) 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>*).

   g. *<Quote1>* and *<Quote2>* are considered to be escaped in an unquoted component if and only if *EscapeChar>* is preceded by a component separator

(that is, sequences *<ComponentSep>* *<EscapeChar>* *<Quote1>* or *<ComponentSep>* *<EscapeChar>* *<Quote2>*). Other occurrences of *<Quote1>* and *<Quote2>* in an unquoted component are treated just as any other *<SimpleChar>*.

h. Any occurrence of escaped <MetaChar>, escaped *<Quote1>*, or escaped *<Quote2>* in unquoted components is substituted by the corresponding unescaped character.

i. No substitution is done for *<EscapeChar> SimpleChar>*. *<EscapeChar> SimpleChar>* maps to *<EscapeChar>* *<SimpleChar>*.

XFN Composing the Composite Name String

The function `fn_string_from_composite_name()` returns the string representation of an XFN composite name given its structural form (`FN_composite_name_t`). The following are the rules used by `fn_string_from_composite_name()`.

1. The components are added to the composite name string in left to right order (that is, rightmost is the tail).

2. Successive components are separated by the component separator (*<ComponentSep>*).

3. Empty components are handled in the following way:

   a. A leading empty component is represented by a leading *<ComponentSep>*.

   b. A trailing empty component is represented by a trailing *<ComponentSep>*.

   c. 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*.

Glossary

application-level name service	Application-level name services are incorporated in applications offering services such as files, mail, and printing. Application-level name services are bound below enterprise-level name services. The enterprise-level name services provide contexts in which contexts of application-level name services can be bound.
atomic name	An indivisible component of a name as defined by the naming convention.
attribute	Each named object is associated with a set of zero or more attributes. Each attribute in the set has a unique attribute identifier, an attribute syntax, and a set of zero or more distinct attribute values.
binding	The association of an atomic name with an object reference. For simplicity, an object reference and the object it refers to are used interchangeably in this guide.
BNF	Backus-Naur Form.
composite name	A name that spans multiple naming systems. It consists of an ordered list of zero or more components. Each component is a name from the namespace of a single naming system. Composite name resolution is the process of resolving a name that spans multiple naming systems.
compound name	A sequence of atomic names composed according to the naming convention of a naming system.
context	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, and may provide operations such as binding names, unbinding names, and listing bound names.
DNS	Domain Name System. A system that provides the naming policy and mechanisms for mapping domain and machine names to addresses on the Internet.

enterprise-level name service	A name service that names objects within an enterprise. The types of objects named are organizational units, sites, users, hosts, and files. Enterprise-level name services are bound below global name services. Global name services provide contexts in which the root contexts of enterprise-level naming systems can be bound.
enterprise root	The root context of an enterprise. A context for naming objects found at the root of the enterprise namespace.
federated naming service	The service offered by a federated naming system.
federated naming system	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.
federated namespace	The set of all possible names generated according to the policies that govern the relationships among member naming systems and their respective namespaces.
generic context	A context for binding names used in applications.
global context	A context for naming objects that have global names (currently, DNS and X.500 are the only global naming systems specified by XFN).
global name service	A name service that has worldwide scope, such as Internet DNS and X.500. The types of entities named at this global level are typically countries, states, provinces, cities, companies, universities, institutions, and government departments and ministries. Each of these entities can be an enterprise.
host context	A context for naming objects related to a computer.
implicit naming system pointer	An unnamed reference that points to a context in another naming system.
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.
junction	A name in one namespace bound to a context in the next naming system.
naming convention	Every name is generated by a set of syntactic rules called a naming convention.
namespace	The set of all names in a naming system.
namespace identifier	A special atomic name used to refer to the root of a namespace.

name service	The service offered by a naming system. It is accessed through its interface.
naming system	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 the UNIX operating system, for example, the set of directories in a given file system (and the naming operations on directories) constitutes a naming system.
next naming system pointer (NNSP)	Reference to a context in which composite names from subordinate naming systems are resolved.
organizational units	An enterprise is organized into organizational units such as centers, laboratories, departments, divisions, and so on. An organizational unit is a subunit of an enterprise.
organizational unit context	A context for naming objects related to an organizational unit within an enterprise.
parent context	A context in which this context and its siblings are bound.
reference	The thing bound to a name. It contains addresses identifying the communication endpoints of the object.
root context	A context for naming the objects found in the root of the namespace.
service context	A context for naming objects that provide services.
site context	A context for naming objects related to a physical site.
strong separation	The case where the XFN context treats the XFN component separator as the naming system boundary.
subcontext	A context bound within another context.
user context	A context for naming objects related to a human user.
weak separation	The case where the XFN context does not treat the XFN component separator as the naming system boundary.
XFN link	A special form of reference that has a composite name as an address. Like any other type of reference, an XFN link is bound to an atomic name in a context.
X.500	A global-level directory service defined by an Open Systems Interconnection (OSI) standard. The X.500 directory service is not supported in a 64–bit application.

