Remote Administration Daemon Developer Guide

Part No: E36869
July 2014

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu’elles soient exemptes d’erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l’accompagne, est concédé sous licence au Gouvernement des Etats-Unis, ou à toute entité qui délivre la licence de ce logiciel ou l’utilise pour le compte du Gouvernement des Etats-Unis, la notice suivante s’applique:

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d’applications de gestion des informations. Ce logiciel ou matériel n’est pas conçu ni n’est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d’applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l’utilisation de ce logiciel ou matériel pour ce type d’applications.

Ce logiciel ou matériel et la documentation qui l’accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l’accès à des contenus, produits ou services tiers, ou à leur utilisation.

Contents

Using This Documentation

- **Overview** – Provides information about the remote administration daemon for Oracle Solaris. Applications that allow you to remotely administer or configure a system, require programmatic access. This guide provides information on how to use the remote administration daemon to provide programmatic access to the administration and configuration functionality of the Oracle Solaris operating system.
- **Audience** – This book is intended for developers who want to use rad to create administrative interfaces and for developers looking to consume interfaces published using rad by others.
- **Required knowledge** – Readers of this guide should be experienced in developing JAVA or Phython based application interfaces.

Product Documentation Library

Late-breaking information and known issues for this product are included in the documentation library at http://www.oracle.com/pls/topic/lookup?ctx=E36784.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

♦ ♦ ♦ CHAPTER 1

Introduction

The Remote Administration Daemon, commonly referred to by its acronym and command name, `rad`, is a standard system service that offers secure, remote administrative access to an Oracle Solaris system. This book is intended for developers who want to use `rad` to create administrative interfaces and for developers looking to consume interfaces published using `rad` by others. It introduces `rad` and its core concepts, explains the process of developing `rad` extensions and consumers, and includes a reference for the `rad` module API and client libraries.

Remote Administration Daemon

The Oracle Solaris operating system is a set of cooperating components, and correspondingly administering Oracle Solaris is the act of manipulating those components in a variety of ways. The traditional solution consisted of locally applying $EDITOR to text files. More modern approaches include manipulating system components locally using a CLI or an interactive UI, remotely with a browser or client, *en masse* with an enterprise-scale provisioning tool, or automatically by policy logic employed by any of these methods. All of these methods require programmatic access to configuration. The Remote Administration Daemon is the central point where system developers can expose their components for configuration or administration, and where the various programmatic consumers can go to perform those activities.

To provide complete support to consumers written in a variety of languages, consumers running without privilege, and consumers running remotely, `rad` employs a client/server design. `rad` itself acts as a server that services remote procedure calls. `rad` consumers are the clients. The protocol `rad` speaks is efficient and easy to implement, which makes it simple to bring support for *all* administrative tasks exposed via `rad` to a new language.

By providing a procedure call interface, `rad` enables non-privileged local consumers to perform actions on behalf of their users that require elevated privilege, without needing to resort to a CLI-based implementation. Finally, by establishing a stream protocol, these same benefits can be extended to consumers on any machine or device over a variety of secure transport options.

`rad` differs from traditional RPC in a number of ways:

- Procedure calls are made against server objects in a browsable, structured namespace. This process permits a more natural evolution of content than is afforded by the central allocation of program numbers.
- These procedure calls need not be synchronous. Subject to the protocol in use, a client may have multiple simultaneous outstanding requests.
- The interfaces exported by the server objects are fully inspectable. This facilitates interactive use, rich debugging environments, and clients using dynamically-typed languages such as Python.
- In addition to defining procedure calls, `rad` interfaces can define properties and asynchronous event sources. Though the former provides more of a semantic than a functional improvement, the latter is a powerful tool for efficiently observing changes to the system being managed.

- `rad` supports alternate protocols without needing to update its content, which provides even greater flexibility.
- `rad`'s native protocol fully supports asynchronous procedure calls once the client has authenticated. An alternate protocol, e.g. one based on XML-RPC, might not support asynchronous calls due to limitations of the underlying technology.

Features Overview

The main functionality offered by `rad` is as follows:

- **Essentials**
  - Managed and configured by two SMF services, `svc:/system/rad:local` and `svc:/system/rad:remote`
  - Structured, browsable namespace.
  - Inspectable, typed, versioned interfaces.
  - Asynchronous event sources.
  - XML-based IDL ADR supports formally defining APIs. The IDL compiler `radadrgen` generates client language bindings.
- **Security**
  - Full PAM conversation support including use of `pam_setcred`(3PAM) to set the audit context.
  - Authentication via GSSAPI in deployments where `kerberos`(5) is configured.
  - Implicit authentication using `getpeerucred`(3C) when possible.
  - No non-local network connectivity by default. Preconfigured to use TLS.

  - Most operations automatically delegated to lesser-privileged processes.
  - Defines two authorizations (solaris.smf.manage.rad and solaris.smf.value.rad) and two Rights Profiles (`rad` Management and `rad` Configuration) to provide fine-grained separation of powers for managing and configuring the `rad` SMF services.
    - rad authorizations
      - `solaris.smf.manage.rad` — Grants the authorization to enable, disable, or restart the `rad` SMF services.
      - `solaris.smf.value.rad` — Grants the authorization to change `rad` SMF services' property values.
    - rad rights profiles
      - `rad` Management — Includes the `solaris.smf.manage.rad` authorization.
      - `rad` Configuration — Includes the `solaris.smf.value.rad` authorization.
  - Generates `AUE_rad_login`, `AUE_logout`, `AUE_role_login`, `AUE_role_logout`, and `AUE_passwd` audit events.
- **Connectivity**
  - Local access via AF_UNIX sockets.
  - Remote access via TCP sockets.
  - Secure remote access via TLS sockets.
  - Captive execution with access through a pipe.
  - Connection points are completely configurable at the command line or via SMF.
- **Client support**
  - A Java language binding provides access to all defined server interfaces.
  - A Python language binding provides access to all defined server interfaces.
  - A C language binding provides access to all defined server interfaces.
- **Extension**
  - A public native C module interface supports addition of third-party content.
  - `radadrgen` can generate server-side type definitions and stubs from IDL input.
  - A native execution system can automatically run modules with authenticated user's privilege and audit context, simplifying authentication and auditing.
  - Private module interfaces permit defining new transports.

♦ ♦ ♦ CHAPTER 2

Concepts

The concepts that are fundamental to rad are *interfaces*, objects that implement those interfaces, and the namespace in which those objects can be found and operated upon.

This chapter discusses the following concepts that are fundamental to rad.

- "API" on page 13
- "Interface" on page 14
- "rad Namespace" on page 19
- "Data Typing" on page 20

API

An API is the starting point for designing a new RAD component. An API consists of a collection of other subsidiary components: derived types and interfaces. An API is versioned so that a client can specify which version of an API to interact with.

The API acts as the name root for all components of the API, defining a namespace which identifies objects to client. APIs are versioned and a single RAD instance is capable of offering multiple major versions of APIs to different clients.

Version

A version element is required for all APIs. See "Versioning" on page 17 for more details on API versions.

API Namespace and Restricted Names

An API defines a namespace in which all top-level elements are defined. Names of components must be unique. Names must not begin with "_rad", since this is reserved for toolchain provided functionality.

Derived Types

Two classes of derived types may be defined for use within an API: structures and enumerations. Structures are used to introduce new types specific to the API. Enumerations are used to restrict numeric values to a legal specified range.

Interface

An interface defines how a `rad` client can interact with an object. An object implements an interface, providing a concrete behavior to be invoked when a client makes a request.

The primary purpose of `rad` is to consistently expose the various pieces of the system for administration. Not all subsystems are alike, however: each has a data and state model tuned to the problems they are solving. Although there are major benefits to using a common model across components when possible, uniformity comes with trade-offs. The increased inefficiency and client complexity, and risk of decreased developer adoption, often warrant using an interface designed for problem at hand.

An interface is a formal definition of how a client may interact with a `rad` server object. An interface may be shared amongst several objects, for example, when maintaining a degree of uniformity is possible and useful, or may be implemented by only one. A `rad` interface is analogous to an interface or pure abstract class in an object oriented programming language. In the case of `rad`, an interface consists of a name, the set of features a client may interact with, optionally a set of derived types referenced by the features, and a version. The features supported include:

- Methods, which are procedure calls made in the context of a specific object
- Properties, which are functionally equivalent to methods but bear different semantics
- Asynchronous event sources

Name

Each interface has a name. This name is used by the toolchain to construct identifier names when generating code.

Features

The common thing between the three feature types — methods, attributes, and events — is that they are named. All three feature types name exist in the same Interface namespace and must therefore be unique. You can not have both a method and an attribute called “foo”. This exclusion avoids the majority of conflicts that could arise when trying to naturally map these interface features to a client environment. As in the API namespace, features must not begin with “`_rad`”, since this is reserved for use by the RAD toolchain.

Note - Enforcing a common namespace for interface features isn't always enough. Some language environments place additional constraints on naming. For instance, a Java client will see an interface with synthetic methods of the form `get`*function_name*, `set`*function_name*, or `is`*function_name* for accessing attribute *function_name* that must coexist with other method names. Explicitly defining methods with those names may cause a conflict.

Methods

A method is a procedure call made in the context of the object it is called on. In addition to a name, a method may define a return type, can define zero or more arguments, and may declare that it returns an error, optionally with an error return type.

If a method does not define a return type, it returns no value. It is effectively of type void. If a method defines a return type and that type is permitted to be nullable, the return value may be defined to be nullable.

Each method argument has a name and a type. If any argument's type is permitted to be nullable, that argument may be defined to be nullable.

If a method does not declare that it returns an error, it theoretically cannot fail. However, because the connection to `rad` could be broken either due to a network problem or a catastrophic failure in `rad` itself, all method calls can fail with an I/O error. If a method declares that it returns an error but does not specify a type, the method may fail due to API-specific reasons. Clients will be able to distinguish this failure type from I/O failures.

Finally, if a method also defines an error return type, data of that type may be provided to the client in the case where the API-specific failure occurs. Error payloads are implicitly optional, and must therefore be of a type that is permitted to be nullable.

Note - Methods names may not be overloaded.

Attributes

An attribute is metaphorically a property of the object. Attributes have the following characteristics:

- A name
- A type
- A definition as read-only, read-write, or write-only
- Like a method may declare that accessing the attribute returns an error, optionally with an a error return type

Reading a read-only or read-write attribute returns the value of that attribute. Writing a write-only or read-write attribute sets the value of that attribute. Reading a write-only attribute or writing a read-only attribute is invalid. Clients may treat attempts to write to a read-only attribute as a write to an attribute that does not exist. Likewise, attempts to read from a write-only attribute may be treated as an attempt to read from an attribute that does not exist.

If an attribute's type is permitted to be nullable, its value may be defined to be nullable.

An attribute may optionally declare that it returns an error, with the same semantics as declaring (or not declaring) an error for a method. Unlike a method, an attribute may have different error declarations for reading the attribute and writing the attribute.

Attribute names may not be overloaded. Defining a read-only attribute and a write-only attribute with the same name is not valid.

Given methods, attributes are arguably a superfluous interface feature. Writing an attribute of type X can be implemented with a method that takes one argument of type X and returns nothing, and reading an attribute of type X can be implemented with a method that takes no arguments and returns a value of type X. Attributes are included because they have slightly different semantics.

In particular, an explicit attribute mechanism has the following characteristics:

- Enforces symmetric access for reading and writing read-write attributes.
- Can be easily and automatically translated to a form natural to the client language-environment.
- Communicates more about the nature of the interaction. Reading an attribute ideally should not affect system state. The value written to a read-write attribute should be the value returned on subsequent reads unless an intervening change to the system effectively *writes* a new value.

Events

An event is an asynchronous notification generated by `rad` and consumed by clients. A client may subscribe to events by name to register interest in them. The subscription is performed on an object which implements an interface. In addition to a name, each event has a type.

Events have the following characteristics:

- Sequential

- Volatile
- Guaranteed

A client can rely on sequential delivery of events from a server as long as the connection to the server is maintained. If the connection fails, then events will be lost. On reconnection, a client must resubscribe to resume the flow of events.

Once a client has subscribed to an event, event notifications will be received until the client unsubscribes from the event.

On receipt of a subscribed event, a client receives a payload of the defined type.

Commitment

To solve the problem of different features being intended for different consumers, rad defines two commitment levels: private, and committed. All API components: derived types, interfaces and the various interface sub-components (method, attribute, and event) define their commitment level independently.

Commitment levels provide hints to API consumers about the anticipated use and expected stability of a feature. A feature with a commitment of *committed* can be used reliably. The *private* features, are likely to be subject to change and represent implementation details not intended for public consumption.

Versioning

`rad` interfaces are versioned for the following reasons:

- APIs change over time.
- A change to an API might be incompatible with existing consumers.
- A change might be compatible with existing consumers but new consumers might not be able to use the API that was in place before the change occurred.
- Some features represent committed interfaces whose compatibility is paramount, but others are private interfaces that are changed only in lockstep with the software that uses them.

Numbering

The first issue is measuring the compatibility of a change. `rad` uses a simple `major.minor` versioning scheme. When a compatible change to an interface is made, its minor version number is incremented. When an incompatible change is made, its major version number is incremented and its minor version number is reset to 0.

In other words, an implementation of an interface that claims to be version X.Y (where X is the major version and Y is the minor version) must support any client expecting version X.Z, where Z <= Y.

The following interface changes are considered compatible:

- Adding a new event
- Adding a new method
- Adding a new attribute
- Expanding the access supported by an attribute, for example, from read-only to read-write
- A change from nullable to non-nullable for a method return value or readable property, that is, decreasing the range of a feature
- A change from non-nullable to nullable for a method argument or writable property, that is, increasing the domain of a feature

The following interface changes are considered incompatible:

- Removing an event
- Removing a method
- Removing an attribute
- Changing the type of an attribute, method, or event
- Changing a type definition referenced by an attribute, method, or event
- Decreasing the access supported by an attribute, for example, from read-write to read-only
- Adding or removing method arguments
- A change from non-nullable to nullable for a method return value or readable property, that is, increasing the range of a feature
- A change from nullable to non-nullable for a method argument or writable property, that is, decreasing the domain of a feature

---

Note - An interface is more than just a set of methods, attributes, and events. Associated with those features are well-defined behaviors. If those behaviors change, even if the structure of the interface remains the same, a change to the version number might be required.

Clients and Versioning

A `rad` client can access version information from a client binding. The mechanism for accessing the information depends on the client language like C, Java and, Python. For example in

Python, the rad.client module contains the rad_get_version function which may be used to get the version of an API.

rad Namespace

The namespace acts as rad's gatekeeper, associating a name with each object, dispatching requests to the proper object, and providing meta-operations that enable the client make queries about what objects are available and what interfaces they implement.

A rad server may provide access to several objects that in turn expose a variety of different components of the system or even third-party software. A client merely knowing that interfaces exist, or even that a specific interface exists, is not sufficient. A simple, special-purpose client needs some way to identify the object implementing the correct interface with the correct behavior, and an adaptive or general-purpose client needs some way to determine what functionality the rad server has made available to it.

rad organizes the server objects it exposes in a namespace. Much like files in a file system, objects in the rad namespace have names that enable clients to identify them, can be acted upon or inspected using that name, and can be discovered by browsing the namespace. Depending on the point of view, the namespace either is the place one goes to find objects or the intermediary that sits between the client and the objects it accesses. Either way, it is central to interactions between a client and the rad server.

Naming

Unlike a file system, which is a hierarchical arrangement of simple filenames, rad adopts the model used by JMX and maintains a flat namespace of structured names. An object's name consists of a mandatory reverse-dotted domain combined with a non-empty set of key-value pairs.

Equality

Two names are considered equal if they have the same domain and the same set of keys, and each key has been assigned the same value.

Patterns

Some situations call for referring to groups of objects. In these contexts, a glob style pattern, or a regex style pattern should be used. For more information, see "Sophisticated Searching" on page 27.

Data Typing

All data returned submitted to or obtained from `rad` APIs adheres to a strong typing system similar to that defined by XDR. For more information about XDR, see the XDR (http://tools.ietf.org/rfc/rfc4506.txt) standard. This makes it simpler to define interfaces that have precise semantics, and makes server extensions (which are written in C) easier to develop. Of course, the rigidity of the typing exposed to an API's consumer is primarily a function of the client language and implementation.

Base Types

`rad` supports the following base types:

`boolean`	A boolean value (true or false).
`integer`	A 32-bit signed integer value.
`uinteger`	A 32-bit unsigned integer value.
`long`	A 64-bit signed integer value.
`ulong`	A 64-bit unsigned integer value.
`float`	A 32-bit floating-point value.
`double`	A 64-bit floating-point value.
`string`	A UTF-8 string.
`opaque`	Raw binary data.
`secret`	An 8-bit clean “character” array. The encoding is defined by the interface using the type. Client/server implementations may take additional steps, for example, zeroing buffers after use, to protect the contents of secret data.
`time`	An absolute UTC time value.
`name`	The name of an object in the `rad` namespace.
`reference`	A reference to an object.

Derived Types

In addition to the base types, `rad` supports several derived types.

An enumeration is a set of user-defined tokens. Like C enumerations, `rad` enumerations may have specific integer values associated with them. Unlike C enumerations, `rad` enumerations and integers are not interchangeable. Among other things, this aspect means that an enumeration data value may not take on values outside those defined by the enumeration, which precludes the common but questionable practice of using enumerated types for bitfield values.

An array is an ordered list of data items of a fixed type. Arrays do not have a predefined size.

A structure is a record consisting of a fixed set of typed, uniquely named fields. A field's type may be a base type or derived type, or even another structure type.

Derived types offer almost unlimited flexibility. However, one important constraint imposed on derived types is that recursive type references are prohibited. Thus, complex self-referencing data types, for example, linked lists or trees, must be communicated after being mapped into simpler forms.

Optional Data

In some situations, data may be declared as nullable. Nullable data can take on a “non-value”, for example, `NULL` in C, `None` in Python, or `null` in Java. Conversely, non-nullable data cannot be `NULL`. Only data of type opaque, string, secret, array, or structure may be declared nullable. Additionally, only structure fields and certain API types can be nullable. Specifically, array data cannot be nullable because the array type is actually more like a list than an array.

♦ ♦ ♦ CHAPTER 3

Client Libraries

rad provides support for three client language environments: C, Java, and Python.

C Client

The general (non module specific) public interfaces are exported in the library /usr/lib/libradclient.so and defined in the headers

- /usr/include/rad/radclient.h – The client function and datatype definitions.
- /usr/include/rad/radclient_basetypes.h – Helper routines for managing the built-in rad types.

The examples shown below are just snippets of C code, but at the top of each example for clarity, there is a list of #include statements showing the headers needed for that specific functionality.

Note - A lot of these examples are based on the example zonemgr interface. Refer to the sample in, Appendix A, “zonemgr ADR Interface Description Language” for this module to assist in your understanding of the examples.

Connecting to RAD

Communication with a RAD instance is provided through the rc_connect_* family of functions. There are various functions to get different types of connections to RAD. Each function returns a rc_conn_t reference which acts as a handle for interactions with RAD over that connection. Every connect function has two common arguments: a boolean to specify whether the connection be multithreaded (TRUE is recommended) and the locale to use for the connection. When locale is NULL, the locale of the local client is used. Each connection type also has differing transport specific arguments.

To close the connection,rc_disconnect needs to be called with the connection handle.

Connecting to a Local Instance

Implicit authentication is performed against your user id and most RAD tasks you request with this connection are performed with the privileges available to your user account. The function `rc_connect_unix` takes three arguments: The path to the unix socket, a boolean to determine if the connection is to be multithreaded, and the desired locale for the connection. If the socket path is `NULL`, the default RAD unix socket path is used; it is recommended that the connection be run in multithreaded mode; if the locale is `NULL`, the locale of the local client system is used.

EXAMPLE 3-1 Creating a Local Connection

```
#include <rad/radclient.h>
rc_conn_t conn = rc_connect_unix(NULL, B_TRUE, NULL);

// do something with conn
```

Connecting to a Remote Instance and Authenticating

When connecting to a remote instance, there is no implicit authentication, so the connection is not useful until authentication is performed. To authenticate with PAM, a function `rc_pam_login` is provided. However, the client application source must `#include <rad/client/1/pam_login.h>` and link against the PAM C binding library, `/usr/lib/rad/client/c/libpam_client.so`.

Authentication is done non-interactively and a username and password should be provided. Optionally, a handle to the PAM Authentication object can be returned if a reference is provided as the second argument to `rc_pam_login`.

EXAMPLE 3-2 Remote Connection over TCP IPv4 on port 7777

```
#include <rad/radclient.h>
#include <rad/client/1/pam_login.h>

rc_instance_t *pam_inst;
rc_conn_t conn = rc_connect_tcp("henry",7777, B_TRUE, NULL);

if (conn !=NULL) {
            rc_err_t status = rc_pam_login(conn, &pam_inst, "user", "password");
            if (status == RCE_OK){
                printf("Connected and authenticated!\n");
                }
}
```

Rad Namespace

All RAD objects, which are represented in the ADR IDL as <*interfaces*>, are found by searching the RAD namespace. The key point to note is that to access a RAD object, you should use the list and lookup functions provided by a module's client binding library (`<module>_<interface>__rad_list`, `<module>_<interface>__rad_lookup`. These functions also provide the option to do either strict or relaxed versioning.

Using the functions specific to the interface automatically provides the base name and version details for interface instances. Those names are structured as follows:

```
<domain name>:type=<interface name>[,optional additional key value pairs]
```

The <domain name> and the <interface name> are automatically derived from the ADR IDL definition and are stored in the module binding.

Searching for Objects

As noted in the previous section, a module's client binding provides a search function for each interface defined in the form: `<module>_<interface>__rad_list` where a pattern (glob or regex) may be provided to narrow the search further within objects of a certain interface type.

In addition, libradclient does provide a function, `rc_list`, where the caller can provide the entire name/pattern and version information to search for.

Obtaining a Reference to a Singleton

If a module developer creates a “singleton” to represent an interface, then this can be accessed very simply. For instance, the zonemgr module defines a singleton interface: ZoneInfo. It contains information about the zone which contains the RAD instance with which we are communicating.

EXAMPLE 3-3 Obtaining a Reference to a Singleton

```
#include <rad/radclient.h>
#include<rad/client/1/zonemgr.h>

rc_instance_t *inst;
rc_err_t status;
char *name;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn !=NULL) {
   status = zonemgr_ZoneInfo__rad_lookup(conn, B_TRUE, &inst, 0);
  if(status == RCE_OK) {
    status =zonemgr_ZoneInfo_get_name(inst, &name);
  if (status ==RCE_OK)
     printf("Zone name: %s\n", name);
   }
}
```

We have connected to the local RAD instance and obtained a remote object reference directly using the lookup function provided by the zonemgr binding. Once we have the remote reference we can access properties with a `<module>_<interface>_get_<property>` function.

Listing Instances of an Interface

Most interfaces contain more than one instance of the interface. For instance, the zonemgr module defines a Zone interface and there is an instance for each zone on the system. A module provides a list function for each of its interfaces in the form: `<module>_<interface>__rad_list`.

EXAMPLE 3-4 Listing Interface Instances

```
#include<rad/radclient.h>
#include<rad/radclient_basetypes.h>
#include<rad/client/1/zonemgr.h>

rc_err_t status;
adr_name_t **name_list;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn !=NULL) {
  status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_GLOB, &name_list,
    &name_count, 0);
   if(status == RCE_OK) {
      for (int i =0; i < name_count; i++) {
      char*name =adr_name_tostr(name_list[i]);
        printf("%s\n", name);
      }
     name_array_free(name_list, name_count);
     }
  }
```

Obtaining a Remote Object Reference from a Name

Names (in the form of an adr_name_t reference) are returned when using the list mechanism detailed above. Once we have a "name" we can obtain a remote object reference easily:

EXAMPLE 3-5 Obtaining a Remote Object Reference from a Name

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include<rad/client/1/zonemgr.h>

rc_err_t status;
adr_name_t **name_list;
rc_instance_t *zone_inst;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
        status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_GLOB, &name_list,
          &name_count, 0);
      if (status == RCE_OK) {
               status = rc_lookup(conn, name_list[0],
                  NULL, B_TRUE, &zone_inst);
              if (status == RCE_OK) {
                      char *name;
                       status = zonemgr_Zone_get_name(zone_inst, &name);
                      if (status == RCE_OK)
                               printf("Zone name: %s\n",
                                   name);
                      free(name);
                }
                name_array_free(name_list, name_count);
        }
}
```

Sophisticated Searching

Clearly, the last example is not a very realistic use case. Rarely are we going to want to just pick a (semi-random) zone from a list and interact with it. More often than not we'll be looking for a zone which has a particular name or id or maybe a set of zones where the names all match some kind of pattern. The key idea to bear in mind is that you can extend the use of the “list” functionality to restrict the results. For instance, if zones are uniquely identified by a key: “name”, then we can find a zone with name “test-0” as follows:

EXAMPLE 3-6 Using Glob Patterns

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
adr_name_t **name_list;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
   status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_GLOB, B_TRUE, &name_list,
    &name_count, 1, "name", "test-0");
      if (status == RCE_OK) {
           for (int i = 0; i < name_count; i++) {
                const char *name = adr_name_tostr(name_list[i]);
                printf("%s\n", name);
             }
           name_array_free(name_list, name_count);
        }
}
```

Glob Pattern Searching

We've already seen how we can use glob pattern searching to find a zone with a specific name. We can also use a glob pattern to find zones with wildcard pattern matching. Keys or Values in the pattern may contain "*" which is interpreted as expected. For instance ,if we wanted to find all zones with a name which begins with "test":

EXAMPLE 3-7 Using Glob Patterns with Wildcards

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
adr_name_t **name_list;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
    status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_GLOB, &name_list,
      &name_count, 1, "name", "test*");
     if (status == RCE_OK) {
          for (int i = 0; i < name_count; i++) {
               const char *name = adr_name_tostr(name_list[i]);
               printf("%s\n", name);
             }
             name_array_free(name_list, name_count);
        }
}
```

Regex Pattern Searching

We can also take advantage of RAD's ERE (Extended Regular Expression) search capabilities. If we wanted to find only zones with name "test-0" or "test-1", then:

EXAMPLE 3-8 Using Regex Patterns

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>
```

```
rc_err_t status;
adr_name_t **name_list;
int name_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
      status = zonemgr_Zone__rad_list(conn, B_TRUE, NS_REGEX,
    &name_list, &name_count, 1, "name", "test-0|test-1");
    if (status == RCE_OK) {
         for (int i = 0; i < name_count; i++) {
              const char *name = adr_name_tostr(name_list[i]);
              printf("%s\n", name);
            }
            name_array_free(name_list, name_count);
        }
}
```

The key and the value must be valid Extended Regular Expressions as determined by the instance of RAD to which you are connected. This means that the expression is compiled and executed in the server.

Interface Components

An API is defined by a module developer and contains a variety of components designed to accomplish a task. These components are:

- Enums
  - Values
- Structs
  - Fields
- Interfaces
  - Properties
  - Methods
  - Events

These components are all defined in an ADR Interface Description Language document. The `radadrgen` utility is used to process the document to generate language specific components which facilitate client/server interactions within RAD. More details in the role of ADR and `rad` can be found in Chapter 4, “Abstract Data Representation”. Brief descriptions of each component follows.

Enumerations

Enumerations are primarily used to offer a restricted range of choices for a property, an interface method parameter, result, or error.

Using Enumeration Types

Enumerated types are defined in the binding header with the type prepended with the module name. Values are prepended to follow (as closely as possible) the C coding standard naming conventions.