Index

applications *(continued)*
    attributes,   22
    composite names,   23, 25
    compound names,   22, 23
    contexts,   21
    initial context,   25
    references,   21
    XFN links,   24
as XFN component escape character,   83
ASCII string XFN identifier format,   56
asterisk (*) in BNF notation,   90, 96
atomic names
    in compound names,   22
    in contexts,   21
attribute operations
    attribute-modification operations,   44
    get attribute,   44
    get attribute identifiers,   46
    get attribute values,   45
    get multiple attributes,   46, 47
    modify attribute,   44
    modify multiple attributes,   47
    multiple-attribute operations,   46, 48, 51
    relationship to naming operations,   43
    single-attribute operations,   44, 46
    status objects,   43
    XFN attribute model,   42
attribute-modification lists,   57
attributes
    adding attributes or values,   44
    base attribute interface,   42, 48, 51
        attribute-modification operations,   44
        multiple-attribute operations,   46, 48, 51
        relationship to naming operations,   43
        single-attribute operations,   44, 46
        status objects,   43
        supporting interfaces,   31, 32
        XFN attribute model,   42
    described,   22
    getting,   44
        identifiers,   46
        multiple attributes,   46, 47
        syntax attributes of context,   41
        values,   45
    sets,   57
    syntax attributes,   64, 65
        getting,   41

attributes, getting *(continued)*
    XFN interface parameters,   57
    XFN model,   42

B
backslash () as XFN component escape
    character,   83
Backus-Naur Form (BNF),   90, 91, 96, 97
base attribute interface,   42, 48, 51
    abstract data types,   32
    attribute-modification operations,   44
    conventions,   32
    memory management policies,   33
    multiple-attribute operations,   46, 48, 51
    parameters,   55, 63
        attribute modification lists,   57
        attribute sets,   57
        attributes and attribute values,   57
        composite names,   55
        identifiers,   56
        references and addresses,   56
        strings,   57
    parsing compound names,   63
        syntax attributes,   64, 65
        XFN standard syntax model,   64, 65
    relationship to naming operations,   43
    single-attribute operations,   44, 46
    status objects,   43
    supporting interfaces,   31, 32
    usage,   32
    XFN attribute model,   42
base context interface,   33
    abstract data types,   32
    context handles,   35, 36
    conventions,   32
    lookup and list contexts,   36, 38
    managing contexts,   40
    memory management policies,   33
    names in context operations,   34
    other context operations,   40, 41
    parameters,   55, 63
        attribute modification lists,   57
        attribute sets,   57
        attributes and attribute values,   57
        composite names,   55

messages, *See* error messages
modules, context shared object,   27
multiple addresses,   21
multiple attributes
  getting,   46, 47
  getting identifiers,   46
  modifying,   47
myens or _myens namespace identifier
  initial context binding,   19
myorgunit or _myorgunit namespace identifier
  initial context binding,   19
myself or _myself namespace identifier
  initial context binding,   19

N
name resolution
  context operation support requirements,   34,
    35
  status object information,   52
  XFN composite names,   87, 89
    coexistence of explicit and implicit
      NNSPs,   89
    explicit NNSPs,   87
    implicit NNSPs,   88
    XFN links,   89
namespace browser programming example,   70
  code,   70
  commands,   76
  diagram,   70
  sample output,   76
namespace identifiers
  enterprise level
    initial context bindings,   19
namespace policies, *See* policies
naming
  context operation names,   34
  XFN attribute operations and,   43
  XFN interface conventions,   32
naming system boundaries and component
  separators,   85, 86
  strong separation,   85
  weak separation,   86

navigating, *See* browsing
next naming system pointers (NNSPs)
  XFN composite name resolution
    coexistence of explicit and implicit
      NNSPs,   89
    explicit NNSPs,   87
    implicit NNSPs,   88
NNSPs, *See* next naming system pointers
  (NNSPs)

O
operations, *See* attribute operations
org namespace identifier
  initial context binding,   19
organizational units
  composite name examples,   20
  described,   16
  enterprise namespace policies,   13
orgunit or _orgunit namespace identifier
  FNS policy,   13
  initial context binding,   19
OSF DCE UUID XFN identifier format,   56

P
parsing
  compound names,   63
    syntax attributes,   64, 65
    XFN standard syntax model,   64, 65
  XFN composite names,   91, 93, 97, 99
periods, *See* dots (.)
pipe character (|) in BNF notation,   90, 96
plus sign (+) in BNF notation,   90, 96
pointer types,   33
policies
  enterprise namespace
    arrangement of objects,   16
    illustrated,   15
    initial context bindings,   17
    table of policies,   13
  global namespace
    illustrated,   15
  information not specified,   15
  information specified,   12