EXAMPLE 3-9 zonemgr ErrorCode Enumeration

```
typedef enum zonemgr_ErrorCode {
     ZEC_NONE =0,
     ZEC_FRAMEWORK_ERROR = 1,
     ZEC_SNAPSHOT_ERROR = 2,
     ZEC_COMMAND_ERROR = 3,
     ZEC_RESOURCE_ALREADY_EXISTS = 4,
     ZEC_RESOURCE_NOT_FOUND = 5,
     ZEC_RESOURCE_TOO_MANY = 6,
     ZEC_RESOURCE_UNKNOWN = 7,
     ZEC_ALREADY_EDITING = 8,
     ZEC_PROPERTY_UNKNOWN = 9,
     ZEC_NOT_EDITING = 10,
     ZEC_SYSTEM_ERROR = 11,
     ZEC_INVALID_ARGUMENT = 12,
     ZEC_INVALID_ZONE_STATE = 13,
}zonemgr_ErrorCode_t;
```

Structs

Structs are used to define new types and are composed from existing built-in types and other user defined types. In essence, they are simple forms of interfaces: no methods or events and they are not present in the RAD namespace.

Using Struct Types

The zonemgr module defines a Property struct which represents an individual Zone configuration property. The structure has the following members: name, type, value, listValue, and complexValue. Like enumerations, structures are defined in the binding header and follow similar naming conventions.

To free a structure, free functions (`<module>_<structure>_free`) are provided by the binding to ensure proper cleanup of any memory held by nested data.

EXAMPLE 3-10 The zonemgr Property Struct Definition and its Free Function

```
typedef enum zonemgr_PropertyValueType {
     ZPVT_PROP_SIMPLE = 0,
     ZPVT_PROP_LIST = 1,
     ZPVT_PROP_COMPLEX = 2,
} zonemgr_PropertyValueType_t;

typedef struct zonemgr_Property {
 char * zp_name;
 char * zp_value;
 zonemgr_PropertyValueType_t zp_type;
 char * * zp_listvalue;
 int zp_listvalue_count;
 char * * zp_complexvalue;
 int zp_complexvalue_count;
} zonemgr_Property_t;

void zonemgr_Property_free(zonemgr_Property_t *);
```

Interfaces/Objects

Interfaces (also known as objects) are the entities which populate the RAD namespace. They must have a "*name*". An interface is composed of Events, Properties and Methods.

Obtaining an Object Reference

See the "Rad Namespace" on page 25 section.

Working with Object References

Once we have an object reference we can use this to interact with RAD in a very straightforward fashion. All attributes and methods defined in the IDL are accessible by invoking calling functions in the generated client binding.

Here is an example which gets a reference to a zone and then boots the zone.

EXAMPLE 3-11 Working with Object References

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
rc_instance_t *zone_inst;
zonemgr_Result_t *result;
zonemgr_Result_t *error;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
     status = zonemgr_Zone__rad_lookup(conn, B_TRUE, &zone_inst, 1, "name", "test-0");
     if (status == RCE_OK) {
          status = zonemgr_Zone_boot(zone_inst, NULL, 0, &result, &error);
          rc_instance_rele(zone_inst);
```

```
      }
}
```

Accessing a Remote Property

Here is an example for accessing a remote property.

EXAMPLE 3-12 Accessing a Remote Property

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
rc_instance_t *zone_inst;
char *name;
zonemgr_Property_t *result;
zonemgr_Result_t *error;
int result_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
         status = zonemgr_Zone__rad_lookup(conn, B_TRUE, &zone_inst, 1, "name", "test-0");
         if (status == RCE_OK) {
            zonemgr_Resource_t global = { .zr_type = "global"};
            status = zonemgr_Zone_getResourceProperties(zone_inst, &global, NULL, 0, &result,
 &result_count, &error);
            if (status == RCE_OK) {
                    for (int i = 0; i < result_count; i++){
                            if (result[i].zp_value != NULL && result[i].zp_value[0] != '\0')
                                    printf("%s=%s\n", result[i].zp_name, result[i].zp_value);
                    }
                    zonemgr_Property_array_free(result, result_count);
            }
            rc_instance_rele(zone_inst);
         }
}
```

In this example, we accessed the list of `Zone` global resource properties and printed out the name and value of every Property that has a value.

RAD Event Handling

In this next example we are going to look at events. The ZoneManager instance defines a "stateChange" event which clients can subscribe to for information about changes in the runtime state of a zone.

EXAMPLE 3-13 Subscribing and Handling Events

```
#include <unistd.h>
```

```
#include <time.h>
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

void stateChange_handler(rc_instance_t *inst, zonemgr_StateChange_t *payload, struct timespec
 timestamp, void *arg)
{
    printf("event: zone state change\n");
    printf("payload:\n zone: %s\n old state: %s\n new state: %s\n",
    payload->zsc_zone, payload->zsc_oldstate, payload->zsc_newstate);

    zonemgr_StateChange_free(payload);
}

rc_err_t status;
rc_instance_t *zm_inst;
int result_count;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
        status = zonemgr_ZoneManager__rad_lookup(conn, B_TRUE, &zm_inst, 0);
        if (status == RCE_OK) {
            status = zonemgr_ZoneManager_subscribe_stateChange(zm_inst, stateChange_handler, NULL);
            if (status == RCE_OK)
                printf("Successfully subscribed to statechange event!\n");
            rc_instance_rele(zm_inst);
        }
 }
 for (;;)
 sleep(1);
```

This is a simple example. We subscribe to the single event and pass in a handler and a handle for our ZoneManager object. The handler will be invoked asynchronously by the framework with the various event details and the supplied user data (the user data in this case being `NULL`).

RAD Error Handling

Finally, a quick look at error handling when manipulating remote references. The list of possible errors is defined by the enum `rc_err_t`. There are a variety of errors which can be delivered by RAD, but the one which potentially requires additional handling is the `rc_err_t` value `RCE_SERVER_OBJECT`. The following snippet, shows how it can be used:

EXAMPLE 3-14 Handling RAD Errors

```
#include <rad/radclient.h>
#include <rad/radclient_basetypes.h>
#include <rad/client/1/zonemgr.h>

rc_err_t status;
rc_instance_t *zone_inst;
zonemgr_Result_t *result;
zonemgr_Result_t *error;

rc_conn_t *conn = rc_connect_unix(NULL, B_TRUE, NULL);
if (conn != NULL) {
      status = zonemgr_Zone__rad_lookup(conn, B_TRUE, &zone_inst, 1, "name", "test-0");
     if (status == RCE_OK) {
             status = zonemgr_Zone_boot(zone_inst, NULL, 0, &result, &error);
            if (status == RCE_SERVER_OBJECT) {
                 printf("Error Code %d\n", error->zr_code);
                    if (error->zr_stdout != NULL)
                         printf("stdout: %s\n", error->zr_stdout);
                     if (error->zr_stderr != NULL)
                         printf("stderr: %s\n", error->zr_stderr);
                      zonemgr_Result_free(error);
                }
                rc_instance_rele(zone_inst);
        }
}
```

Note - With an `rc_err_t` value of `RCE_SERVER_OBJECT` you may get a payload. This is only present if your interface method or property has defined an "error" element, in which case the payload is the content of that error. If there is no "error" element for the interface method (or property), then there is no payload and there will be no error reference argument for the method or property get/set functions.

Java Client

The public Java interfaces are exported in two packages:

- com.oracle.solaris.rad.client – The client implementation of the RAD protocol plus associated functionality
- com.oracle.solaris.rad.connect – Classes for connecting to a RAD instance

Note - A lot of these examples are based on the example zonemgr interface. Refer to the sample in, Appendix A, “`zonemgr` ADR Interface Description Language” for this module to assist in your understanding of the examples.

Connecting to RAD

Communication with a RAD instance is provided through the Connection class. There are various factory interfaces to get different types of connections to a RAD instance. Each mechanism returns a Connection instance which provides a standard interface to interact with RAD. The connection can be closed with the `close` method.

Connecting to a Local Instance

Implicit authentication is performed against your user id and most RAD tasks you request with this connection are performed with the privileges available to your user account.

EXAMPLE 3-15 Creating a Local Connection

```
import com.oracle.solaris.rad.connect.Connection;

Connection con = Connection.connectUnix();

//do something with con
```

Connecting to a Remote Instance and Authenticating

When connecting to a remote instance, there is no implicit authentication, so the connection is not useful until authentication is performed. The `com.oracle.solaris.rad.client` package provides a utility class (`RadPamHandler`) which may be used to perform a PAM login. If you supply a locale, username and password, authentication proceeds in a non-interactive fashion. If locale is null, then "C" is used.

Here is an example for Remote Connection to a TCP instance on port 7777

EXAMPLE 3-16 Remote Connection to a TCP instance on port 7777

```
import com.oracle.solaris.rad.client.RadPamHandler;
import com.oracle.solaris.rad.connect.Connection;

Connection con = Connection.connectTCP("henry", 7777);
System.out.println("Connected: " + con.toString());
RadPamHandler hdl = new RadPamHandler(con);
hdl.login("C", "user", "password"); // First argument is locale
con.close();
```

Rad Namespace

All RAD objects, which are represented in the ADR IDL as `<interfaces>`, are found by searching the RAD namespace. The key point to note is that to access a RAD object, you need a proxy, which is used to search the RAD namespace. This capability is provided by an interface proxy class, which is defined in each interface's binding module.

The proxy provides the base name (and version details) for interface instances and is structured as follows:

```
<domain name>:type=<interface name>[,optional additional key value pairs]
```

The `<domain name>` and the `<interface name>` are automatically derived from the ADR IDL definition and are stored in the module binding.

Searching for Objects

The `Connection` class provides mechanisms for listing objects by name and for obtaining a remote object reference.

Obtaining Reference to a Singleton

If a module developer creates a "singleton" to represent an interface, then this can be accessed very simply. For instance, the zonemgr module defines a singleton interface: ZoneInfo. It contains information about the zone which contains the RAD instance with which we are communicating.

In Java we need to compile our code with the language binding in our CLASSPATH. RAD Java Language bindings are in the:

`system/management/rad/client/rad-java` package.

The JAR files for the various bindings are installed in `/usr/lib/rad/java`. Each major interface version is accessible here in a JAR file which is named after the source ADR document and it's major version number. For instance, to access major version 1 of the zonemgr APIs, use `/usr/lib/rad/java/zonemgr_1.jar`. Symbolic links are provided as an indication of the "default" version a client should use.

EXAMPLE 3-17 Obtaining Reference to a Singleton

```
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.ZoneInfo;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());
ZoneInfo zi = con.getObject(new ZoneInfo());
System.out.println("ZoneInfo: " + zi.getname());
```

We have imported ZoneInfo and Connection from our zonemgr binding and the `rad.connect` package, connected to the local RAD instance and obtained a remote object reference directly using a proxy instance. Once we have the remote reference we can access properties and method directly. In the RAD Java implementation all properties are accessed using getter/setter syntax. Thus to access the "name" property, we invoke `getname`.

Listing Instances of an Interface

Most interfaces contain more than one instance of the interface. For instance, the zonemgr module defines a Zone interface and there is an instance for each zone on the system. The `Connection` class provides the `list_objects` method to list interface instances. For instance:

EXAMPLE 3-18 Listing Interface Instances

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

for (ADRName name: con.listObjects(new Zone())) {
    System.out.println("ADR Name: " + name.toString());
}
```

Obtaining a Remote Object Reference from a Name

A list of names (ADRName is the class name) are returned by the `list_objects` method from the `Connection` class. Once we have a "name" we can obtain a remote object reference easily.

EXAMPLE 3-19 Obtaining a Remote Object Reference from a Name

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

for (ADRName name: con.listObjects(new Zone())) {
    Zone zone =  con.getObject(name);
    System.out.println("Name: " + zone.getname());
}
```

Sophisticated Searching

Clearly, the last example is not a very realistic use case. Rarely are we going to want to just pick a (semi-random) zone from a list and interact with it. More often than not we'll be looking for a zone which has a particular name or id or maybe a set of zones where the names all match some kind of pattern. The key idea to bear in mind is that you can extend the basic definition of a name provided by a Proxy. For instance, if zones are uniquely identified by a key: "name", then we can find a zone with name "test-0" as follows:

EXAMPLE 3-20 Using Glob Patterns

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

String keys[] = { "name" };
String values[] = { "test-0" };
ADRGlobPattern pat = new ADRGlobPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    System.out.println("ADR Name: " + name.toString());
}
```

In this example, the `ADRGlobPattern` class (imported from the com.oracle.solaris.rad.client package) is used to refine the search. The `list_objects` method from the `Connection` class is used, but the search is refined by extending the name definition. `ADRGlobPattern` takes an array of keys and an array of values and extends the name used in the search.

Glob Pattern Searching

We've already seen how we can use glob pattern searching to find a zone with a specific name. We can also use a glob pattern to find zones with wildcard pattern matching. Keys or Values in the pattern may contain "*" which is interpreted as expected. For instance if we wanted to find all zones with a name which begins with "test":

EXAMPLE 3-21 Using Glob Patterns with Wildcards

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

String keys[] = { "name" };
String values[] = { "test*" };
ADRGlobPattern pat = new ADRGlobPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    System.out.println("ADR Name: " + name.toString());
}
```

Using Maps when Pattern Searching

It can be more convenient to use Maps rather than arrays of keys/values. Here is the previous example re-worked to use a Map of keys/values rather than arrays of keys/values.

EXAMPLE 3-22 Using Maps with Patterns

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

Map<String, String> kvpairs = new HashMap<String, String>();
kvpairs.put("name", "test*");
ADRGlobPattern pat = new ADRGlobPattern(kvpairs);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    System.out.println("ADR Name: " + name.toString());
}
```

Regex Pattern Searching

We can also take advantage of RAD's ERE (Extended Regular Expression) search capabilities. If we wanted to find only zones with name "test-0" or "test-1", then:

EXAMPLE 3-23 Using Regex Patterns

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRRegexPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

String keys[] = { "name" };
String values[] = { "test-0|test-1" };
ADRRegexPattern pat = new ADRRegexPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    System.out.println("ADR Name: " + name.toString());
}
```

The key and the value must be valid Extended Regular Expressions as determined by the instance of RAD to which you are connected. (i.e.: the expression is compiled and executed on the server.)

Interface Components

An API is defined by a module developer and contains a variety of components designed to accomplish a task. These components are:

- Enums
  - Values
- Structs
  - Fields
- Interfaces
  - Properties
  - Methods
  - Events

These components are all defined in an ADR Interface Description Language document. The `radadrgen` utility is used to process the document to generate language specific components which facilitate client/server interactions within RAD. More details in the role of ADR and `rad` can be found in Chapter 4, "Abstract Data Representation". Brief descriptions of each component follows.

Enumerations

Enumerations are primarily used to offer a restricted range of choices for a property, an interface method parameter, result, or error.

Using Enumeration Types

To access an enumerated type, simply import the generated class and interact with the enumeration.

EXAMPLE 3-24 Using Enumerations

```
import com.oracle.solaris.rad.zonemgr.ErrorCode;

System.out.println(ErrorCode.NONE);
System.out.println(ErrorCode.COMMAND_ERROR);
```

Structs

Structs are used to define new types and are composed from existing built-in types and other user defined types. In essence, they are simple forms of interfaces. They do not have methods or events and are not present in the RAD namespace.

Using Struct Types

The zonemgr module defines a Property struct which represents an individual Zone configuration property. The structure has the following members name, type, value, listValue, and complexValue. Like enumerations, structures can be interacted with directly once the binding is imported.

EXAMPLE 3-25 Using Structs

```
import com.oracle.solaris.rad.zonemgr.Property;

Property prop = new Property();
prop.setName("my name");
prop.setValue("a value");
System.out.println(prop.getName());
System.out.println(prop.getValue());
```

Interfaces/Objects

Interfaces (also known as objects) are the entities which populate the RAD namespace. They must have a "*name*". An interface is composed of Events, Properties and Methods.

Obtaining an Object Reference

See the "Rad Namespace" on page 35 section.

Working with Object References

Once we have an object reference we can use this to interact with RAD in a very straightforward fashion. All attributes and methods defined in the IDL are accessible directly as attributes and methods of the Java objects which are returned by `getObject`. Attributes are accessed using automatically generated getters/setters. For example, if the property is name, then you would use `getname`/ `setname(<value>)`. Here is an example which gets a reference to a zone and then boots the zone.

EXAMPLE 3-26 Invoking a Remote Method

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.Zone;

Connection con = Connection.connectUnix();
```

```
System.out.println("Connected: " + con.toString());

String keys[] = { "name" };
String values[] = { "test-0" };
ADRGlobPattern pat = new ADRGlobPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    Zone z = (Zone) con.getObject(name);
    z.boot(null);
}
```

In the above example we have connected to our RAD instance, created a search for a specific object, retrieved a reference to the object and invoked a remote method on it.

Accessing a Remote Property

Accessing a remote property is just as simple as using a remote method.

EXAMPLE 3-27 Accessing a Remote Property

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.ADRGlobPattern;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.*;

Connection con = Connection.connectUnix();
System.out.println("Connected: " + con.toString());

String keys[] = { "name" };
String values[] = { "test-0" };
ADRGlobPattern pat = new ADRGlobPattern(keys, values);
for (ADRName name: con.listObjects(new Zone(), pat)) {
    Zone z = (Zone) con.getObject(name);
    Resource filter = new Resource("global", null, null);
    List<Property> props = z.getResourceProperties(filter, null);
    System.out.println("Properties:");
    for (Property prop: props) {
        System.out.printf("\t%s = %s\n",prop.getName(), prop.getValue());
    }
}
```

In this example, we accessed the list of `Zone` global resource properties and printed out the name and value of every Property.

RAD Event Handling

In this next example we are going to look at events. The ZoneManager instance defines a "stateChange" event which clients can subscribe to for information about changes in the runtime state of a zone.

EXAMPLE 3-28 Subscribing and Handling Events

```
import com.oracle.solaris.rad.client.ADRName;
import com.oracle.solaris.rad.client.RadEvent;
import com.oracle.solaris.rad.client.RadEventHandler;
import com.oracle.solaris.rad.connect.Connection;
import com.oracle.solaris.rad.zonemgr.*;

ZoneManager zmgr = con.getObject(new ZoneManager());
    con.subscribe(zmgr, "statechange", new StateChangeHandler());
    Thread.currentThread().sleep(100000000);

class StateChangeHandler extends RadEventHandler {

    public void handleEvent(RadEvent event, Object payload) {
        StateChange obj = (StateChange) payload;
        System.out.printf("Event: %s", event.toString());
        System.out.printf("\tcode: %s\n", obj.getZone());
        System.out.printf("\told: %s\n", obj.getOldstate());
        System.out.printf("\tnew: %s\n", obj.getNewstate());
       }
}
```

To handle an event, implement the `RadEventInterface`. The `com.oracle.solaris.rad.client` package provides a default implementation (RadEventHandler) with limited functionality. This class may be extended to provide additional event handling logic as in example above.

In this simple example we subscribe to the single event and pass in a handler and a reference to our ZoneManager object. The handler will be invoked asynchronously by the framework with the various event details and the supplied user data.

RAD Error Handling

Finally, a quick look at error handling when manipulating remote references. Python provides a rich exception handling mechanism and RAD errors are propagated using this mechanism. There are a variety of errors which can be delivered by RAD, but the main one which requires handling is `RadObjectException`. The following snippet, shows how it can be used:

EXAMPLE 3-29 Handling RAD Errors

```
<imports..>

Connection con = Connection.connectUnix();
    for (ADRName name: con.listObjects(new Zone())) {
        Zone zone =  con.getObject(name);
        try {
            zone.boot(null);
        }catch (RadObjectException oe) {
            Result res = (Result) oe.getPayload();
            System.out.println(res.getCode());
            if (res.getStdout() != null)
                System.out.println(res.getStdout());
            if (res.getStderr() != null)
                System.out.println(res.getStderr());
        }
}
```

Note - With RadException exceptions you may get a payload. This is only present if your interface method or property has defined an "error" element, in which case the payload is the content of that error. If there is no "error" element for the interface method (or property), then there is no payload and it will have a value of null.

Python Client

The public interfaces are exported in three modules

- rad.auth – Useful functions/classes for performing authentication
- rad.client – The client implementation of the RAD protocol plus associated useful functionality
- rad.connect – Useful functions/classes for connecting to a RAD instance.

Note - A lot of these examples are based on the example zonemgr interface. Refer to the sample in, Appendix A, “`zonemgr` ADR Interface Description Language” for this module to assist in your understanding of the examples.

Alternatively, you can import the module and examine the module help

Accessing Help for a Binding Module

EXAMPLE 3-30 Accessing Help for a Binding Module

user@henry:/var/tmp# python

Python 2.6.8 (unknown, Feb 5 2013, 00:27:10) [C] on sunos5

Type "help", "copyright", "credits" or "license" for more information.

>>> import rad.bindings.com.oracle.solaris.rad.zonemgr_1 as zonemgr

>>> help(zonemgr)

Connecting to RAD

Communication with a RAD instance is provided through the `RADConnection` class. There are various mechanism to get different types of connections to RAD. Each mechanism returns a `RADConnection` instance which provides a standard interface to interact with RAD.

The preferred approach for managing a connection is to use the "with" keyword. The connection makes use of system resources and this style ensures that the resource is closed correctly when the object goes out of scope. If this style isn't used, then the system resources can be reclaimed explicitly with the `close` method.

Note - If you print the `RadConnection` object, it displays the state of the connection and will say closed if the connection is closed.

Connecting to a Local Instance

Implicit authentication is performed against your user id and most RAD tasks you request with this connection are performed with the privileges available to your user account.

EXAMPLE 3-31 Creating a Local Connection

```
>>> import rad.connect as radcon

>>> with radcon.connect_unix() as rc:
```

Connecting to a Remote Instance and Authenticating

When connecting to a remote instance, there is no implicit authentication, so the connection is not useful until authentication is performed. The `rad.auth` module provides a utility class (RadAuth) which may be used to perform a PAM login. If you supply a username and password, authentication proceeds in a non-interactive fashion. If either is absent, you will receive a console prompt for the missing information.

EXAMPLE 3-32 Remote Connection over TLS

```
>>> import rad.connect as radcon
>>> import rad.auth as rada
>>> rc=radcon.connect_tls("henry")
>>> # Illustrate examining RadConnection state.
>>> print rc
<open RadConnection >
>>> auth = rada.RadAuth(rc)
>>> auth.pam_login("garypen", "xxxpasswordxxx")
>>> <now authenticated and can use this connection>
>>> rc.close()
>>> print rc
<closed RadConnection >
>>>
```

Rad Namespace

All RAD objects, which are represented in the ADR IDL as `<interfaces>`, are found by searching the RAD namespace. The key point to note is that to access a RAD object, you need a proxy, which is used to search the RAD namespace. This capability is provided by an interface proxy class, which is defined in each interface's binding module.

The proxy provides the base name (and version details) for interface instances and is structured as follows:

```
<domain name>:type=<interface name>[,optional additional key value pairs]
```

The <domain name> and the <interface name> are automatically derived from the ADR IDL definition and are stored in the module binding.

Searching for Objects

The `RADConnection` class provides mechanisms for listing objects by name and for obtaining a remote object reference.

Obtaining a Reference to a Singleton

If a module developer creates a "singleton" to represent an interface, then this can be accessed very simply. For instance, the zonemgr module defines a singleton interface: ZoneInfo. It contains information about the zone which contains the RAD instance with which we are communicating.

EXAMPLE 3-33 Obtaining a Reference to a Singleton

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
...     zi = rc.get_object(zonemgr.ZoneInfo())
...     print zi.name
...
global
>>>
```

We have imported our binding and the `rad.connect` module, connected to the local RAD instance and obtained a remote object reference directly using a proxy instance. Once we have the remote reference we can access properties and method directly as we would with any Python object.

Listing Instances of an Interface

Most interfaces contain more than one instance of the interface. For instance, the zonemgr module defines a Zone interface and there is an instance for each zone on the system. The `RADConnection` class provides the `list_objects` method to list interface instances. For instance:

EXAMPLE 3-34 Listing Interface Instances

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone())
... print zonelist
...
[Name: com.oracle.solaris.rad.zonemgr:type=Zone,name=test-0,id=-1 Version: (1.0), Name:
com.oracle.solaris.rad.zonemgr:type=Zone,name=test-1,id=-1 Version: (1.0), Name:
com.oracle.solaris.rad.zonemgr:type=Zone,name=NOT-TEST,id=-1 Version: (1.0)]
>>>
```

Obtaining a Remote Object Reference from a Name

Names (ADRName is the class name) are returned by the RADConnection list_objects method. Once we have a "name" we can obtain a remote object reference easily:

EXAMPLE 3-35 Obtaining a remote object reference

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone())
... zone = rc.get_object(zonelist[0])
... print zone.name
...
```

```
test-0
>>>
```

Sophisticated Searching

Clearly, the last example is not a very realistic use case. Rarely are we going to want to just pick a (semi-random) zone from a list and interact with it. More often than not we'll be looking for a zone which has a particular name or id or maybe a set of zones where the names all match some kind of pattern. The key idea to bear in mind is that you can extend the basic definition of a name provided by a Proxy. For instance, if zones are uniquely identified by a key: "name", then we can find a zone with name "test-0" as follows:

EXAMPLE 3-36 Using Glob Patterns

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone(), radc.ADRGlobPattern({"name" : "test-0"}))
... print zonelist
...
[Name: com.oracle.solaris.rad.zonemgr:type=Zone,name=test-0,id=-1 Version: (1.0)]
>>>
```

The `ADRGlobPattern` class (imported from the `rad.client` module) to refine our search. We are using the same RADConnection `list_objects` method, but we are refining the search by extending the name definition. `ADRGlobPattern` takes a key:value dictionary and extends the name used for the search.

Glob Pattern Searching

We have already seen how we can use glob pattern searching to find a zone with a specific name. We can also use a glob pattern to find zones with wildcard pattern matching. Keys or Values in the pattern may contain "*" which is interpreted as expected. For instance if we wanted to find all zones with a name which begins with "test":

EXAMPLE 3-37 Using Glob Patterns with Wildcards

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone(), radc.ADRGlobPattern({"name" : "test*"}))
...  print zonelist
...
```

```
[Name: com.oracle.solaris.rad.zonemgr:type=Zone,name=test-0,id=-1 Version: (1.0), Name:
com.oracle.solaris.rad.zonemgr:type=Zone,name=test-1,id=-1 Version: (1.0)]
>>>
```

Regex Pattern Searching

We can also take advantage of RAD's ERE (Extended Regular Expression) search capabilities. If we wanted to find only zones with name "test-0" or "test-1", then:

EXAMPLE 3-38 Using Regex Patterns

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
... zonelist = rc.list_objects(zonemgr.Zone(), radc.ADRRegexPattern({"name" : "test-0|test-1"}))
...  print zonelist...
[Name: com.oracle.solaris.rad.zonemgr:type=Zone,name=test-0,id=-1 Version: (1.0), Name:
com.oracle.solaris.rad.zonemgr:type=Zone,name=test-1,id=-1 Version: (1.0)]
>>>
```

The key and the value must be valid Extended Regular Expressions as determined by the instance of RAD to which you are connected. (i.e.: the expression is compiled and executed in the server.)

Interface Components

An API is defined by a module developer and contains a variety of components designed to accomplish a task. These components are:

- Enums
  - Values
- Structs
  - Fields
- Interfaces
  - Properties
  - Methods
  - Events

These components are all defined in ADR Interface Description Language document. The `radadrgen` utility is used to process the document to generate language specific components which facilitate client/server interactions within RAD. More details in the role of ADR and `rad` can be found in Chapter 4, “Abstract Data Representation”. Brief descriptions of each component follows.

Enumerations

Enumerations are primarily used to offer a restricted range of choices for a property, an interface method parameter, result, or error.

Using Enumeration Types

To access an enumerated type, simply import the binding and interact with the enumeration.

EXAMPLE 3-39 Using Enumerations

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> print zonemgr.ErrorCode.NONE
NONE
>>> print zonemgr.ErrorCode.COMMAND_ERROR
COMMAND_ERROR
>>>
```

Structs

Structs are used to define new types and are composed from existing built-in types and other user defined types. In essence, they are simple forms of interfaces: no methods or events and they are not present in the RAD namespace.

Using Struct Types

The zonemgr module defines a Property struct which represents an individual Zone configuration property. The structure has the following members name, type, value, listValue, and complexValue. Like enumerations, structures can be interacted with directly once the binding is imported.

EXAMPLE 3-40 Using Structs

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> prop = zonemgr.Property("autoboot", "false")
>>> print prop
Property(name = 'autoboot', value = 'false', type = None, listvalue = None, complexvalue = None)
>>> prop.name = "my name"
>>> prop.value = "a value"
>>> print prop.name
my name
>>> print prop.value
a value
>>>
```

Interfaces/Objects

Interfaces (also known as objects) are the entities which populate the RAD namespace. They must have a "name". An interface is composed of Events, Properties and Methods.

Obtaining an Object Reference

See the “Rad Namespace” on page 46 section.

Working with Object References

Once we have an object reference we can use this to interact with RAD in a very straightforward fashion. All attributes and methods defined in the IDL are accessible directly as attributes of the python objects which are returned by `get_object`.

Here is an example which gets a reference to a zone and then boots the zone.

EXAMPLE 3-41 Working with Object References

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
...     patt = radc.ADRGlobPattern({"name" : "test-0"})
...     zone = rc.get_object(zonemgr.Zone(), patt)
...     print zone.name
...     zone.boot(None)
>>>
```

In the above example we have connected to our RAD instance, created a search for a specific object, retrieved a reference to the object and accessed a remote property on it. This does not include error handling yet.

Accessing a Remote Property

Here is another example for accessing a remote property.

EXAMPLE 3-42 Accessing a Remote Property

```
>>> import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
>>> import rad.client as radc
>>> import rad.connect as radcon
>>> with radcon.connect_unix() as rc:
...     name = rc.list_objects(zonemgr.Zone(), radc.ADRGlobPattern({"name" : "test-0"}))
...     zone = rc.get_object(name[0])
...     for prop in zone.getResourceProperties(zonemgr.Resource("global")):
...         if prop.name == "brand":
...             print "Zone: %s, brand: %s" % (zone.name, prop.value)
...             break
...
Zone: test-0, brand: solaris
>>>
```

In this example, we accessed the list of `Zone` global resource properties and search the list of properties for a property with the name property of "brand". When we find it we print the value of the "brand" property and then terminate the loop.

RAD Event Handling

In this next example we are going to look at events. The ZoneManager instance defines a "stateChange" event which clients can subscribe to for information about changes in the runtime state of a zone.

EXAMPLE 3-43 Subscribing and Handling Events

```
import rad.connect as radcon
import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
import signal

def handler(event, payload, user):
   print "event: %s" % str(event)
   print "payload: %s" % str(payload)
   print "zone: %s" % str(payload.zone)
   print "old: %s" % str(payload.oldstate)
   print "new: %s" % str(payload.newstate)

with radcon.connect_unix() as rc:
    zm = rc.get_object(zonemgr.ZoneManager())
    rc.subscribe(zm, "stateChange", handler, zm)
    signal.pause()
```

This is a simple example. We subscribe to the single event and pass in a handler and a reference to our ZoneManager object. The handler will be invoked asynchronously by the framework with the various event details and the supplied user data. The user data is an optional argument at subscription and if not provided, the handler will receive None as the user parameter.

RAD Error Handling

Finally, a quick look at error handling when manipulating remote references. Python provides a rich exception handling mechanism and RAD errors are propagated using this mechanism. There are a variety of errors which can be delivered by RAD, but the main one which requires handling is `rad.client.ObjectError`. The following snippet, shows how it can be used:

EXAMPLE 3-44 Handling RAD Errors

```
import rad.client as radc
import rad.conect as radcon
import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr
import logging
import sys

logging.basicConfig(filename='/tmp/example.log', level=logging.DEBUG)
with radcon.connect_unix() as rc:
    patt = radc.ADRGlobPattern({"name" : "test-0"})
    test0 = rc.get_object(zonemgr.Zone(), patt)
    print test0.name
    try:
        test0.boot(None)
    except radc.ObjectError as ex:
        error = ex.get_payload()
        if not error:
            sys.exit(1)
        if error.stdout is not None:
            logging.info(error.stdout)
        if error.stderr is not None:
            logging.info(error.stderr)
        sys.exit(1)
```

Note - With ObjectError exceptions you may get a payload. This is only present if your interface method or property has defined an "error" element, in which case the payload is the content of that error. If there is no "error" element for the interface method (or property), then there is no payload and it will have a value of None.

♦ ♦ ♦ CHAPTER 4

Abstract Data Representation

The data model used by `rad` is known as the Abstract Data Representation (ADR). ADR defines a formal IDL for describing types and interfaces supplies a toolchain for operating on that IDLand provides libraries used by `rad`, its extension modules, and its clients.

ADR Interface Description Language

The APIs used by `rad` are defined using an XML-based IDL. The normative schema for this language can be found in `/usr/share/lib/xml/rng/adr.rng.1`. The namespace name is `http://xmlns.oracle.com/radadr`.

Overview

The top-level element in an ADR definition document is an `api`. The `api` element has one mandatory attribute, `name`, which is used to name the output files. The element contains one or more derived type or interface definitions. Because there is no requirement that an interface use derived types, there is no requirement that any derived types be specified in an API document. To enable consumers to use the data typing defined by ADR for non-interface purposes, there is no requirement that an interface is defined either. However, note that either a derived type or an interface must be defined.

Two derived types are available for definition and use by interfaces: a structured type that can be defined with a `struct` element, and an enumeration type that can be defined with an `enum` element. Interfaces are defined using `interface` elements. The derived types defined in an API document are available for use by all interfaces defined in that document.

This is what an API looks like:

EXAMPLE 4-1 Skeleton API document

```
<api xmlns="http://xmlns.oracle.com/radadr" name="com.oracle.solaris.rad.example" register="true">
        <version/>
        <struct>...</struct>
        <struct>...</struct>
        <enum>...</enum>
        <interface>...</interface>
        <interface>...</interface>
</api>
```

The xmlns is required to indicate the type of the XML document. The name attribute is identifying the name of the API, the namespace within which all subsidiary interfaces are to be found. There are additional attributes to assist in the generation of server module code.

The register attribute is a `boolean` which is optional and true by default. If true, then radadrgen will automatically generates a `_rad_reg` function when generating server implementation code. If false, the function is not generated and the module author will need to provide a `_rad_reg` function. This option is primarily provided for the creation of special types of modules, such as protocol or transport modules, in general it does not need to be specified, since the default generated function is enough for most purposes.

Version

A version element is required for all APIs.

The initial version of an API should always be defined as follows:

```
<version major="1" minor="0"/>
```

This indicates that the module is starting at version 1.0.

Enumeration Definitions

The `enum` element has a single mandatory attribute, `name`. The `name` is used when referring to the enumeration from other derived type or interface definitions. An `enum` contains one or more `value` elements, one for each user-defined enumerated value. A `value` element has a mandatory `name` attribute that gives the enumerated value a symbolic name. The symbolic name isn't used elsewhere in the API definition, only in the server and various client environments. How the symbolic name is exposed in those environments is environment-dependent. An environment offering an explicit interface to `rad` should provide an interface that accepts the exact string values defined by the `value` elements' `name` attributes.

Some language environments support associating scalar values with enumerated type values, for example C. To provide richer support for these environments, ADR supports this concept as well. By default, an enumerated value has an associated scalar value 1 greater than the preceding enumerated value's associated scalar value. The first enumerated value is assigned a scalar value of 0. Any enumerated value element may override this policy by defining a `value` with the desired value. A `value` attribute must not specify a scalar value already assigned, implicitly or explicitly, to an earlier value in the enumeration and `value` elements contain no other elements.

EXAMPLE 4-2 Enumeration Definition

```
<enum name="Colors">
<value name="RED" /> <!-- scalar value: 0 -->
<value name="ORANGE" /> <!-- scalar value: 1 -->
<value name="YELLOW" /> <!-- scalar value: 2 -->
<value name="GREEN" /> <!-- scalar value: 3 -->
<value name="BLUE" /> <!-- scalar value: 4 -->
<value name="VIOLET" value="6" /> <!-- indigo was EOLed -->
</enum>
```

Structure Definitions

Like the `enum` element, the `struct` element has a single mandatory attribute, `name`. The name is used when referring to the structure from other derived type or interface definitions. A `struct` contains one or more `field` elements, one for each field of the structure. A `field` element has a mandatory `name` attribute that gives the field a symbolic name. The symbolic name isn't used elsewhere in the API definition, only in the server and various client environments. In addition to a name, each field must specify a type.

You can define the type of a field in multiple ways. If a field is a plain base type, that type is defined with a `type` attribute. If a field is a derived type defined elsewhere in the API document, that type is defined with a `typeref` attribute. If a field is an array of some type (base or derived), that type is defined with a nested `list` element. The type of the array is defined in the same fashion as the type of the field: either with a `type` attribute, a `typeref` attribute, or another nested `list` element.

A field's value may be declared nullable by setting the `field` element's `nullable` attribute to true.

Note - Structure fields, methods return values, method arguments, attributes, error return values, and events all have types, and in the IDL, use identical mechanisms for defining those types.

EXAMPLE 4-3 `struct` Definition

```
<struct name="Name">
        <field name="familyName" type="string" />
        <field name="givenNames">
                <list type="string" />
        </field>
</struct>
```

```
<struct name="Person">
        <field name="name" typeref="Name" />
        <field name="title" type="string" nullable="true" />
        <field name="shoeSize" type="int" />
</struct>
```

Interface Definitions

An interface definition has a name, and one or more attributes, methods, or events. An interface's name is defined with the `interface` element's mandatory `name` attribute. This name is used when referring to the inherited interface from other interface definitions, as well as in the server and various client environments. The other characteristics of an interface are defined using child elements of the `interface` element.

Methods

Each method in an interface is defined by a `method` element. The name of a method is defined by this element's mandatory `name` attribute. The other properties of a method are defined by child elements of the `method`.

If a method has a return value, it is defined using a single `result` element. The type of the return value is specified in the same way the type is specified for a structure field. If no `result` element is present, the method has no return value.

If a method can fail for an API-specific reason, it is defined using a single `error` element. The type of an error is specified the same way the type is specified for a structure field. Unlike a structure field, an error need not specify a type. Such a situation is indicated by an `error` element with no attributes or child elements. If no `error` element is present, the method will only fail if there is a connectivity problem between the client and the server.

A method's arguments are defined, in order, with zero or more `argument` elements. Each `argument` element has a mandatory `name` attribute. The type of an argument is specified in the same way the type is specified for a structure field.

EXAMPLE 4-4 Method Definition

```
<struct name="Meal">...</struct>
<struct name="Ingredient">...</struct>

<method name="cook">
        <result typeref="Meal" />
        <error />
        <argument type="string" name="name" nullable="true" />
        <argument name="ingredients">
```

```
                <list typeref="Ingredient" />
        </argument>
</method>
```

Attributes

Each attribute in an interface is defined by a `property` element. The name of an attribute is defined by this element's mandatory `name` attribute. The types of access permitted are defined by the mandatory `access` attribute, which takes a value of `ro`, `wo`, or `rw`, corresponding to read-only access, write-only access, or read-write access, respectively.

The type of an attribute is specified in the same way the type is specified for a structure field.

If access to an attribute can fail for an API-specific reason, it is defined using one or more `error` elements. An `error` element in a `property` may specify a `for` attribute, which takes a value of `ro`, `wo`, or `rw`, corresponding to the types of access the error return definition applies to. An `error` element with no `for` attribute is equivalent to one with a `for` attribute set to the access level defined on the `property`. Two error elements may not specify overlapping access types. For example, on a read-write property it is invalid for one `error` to have no `for` attribute (implying `rw`) and one to have a `for` attribute of `wo` they both specify an error for writing.

The type of an error is specified the same way the type is specified for a method. It is identical to defining the type of a structure, with the exception that a type need not be defined.

EXAMPLE 4-5 Attribute Definition

```
<struct name="PrivilegeError">...</struct>

<property name="guestList" access="rw">
        <list type="string" />
        <error for="wo" typeref="PrivilegeError" />
        <!-- Reads cannot fail -->
</property>
```

Events

Each event in an interface is defined by a `event` element. The name of an event is defined by this element's mandatory `name` attribute. The type of an event is specified in the same way the type is specified for a structure field.

EXAMPLE 4-6 Event Definition

```
<struct name="TremorInfo">...</struct>

<event name="earthquakes" typeref="TremorInfo" />
```

Example

EXAMPLE 4-7 Complete API Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<api xmlns="http://xmlns.oracle.com/radadr" name="com.oracle.solaris.rad.example">

         <version major="1" minor="0"/>

         <struct name="StringInfo">
                <field type="integer" name="length" />
                <field name="substrings">
                        <list type="string" />
                </field>
        </struct>

        <struct name="SqrtError">
                <field type="float" name="real" />
                <field type="float" name="imaginary" />
        </struct>

        <enum name="Mood">
                <value name="IRREVERENT" />
                <value name="MAUDLIN" />
        </enum>

        <struct name="MoodStatus">
                <field typeref="Mood" name="mood" />
                <field type="boolean" name="changed" />
        </struct>

        <interface name="GrabBag" stability="private">

                <method name="sqrt">
                        <result type="integer" />
                        <error typeref="SqrtError" />
                        <argument type="integer" name="x" />
                </method>

                <method name="parseString">
                        <result typeref="StringInfo" nullable="true" />
                        <argument type="string" name="str" nullable="true" />
                </method>

                <property typeref="Mood" name="mood" access="rw">
                        <error for="wo" />
                </property>

                <event typeref="MoodStatus" name="moodswings" />
        </interface>
</api>
```

radadrgen

radadrgen is the ADR IDL processing tool. It is used to generate API-specific language bindings for the rad server and various client environments. See the radadrgen(1) man page for details on all its options.

♦ ♦ ♦ CHAPTER 5

libadr

The library libadr provides structure definitions and subroutines essential to C programs using ADR. Code generated by radadrgen requires this library, and rad itself is based on it. libadr contains three major areas of functionality: data management, API management, and object name operations.

libadr is delivered in the system/management/rad package. C programs can link with it by specifying -ladr on the compile or link line.

Data Management

Consumers of the ADR data management routines should include the rad/adr.h header file:

```
#include <rad/adr.h>
```

This file contains definitions for the two fundamental data management types, adr_type_t and adr_data_t, as well as prototypes for data allocation, access, and validation routines.

adr_type_t Type

Each data type is represented by an adr_type_t type, whether it is just a base type or a complex type of nested structures and arrays. The adr_type_t contains all the information necessary to understand the structure of the type. libadr provides statically-allocated singletons of adr_type_t type for the base types. These singleton types are more than a convenience: they must be used when referencing the base types.

The base types and their corresponding array types are listed in the following table.

TABLE 5-1 Base and Array Types

ADR type	C adr_type_t	C array adr_type_t
string	adr_t_string	adr_t_array_string

ADR type	C adr_type_t	C array adr_type_t
integer	adr_t_integer	adr_t_array_integer
uinteger	adr_t_uinteger	adr_t_array_uinteger
long	adr_t_long	adr_t_array_long
ulong	adr_t_ulong	adr_t_array_ulong
time	adr_t_time	adr_t_array_time
name	adr_t_name	adr_t_array_name
boolean	adr_t_boolean	adr_t_array_boolean
opaque	adr_t_opaque	adr_t_array_opaque
secret	adr_t_secret	adr_t_array_secret
float	adr_t_float	adr_t_array_float
double	adr_t_double	adr_t_array_double
reference	adr_t_double	adr_t_array_reference

The `adr_type_t` for a derived type should also be unique, but obviously they cannot be defined by libadr. Although technically `adr_type_t` could be dynamically allocated, at the moment, the only supported way of defining an `adr_type_t` is to generate a definition using the ADR IDL and `radadrgen`.

`adr_data_t` Type

The most frequently used type defined by `rad/adr.h` is `adr_data_t`. An `adr_data_t` object represents a unit of typed data. It could be of a base type, such as an integer (“1”) or string (“banana”), or of a derived type like a structure or an array. Each `adr_data_t` maintains a pointer to its `adr_type_t`.

A few common traits simplify access to `adr_data_t` objects. The first is that, except for the structure and array derived types (not enumerations), all `adr_data_t` values are immutable. They are assigned a value when they are created, and may not be changed thereafter.

Another is that all `adr_data_t` values are reference counted. Sometimes data structures need to be used by multiple consumers simultaneously, or simply retained for subsequent use. Reference counting is a cheap way to cut down on the cost of copying large data structures and the complexity of handling allocation failures. Though the reference counting is thread-safe, there is no other locking, which is not a problem for an immutable `adr_data_t`. Though the value of a non-immutable `adr_data_t` may be modified post-creation, the convention used throughout `rad` and its associated libraries is that once visibility of an `adr_data_t` has spread past its creator, it may no longer be modified. This eliminates the need for additional synchronization.

```
adr_data_t *adr_data_ref(adr_data_t *data);
void adr_data_free(adr_data_t *data);
```

The reference count on the adr_data_t data is incremented with adr_data_ref. For convenience, adr_data_ref returns data. Symmetrically, the reference count on the adr_data_t data is decremented with adr_data_free. As the name implies, this may result in data being freed; after calling adr_data_free the caller must not access data in any way. Neither adr_data_ref nor adr_data_free can fail.

A third trait is that interfaces that accept adr_data_t values take ownership of the caller's reference on the adr_data_t. If the caller needs to refer to the adr_data_t after passing a pointer to it to a libadr interface, it must first secure an additional reference with adr_data_ref. Interfaces that return adr_data_t that are referenced by other adr_data_t do not increase the reference count on the returned adr_data_t. The returned value is guaranteed to persist only as long as the caller retains a reference on the referring adr_data_t, or if the caller uses adr_data_ref to acquire its own reference on the returned adr_data_t. The net result is that in the common case where an adr_data_t does not have multiple simultaneous consumers, libadr consumers need not perform any explicit reference counting at all. They can naively allocate and free adr_data_t values as if they were any other data structure. Therefore the adr_data_t implementation can optimize for the case where the reference count is 1.

Lastly, many adr_data_t management routines rely on dynamic memory allocation, which means that proper error handling is essential. To increase the clarity and maintainability of adr_data_t consumers, and reduce the likelihood of mishandling errors, libadr interfaces explicitly accept NULL adr_data_t inputs and fail in sympathy. This means that a libadr consumer can perform a large number of operations on the instances of adr_data_t, checking only the final result for failure. Additionally, if a libadr routine is going to fail for any reason, references to a non-NULL adr_data_t passed to the routine is released. In other words, no special clean-up is needed when a libadr routine fails.

Allocating adr_data_t Values

The first phase in the lifecycle of an adr_data_t is allocation. For each ADR type, there is at least one allocation routine. The arguments to an allocation routine depend on the type. In the case of mandatorily immutable types, allocation implies initialization, and their allocation routines take as arguments the value the adr_data_t is to have. Structures and arrays each have a single generic allocation routine that takes an adr_type_t* specifying the type of the structure or array. An adr_data_t is assigned values using a separate set of routines.

All allocation routines return a non-NULL adr_data_t * on success, or NULL on failure.

Note - The allocation and initialization routines for immutable types may elect to return a reference to a shared `adr_data_t` for a commonly used value, for example, boolean true or false. This substitution should be undetectable by `adr_data_t` consumers who correctly manage `adr_data_t` reference counts and respect the immutability of these types.

Allocating Strings

```
adr_data_t *adr_data_new_string(const char *s, lifetype_t lifetime);
```

Allocates a new string `adr_data_t`, initializing it to the NULL-terminated string pointed to by s. If s is `NULL`, `adr_data_new_string` will fail.

The value of the `lifetime` determines how the string `s` is to be used.

`LT_COPY` — `adr_data_new_string` must allocate and make a copy of the string pointed to by s. This copy will be freed when the `adr_data_t` is freed.

`LT_CONST` — The string pointed to by s is a constant that will never be changed or deallocated. Therefore, `adr_data_new_string` need not copy the string; it can instead refer directly to s indefinitely. This is the recommended lifetime value when passing a string literal to `adr_data_new_string`.

`LT_FREE` — The string pointed to by s was dynamically allocated using `malloc` and is no longer needed by the caller. `adr_data_new_string` will ensure that the string is eventually freed. It may choose to use the string directly instead of making a copy of it. Obviously, this lifetime value should never be used with string literals.

If lifetime is `LT_FREE` and `adr_data_new_string` fails for any reason, s will automatically be freed.

```
adr_data_t *adr_data_new_fstring (const char *format, ...);
```

Allocates a new string `adr_data_t`, initializing it to the string generated by calling `sprintf` on format and any additional arguments provided.

```
adr_data_t *adr_data_new_nstring (const char *s, size_t count);
```

Allocates a new string `adr_data_t`, initializing it to the first count bytes of `s`.

Allocating `boolean`

```
adr_data_t *adr_data_new_boolean (boolean_t b);
```

Allocates a new boolean `adr_data_t`, initializing it to the boolean value specified by `b`.

Allocating Numeric Types

```
adr_data_t *adr_data_new_integer (int i);
```

```
adr_data_t *adr_data_new_long (long long l);
```

```
adr_data_t *adr_data_new_uinteger (unsigned int ui);
```

```
adr_data_t *adr_data_new_ulong (unsigned long long ul);
```

```
adr_data_t *adr_data_new_float (float f);
```

```
adr_data_t *adr_data_new_double (double d);
```

Allocates a new integer, `long`, `uinteger`, `ulong`, `float`, or `double adr_data_t`, respectively, initializing it to the value of the single argument provided.

Allocating Times

```
adr_data_t *adr_data_new_time (long long sec, int nano);
```

```
adr_data_t *adr_data_new_time_ts (timespec &t);
```

```
adr_data_t *adr_data_new_time_now (void );
```

Allocates a new `time adr_data_t`, initializing it to the argument, if any, provided.

Allocating Opaques

```
adr_data_t *adr_data_new_opaque (void *buffer, size_t length, adr_lifetime_t lifetime);
```

Allocates a new opaque `adr_data_t`, initializing it to the length bytes found at buffer. How `adr_data_new_opaque` uses buffer depends on lifetime, which takes on the same meanings as it does when used with `adr_data_new_string`.

Allocating Secrets

```
adr_data_t *data_new_secret (const char *p);
```

Allocates a new secret `adr_data_t`, initializing it to the contents of the `NULL`-terminated 8-bit character array pointed to by `p`. The secret type is used to hold sensitive data such as passwords. Client/server implementations may take additional steps to protect the content of the character array data, for example, zeroing buffers after use.

Allocating Names

```
adr_data_t *adr_data_new_name (adr_name_t *name);
```

Allocates a new name adr_data_t, initializing it to the value of name. adr_name_t types are reference counted; the reference on name held by the caller is transferred to the resulting adr_data_t by the call to adr_data_new_name. A caller that needs to continue using name should secure an additional reference to it before calling adr_data_new_name. If adr_data_new_name fails for any reason, the caller's reference to name will be released.

Allocating Enumerations

```
adr_data_t *adr_data_new_enum (adr_type_t *type, int value);
```

```
adr_data_t *adr_data_new_enum_byname (adr_type_t *type, const char * name);
```

The two ways to allocate an enumeration adr_data_t both require that the adr_type_t of the enumeration be specified. The first form, adr_data_new_enum, takes a scalar value as an argument and initializes the enumeration adr_data_t to the enumerated value that was assigned (implicitly or explicitly) that scalar value. The second form, adr_data_new_enum_byname, takes a pointer to a string as an argument and initializes the enumeration adr_data_t to the enumerated value that has that name. If value does not correspond to an assigned scalar value or name does not correspond to an enumerated value name, the respective allocation routine fails.

The nature of an enumeration is that all possible values are known. Enumerated types generated by radadrgen have singleton adr_data_t values that will be returned by adr_data_new_enum and adr_data_new_enum_byname. For efficiency and to reduce the error handling that needs to be performed at runtime, these values have defined symbols that may be referenced directly.

The value of type must be an enumeration data-type.

Allocating Structures

```
adr_data_t *adr_data_new_struct (adr_type_t *type);
```

Allocates an uninitialized structure adr_data_t of type type. Any post-allocation initialization that occurs must be consistent with type.

The value of type must be a structured type.

Allocating Arrays

```
adr_data_t *adr_data_new_array (adr_type_t *type, int size);
```

Allocates an empty array adr_data_t of type type. Arrays will automatically adjust their size to fit the amount of data placed in them. The size argument can be used to initialize the size of the array if it is known beforehand.

The value of type must be an array type.

Accessing Simple `adr_data_t` Values

`rad/adr.h` defines macros that behave like the following prototypes:

```
const char *adr_data_to_string(adr_data_t *data);

int adr_data_to_integer(adr_data_t *data);

unsigned int adr_data_to_uinteger(adr_data_t *data);

long long adr_data_to_longint(adr_data_t *data);

unsigned long long adr_data_to_ulongint(adr_data_t *data);

boolean_t adr_data_to_boolean(adr_data_t *data);

adr_name_t *adr_data_to_name(adr_data_t *data);

const char *adr_data_to_secret(adr_data_t *data);

float adr_data_to_float(adr_data_t *data);

double adr_data_to_double(adr_data_t *data);

const char * adr_data_to_opaque(adr_data_t *data);

long long adr_data_to_time_secs(adr_data_t *data);

int adr_data_to_time_nsecs(adr_data_t *data);
```

In all cases, pointer return values will point to data that is guaranteed to exist only as long as the caller retains their reference to the data parameter.

Additionally, the following functions are provided for interpreting enumeration values:

```
const char *adr_enum_tostring(adr_data_t *data);
int adr_enum_tovalue(adr_data_t *data);
```

`adr_enum_tostring` maps data to the value's string name. `adr_enum_tovalue` maps data to its scalar value.

The behavior is undefined if a macro or function is called on an `adr_data_t` of the wrong type.

Manipulating Derived Type `adr_data_t`

Structure and array derived types are assigned no value when they are allocated. As a best practice, you should assign some value to them before use; in the case of structured types with non-nullable fields, it is required. In either case, once a reference to a derived type is shared, it may no longer be modified.

Manipulating Array `adr_data_t` Values

`rad/adr.h` defines array-access macros that behave like the following prototypes:

```
int adr_array_size(adr_data_t *array);
adr_data_t *adr_array_get(adr_data_t *array, int index);
```

`adr_array_size` returns the number of elements in array. `adr_array_get` returns the index element of array. The `adr_data_t` returned by `adr_array_get` is valid as long as the caller retains its reference to array; if it is needed longer, the caller should take a hold on the `adr_data_t` (see "`adr_data_t` Type" on page 64). If the index element of array has not been set, the behavior of `adr_array_get` is undefined.

The following functions modify arrays:

```
int adr_array_add(adr_data_t *array, adr_data_t * value);
```

`adr_array_add` adds value to the end of array. As described in "`adr_data_t` Type" on page 64, the caller's reference to value is transferred to the array. `adr_array_add` might need to allocate memory and can therefore fail. When `adr_array_add` succeeds, it returns 0. When `adr_array_add` fails, it will return 1 and array will be marked invalid. For more information, see "Validating `adr_data_t` Values" on page 71.

```
void adr_array_remove(adr_data_t *array, int index);
```

`adr_array_remove` removes the index element from array. The array's reference count on the element at index is released, possibly resulting in its deallocation. All elements following index in array are shifted to the next lower position in the array, for example, element index+1 is moved to index. The behavior of `adr_array_remove` is undefined if index is greater than or equal to the size of array as returned by `adr_array_size`.

```
int adr_array_vset(adr_data_t *array, int index, adr_data_t * value);
```

`adr_array_vset` sets the index element of array to value. If an element was previously at index, the reference on that element held by the array is released. `adr_array_vset` may need to allocate memory and can therefore fail. When `adr_array_vset` succeeds, it returns 0. When `adr_array_vset` fails, it will return 1 and array will be marked invalid. For more information, see "Validating `adr_data_t` Values" on page 71.

Manipulating the Structure of an `adr_data_t` Type

The primary interface for accessing an `adr_data_t` structure is `adr_struct_get`:

```
adr_data_t *adr_struct_get(adr_data_t *struct, const char *field);
```

adr_struct_get returns the value of the field named field. If the field is nullable and has no value or if the field hasn't been given a value (that is the structure was incompletely initialized), adr_struct_get returns NULL. The adr_data_t returned by adr_struct_get is valid as long as the caller retains its reference to struct. If it is needed longer the caller should take a hold on the adr_data_t. If struct does not have a field named field, the behavior of adr_struct_get is undefined.

The primary interface for writing to an adr_data_t structure is adr_struct_set:

```
void adr_struct_set(adr_data_t *struct, const char *field, adr_data_t *value);
```

adr_struct_set writes value to the field named field. If field previously had a value, the reference on that value held by the structure is released. If struct does not have a field named field, or if the type of value does not match that of the specified field the behavior of adr_struct_set is undefined.

Validating adr_data_t Values

libadr provides a rich environment for examining and manipulating typed data. However, unlike C's native typing system, the compiler is unaware of libadr type relationships and is therefore unable to perform static type-checking at compile time. All type checking must be performed at runtime.

The most useful of the type-checking tools provided by libadr is adr_data_verify:

```
boolean_t adr_data_verify(adr_data_t *data, adr_type_t *type, boolean_t recursive);
```

adr_data_verify takes an adr_data_t to type-check and an adr_type_t to type-check against. It can be instructed to check only the adr_data_t data or data and the transitive closure of every adr_data_t it references. adr_data_verify returns B_TRUE if data matches type, and B_FALSE if not. If type is NULL, data is tested against the type it claims to be. Although this method is not a good idea for input validation, it can be useful for error handling.

In order for data to be verified as type type, the following must be true:

- Data must not be NULL.
- Data must claim to be of type type.
- If type is an enumeration, data must be a value in that enumeration.
- If data is an array, it must be not have been marked invalid by a failed adr_array_add or adr_array_vset operation.
- If data is an array, it must have no NULL elements.

- If data is an array and recursive is true, each element of the array must satisfy these criteria given the array's element type.
- If data is a structure, every non-nullable field must have a value, that is, be non-NULL.
- If data is a structure and recursive is true, every non-NULL field value must satisfy these criteria considering the field's type.

The adr_data_verify is useful when validating input from an untrusted source. A second, less obvious application of adr_data_verify is as a powerful error-handling tool. Suppose you are writing a function that needs to return a complex data value. A traditional way of implementing it would be to check each call for failure individually, as shown in the following example.

EXAMPLE 5-1 Traditional Error Handling

```
adr_data_t *tmp, *name, *result;
if ((name = adr_data_new_struct(name_type)) == NULL) {
/* handle failure */
}
if ((tmp = adr_data_new_string("Jack")) == NULL) {
/* handle failure */
}
adr_struct_set(name, "first", tmp);
if ((tmp = adr_data_new_string("O'Neill")) == NULL) {
/* handle failure */
}
adr_struct_set(name, "last", tmp);
if ((record = adr_data_new_struct(record_type)) == NULL) {
/* handle failure */
}
adr_struct_set(record, "name", name);
/* ...and so on */
```

This approach is difficult to implement and difficult to maintain. It is more likely to have a flaw in it than the allocations it is testing are to fail. Instead, using adr_data_verify and the error handling behaviors described in "adr_data_t Type" on page 64, the entire non-truncated function can be reduced to the method shown in the following example.

EXAMPLE 5-2 Error Handling With adr_data_verify

```
adr_data_t *name = adr_data_new_struct(name_type);
adr_struct_set(name, "first", adr_data_new_string("Jack"));
adr_struct_set(name, "last", adr_data_new_string("O'Neill"));
adr_data_t *record = adr_data_new_struct(record_type);
adr_struct_set(record, "name", name);
adr_struct_set(record, "rank", adr_data_new_enum_byname("COLONEL"));
adr_struct_set(record, "l_count", adr_data_new_integer(2));

if (!adr_data_verify(record, NULL, B_TRUE)) { /* Recursive type check */
   adr_data_free(record);
   return (NULL); /* NULL means something failed */
}
```

```
return (record); /* Non-NULL means success */
```

An important limitation to this technique is the possibility for structure fields to be nullable, and the `NULL` indicating that the field has no value is indistinguishable from the `NULL` that indicates that the allocation of that field's value failed. In such cases, explicitly testing each nullable value's allocation is necessary. Even with such explicit checks, however, the net savings in complexity can be substantial.

ADR Object Name Operations

`libadr` supports ADR object names by providing an `adr_name_t` type and a suite of routines for creating and inspecting them. Consumers needing to operate on object names should include the `rad/adr_name.h` header file:

```
#include <rad/adr_name.h>
```

This file contains definitions for all the ADR-name related functionality provided by `libadr`.

`adr_name_t` Type

The `adr_name_t` type represents an object name. The internal structure of an `adr_name_t` is private. All operations on an `adr_name_t` are performed using accessor functions provided by `libadr`. Like `adr_data_t` values, `adr_name_t` values are immutable and reference counted. The following functions are provided for handling `adr_name_t` reference counts:

```
adr_name_t *adr_name_hold(adr_data_t *name);
void adr_name_rele(adr_name_t *name);
```

The reference count on the `adr_name_t` name is incremented with `adr_name_hold`. For convenience, `adr_name_hold` returns name. Symmetrically, the reference count on the `adr_name_t` name is decremented with `adr_name_rele`. When then last reference on an `adr_name_t` is released, the name is freed; after calling `adr_name_rele` the caller must not access name in any way. Neither `adr_name_hold` nor `adr_name_rele` can fail.

Creating `adr_name_t` Type

ADR names are composed of a domain and a set of key/value pairs. Two functions are provided that take exactly those arguments and return an `adr_name_t`:

```
adr_name_t *adr_name_create(const char *domain, int count,
  const char * const *keys, const char * const *values);
```

```
adr_name_t *adr_name_vcreate(const char *domain, int count, ...);
```

Both forms take a domain argument, which should be a reverse-dotted domain name, and the number of key/value pairs as count. The two differ in how the key/value values are communicated. In the first form, `adr_name_create`, two `char` * arrays are provided, one for keys and the other for values, as shown in the following example.

EXAMPLE 5-3 `adr_name_create`

```
const char *keys[] = { "key1", "key2" };
const char *values[] = { "value1", "value2" };
name = adr_name_create("com.example", 2, keys, values);
```

In the second form, `adr_name_vcreate`, keys and values are provided as alternating varargs. The previous example written using `adr_name_vcreate` would look like the following example.

EXAMPLE 5-4 `adr_name_vcreate`

```
name = adr_name_vcreate("com.example", 2, "key1", "value1", "key2", "value2");
```

If either routine fails to create the `adr_name_t`, it will return `NULL`. All data provided to `adr_name_create` is copied and can subsequently be modified or freed without affecting existing `adr_name_t` types.

Inspecting `adr_name_t` Type

`adr_name_t` types are immutable, so all operations on them are read-only. The two most common operations one needs to perform on an `adr_name_t` are obtaining the name's domain and obtaining the value associated with a particular key.

```
const char *adr_name_domain(const adr_name_t *name);
const char *adr_name_key(const adr_name_t *name, const char *key);
```

`adr_name_domain` returns name's reverse-dotted domain as a string. The string returned is part of name and therefore must not be modified or freed, and must not be accessed after the caller's reference on name has been released. Likewise, `adr_name_key` returns the value associated with key. The string returned by `adr_name_key` is subject to the same restrictions as the return value of `adr_name_domain`.

The two functions for comparing `adr_name_t` types are:

```
int adr_name_cmp(const adr_name_t *name1, const adr_name_t *name2);

boolean_t adr_name_match(const adr_pattern_t *pattern, const adr_name_t *name);
```

adr_name_cmp compares two adr_name_t types, returning 0 if the name1 and name2 are equal (that is, if the two names have the same domain, same names and the same keys, and each key has the same value on both names). It returns an integer less than 0 if name1 is less than name2, or and integer greater than 0 if name1 is greater than name2.

adr_name_match is a pattern-matching operation. The adr_name_t pattern is treated as a collection of attributes against which name is compared. adr_name_match returns B_TRUE if and only if the domains of name and pattern are equal, and every key present in pattern is present in name and has the same value. While an adr_name_t must have a domain and at least one key/value pair, pattern is permitted to have only a domain and no key/value pairs.

String Representation

It is sometimes necessary to represent, either in human-readable output or in persistent storage, an ADR object name as a string. libadr provides routines for converting to a canonical string form.

```
char *adr_name_tostr(const adr_name_t *name);
```

adr_name_tostr takes an adr_name_t and formats it in string form. The return value is allocated using malloc and should be freed when the caller is done with it. adr_name_tostr will return NULL if it is unable to allocate memory for its return value.

API Management

libadr provides support for defining APIs in rad/adr_object.h. Defining an API is a complex task. The only supported way to define an API is to do so in the ADR IDL and to generate the definition using radadrgen.

The important type defined in rad/adr_object.h is type adr_object_t. While the constituent pieces of an API definition should be considered implementation details, the end product, the API itself, is of prime interest to the developer. You will never need to create or define an adr_object_t, but when you encounter routines that operate on them, understanding what the type represents is important.

radadrgen-Generated Definitions

Whether you are using libadr in a C-based client or as part of writing a rad server module, you will need to understand the data definitions generated by radadrgen. Fortunately, the definitions are the same in both environments.

Running radadrgen

radadrgen is instructed to produce definitions for C/libadr consumers by using its -c option, as shown in the following example.

EXAMPLE 5-5 Invoking radadrgen

```
$ radadrgen -l c -s server -d output_dir example.adr
```

The -c option produces two files, api_APINAME.h and api_APINAME_impl.c in the *output_dir*, where APINAME is derived from the name attribute of the API document's api element. api_APINAME_impl.c contains the implementation of the interfaces and data types defined by the API. It should be compiled and linked with the software needing those definitions.

api_APINAME.h externs the specific symbols defined by api_APINAME_impl.c that consumers will need to reference, and should be included by those consumers. api_APINAME.h contains no data definitions itself and may be included in as many places as necessary. Neither file should be modified.

For each derived type TYPE, whether enumeration or structure, defined in the API, an adr_type_t named t__TYPE (two underscores) representing that type is generated and externed by the header file. If an array of that type is used anywhere in the API, an adr_type_t named t_array__TYPE (one underscore, two underscores) representing that array type is generated and externed. For each interface INTERFACE defined in the file, an adr_object_t named interface_INTERFACE is defined and externed.

For each value VALUE of an enumeration named TYPE , an adr_data_t named e__TYPE_VALUE is defined and externed. These adr_data_t values are marked as constants and are not affected by adr_data_ref or adr_data_free.

Example radadrgen Output

When radadrgen is run on the Example 4-7 given in the ADR chapter two files result. One, api_example_impl.c, holds the implementation of the GrabBag interface and data types it depends on, and should be simply be compiled and linked with the GrabBag consumer. The other, api_example.h, exposes only the relevant symbols defined by api_example_impl.c and should be included by consumers of the GrabBag interface and its related types as shown in the following example.

EXAMPLE 5-6 Sample radadrgen-Generated C Header File

```
#include <rad/adr.h>
```

```
#include <rad/adr_object.h>
#include <rad/rad_modapi.h>

extern adr_type_t t__Mood;
extern adr_data_t e__Mood_IRREVERENT;
extern adr_data_t e__Mood_MAUDLIN;
extern adr_type_t t__SqrtError;
extern adr_type_t t__StringInfo;
extern adr_type_t t__MoodStatus;
extern adr_object_t interface_GrabBag;
```

A consumer who needs to create a MoodStatus structure indicating the mood is `IRREVERENT` and has changed, would issue the instructions shown in the following example.

EXAMPLE 5-7 Consuming `radadrgen`-Generated Definitions

```
status = adr_data_new_struct(&t__MoodStatus);
adr_struct_set(status, "mood", e__Mood_IRREVERENT);
/* adr_struct_set(status, "mood", adr_data_new_enum_byname(&t__Mood, "IRREVERENT")); */
adr_struct_set(status, "changed", adr_data_new_boolean(B_TRUE));

if (!adr_data_verify(status, NULL, B_TRUE)) {
        ...
```

In addition to showing how to use the type definitions, this example also illustrates the multiple ways of referencing an enumerated value. Using the defined symbols is faster and can be checked by the compiler. The commented-out line uses `adr_data_new_enum_byname` which offers flexibility that could be useful in some situations but necessarily defers error checking until runtime. For example, if you mistype the value `IRREVERENT`, it would not be detected until the code is run. It is preferable to use the enumerated value symbols when possible.

♦ ♦ ♦ CHAPTER 6

Module Development

rad is modular in a variety of ways. Modules may deliver new protocols, new transports, or new API definitions and implementations. This section focuses on new API definitions and implementations.

API Definitions and Implementation

Although an API can be constructed manually, using radadrgen to generate the necessary type definitions is much simpler.

Entry Points

All entry points take a pointer to the object instance and a pointer to the internal structure for the method or attribute. The object instance pointer is essential for distinguishing different objects that implement the same interface. The internal structure pointer is theoretically useful for sharing the same implementation across multiple methods or attributes, but isn't used and may be removed.

Additionally, all entry reports return a conerr_t. If the access is successful, they should return CE_OK. If the access fails due to a system error, they should return CE_SYSTEM. If the access fails due to an expected error which should be noted in the API definition, they should return CE_OBJECT. If an expected error occurs and an error payload is defined, it may be set in *error. The caller will unref the error object when it is done with it.

- A method entry point has the type meth_invoke_f:

  ```
  typedef conerr_t (meth_invoke_f)(rad_instance_t *inst, adr_method_t *meth,
      adr_data_t **result, adr_data_t **args, int count, adr_data_t **error);
  ```

  args is an array of count arguments.

  Upon successful return, *result should contain the return value of the method, if any.

  The entry point for a method named METHOD in interface INTERFACE is named interface_INTERFACE_invoke_METHOD.

- An attribute read entry point has the type `attr_read_f`:

  ```
  typedef conerr_t (attr_read_f)(rad_instance_t *inst, adr_attribute_t *attr,
  adr_data_t **value, adr_data_t **error);
  ```

  Upon successful return, `*value` should contain the value of the attribute, if any.

  The read entry point for an attribute named `ATTR` in interface `INTERFACE` is named `interface_INTERFACE_read_ATTR`.
- An attribute write entry point has the type `attr_write_f`:

  ```
  typedef conerr_t (attr_write_f)(rad_instance_t *inst, adr_attribute_t *attr,
      adr_data_t *newvalue, adr_data_t **error);
  ```

  `newvalue` points to the new value. If the attribute is nullable, `newvalue` can be `NULL`.

  The write entry point for an attribute named `ATTR` in interface `INTERFACE` is named `interface_INTERFACE_write_ATTR`.

`rad` explicitly checks the types of all arguments passed to methods and all values written to attributes. Stub implementations can assume that all data provided is of the correct type. Stub implementations are responsible for returning valid data. Returning invalid data results in an undefined behavior.

Global Variables

Variables	Description
`boolean_t rad_isproxy`	A flag to determine if code is executing in the main or proxy `rad` daemon. Only special system modules, which are integral to the operation of RAD, may use this variable.
`rad_container_t *rad_container`	The rad container that contains the object instance.

Module Registration

Function	Description
`int _rad_init(void *handle);`	A module must provide a `_rad_init`. This is called by the `rad` daemon when the module is loaded and is a convenient point for module initialization including registration. Return

Function	Description
	`0` to indicate that the module successfully initialized.
`int rad_module_register(void *handle, int version, rad_modinfo_t *modinfo);`	`rad_module_register` provides a handle, which is the handle provided to the module in the call to `_rad_init`. This handle is used by the `rad` daemon to maintain the private list of loaded modules. The version indicates which version of the `rad` module interface the module is using. `modinfo` contains information used to identify the module.

Instance Management

Function	Description
`rad_instance_t *rad_instance_create(rad_object_type *type, void *data, void (*)(void *)freef);`	`rad_instance_create` uses the supplied parameters to create a new instance of an object of type. `data` is the user data to store with the instance and the freef function is a callback which will be called with the user data when the instance is removed. If the function fails, it returns `NULL`. Otherwise, a valid instance reference is returned.
`void * rad_instance_getdata(rad_instance_t *instance);`	`rad_instance_getdata` returns the user data (supplied in `rad_instance_create`) of the instance.
`void rad_instance_notify (rad_instance_t *instance, const char *event, long sequence, adr_data_t *data);`	`rad_instance_notify` generates an event on the supplied `instance`. The `sequence` is supplied in the event as the sequence number and the payload of the event is provided in data.

Container Interactions

Function	Description
`conerr_t rad_cont_insert(rad_container_t *container, adr_name_t *name, rad_instance_t *instance);`	Create an instance, `rad_instance_t`, using the supplied name and object and then insert into container. If the operation succeeds, `CE_OK` is returned.
`conerr_t rad_cont_insert_singleton(rad_container_t *container, adr_name_t *name, rad_object_t *object);`	
`void rad_cont_remove(rad_container_t *container, adr_name_t *name);`	Remove the `instance` from the container.

Function	Description
`conerr_t rad_cont_register_dynamic(rad_container_t *container, adr_name_t *name, rad_modinfo_t *modinfo, rad_dyn_list_t listf, rad_dyn_lookup_t lookupf, void *arg);`	Register a dynamic container instance manager. The container defines the container in which the instances will be managed. The `name` defines the name filter for which this instance manager is responsible. A typical name would define the type of the instance which are managed. For example, `zname = adr_name_vcreate (MOD_DOMAIN, 1, "type", "Zone")` would be responsible for managing all instances with a type of "Zone". `listf` is a user-supplied function which is invoked when objects with the matching pattern are listed. `lookupf` is a user-supplied function which is invoked when objects with the matching name are looked up. arg is stored and provided in the callback to the user functions.
`conerr_t (*rad_dyn_list_t)(adr_pattern_t *pattern, adr_data_t **data, void *arg);`	
`conerr_t (*rad_dyn_lookup_t)(adr_name_t **name, rad_instance_t **inst, void *arg);`	

Logging

Function	Description

`void rad_log(rad_logtype_t type,`
`const char * format, ...);`
	Log a message with type and format to the `rad` log. If the type is a lower level than the `rad` logging level, then the message is discarded.

`void rad_log_alloc()`	Log a memory allocation failure with log level RL_FATAL.
`rad_logtype_t rad_get_loglevel()`	Return the logging level.

Using Threads

Function	Description
`void *rad_thread_arg(rad_thread_t *tp);`	Return the `arg` referenced by the thread `tp`.

`void rad_thread_ack(rad_thread_t *tp,`
`rad_moderr_t error);`
	This function is intended to be used from a user function previously supplied as an argument to `rad_thread_create`. It should not be used in any other context.
	Acknowledge the thread referenced by `tp`. This process enables the controlling thread, from which a new thread was

Function	Description
	created using `rad_thread_create`, to make progress. The error is used to update the return value from `rad_thread_create` and is set to `RM_OK` for success.
`rad_moderr_t rad_thread_create(rad_threadfp_t fp, void *arg);`	Create a thread to run `fp`. This function will not return until the user function (`fp`) calls `rad_thread_ack. arg` is stored and passed into `fp` as a member of the `rad_thread_t` data. It can be accessed using `rad_thread_arg`.
`rad_moderr_t rad_thread_create_async( rad_thread_asyncfp_t fp, void *arg);`	Create a thread to run `fp. arg` is stored and passed into `fp`.

Synchronization

Function	Description
`void rad_mutex_init(pthread_mutex_t *mutex);`	Initialize a `mutex.abort` on failure.
`void rad_mutex_enter(pthread_mutex_t *mutex);`	Lock a `mutex. abort` on failure.
`void rad_mutex_exit(pthread_mutex_t *mutex);`	Unlock a `mutex. abort` on failure.
`void rad_cond_init(pthread_cond_t *cond);`	Initialize a condition variable, `cond. abort`, on failure.

Subprocesses

Function	Description
`exec_params_t *rad_exec_params_alloc`	Allocate a control structure for executing a subprocess.
`void rad_exec_params_free(exec_params_t *params);`	Free a subprocess control structure, params.
`void rad_exec_params_set_cwd(exec_params_t *params, const char *cwd);`	Set the current working directory, `cwd`, in a subprocess control structure, `params`.
`void rad_exec_params_set_env(exec_params_t *params, const char **envp);`	Set the environment, `envp`, in a subprocess control structure, `params`.
`void rad_exec_params_set_loglevel( exec_params_t *params, rad_logtype_t loglevel);`	Set the rad log level, `loglevel`, in a subprocess control structure, `params`.

Function	Description
`int rad_exec_params_set_stdin(exec_params_t *params, int fd);`	Set the stdin file descriptor, `fd`, in a subprocess control structure, `params`.
`int rad_exec_params_set_stdout(exec_params_t *params, int fd);`	Set the stdout file descriptor, `fd`, in a subprocess control structure, `params`.
`int rad_exec_params_set_stderr(exec_params_t *params, int fd);`	Set the stderr file descriptor, `fd`, in a subprocess control structure, `params`.
`int rad_forkexec(exec_params_t *params, const char **argv, exec_result_t *result);`	Use the supplied subprocess control structure, `params`, to fork and execute (`execv`) the supplied args, `argv`. If result is not `NULL`, it is updated with the subprocess pid and file descriptor details.
`int rad_forkexec_wait(exec_params_t *params, const char **argv, int *status);`	Use the supplied subprocess control structure, `params`, to fork and execute (`execv`) the supplied args, `argv`. If status is not `NULL`, it is updated with the exit status of the subprocess. This function will wait for the subprocess to terminate before returning.
`int rad_wait(exec_params_t *params, exec_result_t *result, int *status);`	Use the supplied subprocess control structure, params, to wait for a previous invocation of `rad_forkexe` to complete. If result is not `NULL`, it is updated with the subprocess pid and file descriptor details. If status is not `NULL`, it is updated with the exit status of the subprocess. This function will wait for the subprocess to terminate before returning.

Utilities

Function	Description
`void *rad_zalloc(size_t size);`	Return a pointer to a zero-allocated block of size bytes.
`char *rad_strndup(char *string, size_t length);`	Create and return a duplicate of string that is of size, length bytes.
`int rad_strccmp(const char * zstring, const char * cstring, size_t length);`	Compare two strings, up to a maximum size of length bytes.
`int rad_openf(const char *format, int oflag, mode_tmode, ... );`	Open a file with access mode, oflag, and mode, mode, whose path is specified by calling `sprintf` on format.

Function	Description
`FILE *rad_fopenf(const char *format, const char *mode, ...);`	Open a file with mode, whose path is specified by calling `sprintf` on format.

Locales

Function	Description

`int rad_locale_parse(const char *locale,`
`rad_locale_t **rad_locale);`
	Update `rad_locale` with locale details based on locale. If locale is `NULL`, then attempt to retrieve a locale based on the locale of the `rad` connection. Returns 0 on success.

`void rad_locale_free(rad_locale_t *rad_locale);`	Free a locale, `rad_locale`, previously obtained with `rad_locale_parse`.

Transactional Processing

There is no direct support for transactional processing within a module. If a transactional model is desirable, then it is the responsibility of the module creator to provide the required building blocks, start_transaction, commit, rollback, and other related processes.

Asynchronous Methods and Progress Reporting

Asynchronous methods and progress reporting is achieved using threads and events. The pattern is to return a token from a synchronous method invocation which spawns a thread to do work asynchronously. This worker thread is then responsible for providing notifications to interested parties events.

Example:

An interface has a method which returns a Task object. The method is called `installpkg` and takes one argument, the name of the package to install.

```
Task installpkg(string pkgname);
```

The Task instance returned by the method, contains enough information to identify a task. Prior to invoking `installpkg`, the client subscribes to a task-update event. The worker thread is responsible for issuing events about the progress of the work. These events contain information about the progress of the task.

In a minimal implementation, the worker thread would issue one event to notify the client that the task was complete and what the outcome of the task was. A more complex implementation would provide multiple events documenting progress and possibly also provide an additional method that a client could invoke to interrogate the server for a progress report.

rad Namespaces

Objects in the rad namespace can be managed either as a set of statically installed objects or as a dynamic set of objects that are listed or created on demand.

Static Objects

rad_modapi.h declares two interfaces for statically adding objects to a namespace.

rad_cont_insert adds an object to the namespace. In turn, objects are created by calling rad_instance_create with a pointer to the interface the object implements, a pointer to object-specific callback data and a pointer to a function to free the callback data. For example:

```
adr_name _t *uname = adr_name_vcreate("com.oracle.solaris.rad.user", 1, "type", "User");
rad_instance_t *inst = rad_instance_create(&interface_User_svr, kyle_data, NULL);
(void) rad_cont_insert(&rad_container, uname, inst);
adr_name_rele(uname);
```

rad_cont_insert_singleton is a convenience routine that creates an object instance for the specified interface with the specified name and adds it to the namespace. The callback data is set to NULL.

```
adr_name _t *uname = adr_name_vcreate("com.oracle.solaris.rad.user", 1, "type", "User");
(void) rad_cont_insert_singleton(&rad_container, uname, &interface_User_svr);
adr_name_rele(uname);
```

rad Module Linkage

Modules are registered with the RAD daemon in the _rad_reg. This is automatically generated from the information contained within the IDL defining the module.

Each module is required to provide a function, _rad_init, for initializing the module. This function is called before any other function in the module. Similarly, the _rad_fini in the module is called by the RAD daemon just prior to unloading the module.

EXAMPLE 6-1 Module Initialization

```
#include <rad/rad_modapi.h>
```

```
int
_rad_init(void)
{
 adr_name _t *uname = adr_name_vcreate("com.oracle.solaris.rad.user", 1, "type", "User");
    conerr_t cerr = rad_cont_insert_singleton(&rad_container, uname, &interface_User_svr);
    adr_name_rele(uname);

 if (cerr != CE_OK)
 {
       rad_log(RL_ERROR, "failed to insert module in container");
    return(-1);
    }
 return (0);
}
```

♦ ♦ ♦ CHAPTER 7

rad Best Practices

This chapter provides guidance when using rad. The guidance material is grouped around the following topics.

- When to use rad?
- How to use rad?

When To Use rad?

rad is designed to provide remote administrative interfaces for operating system components/ sub-systems. Such interfaces support the distributed administration of systems and greatly increase the abilities of system administrators to support large installations.

It is not intended to be a general purpose mechanism for building distributed applications, many alternative facilities, for example, RPC, RMI, CORBA, and MPI exist for such applications.

How To Use rad?

This section contains specific guidance on how to use rad.

API Guidelines

Designing a rad API requires judgement and the application of domain knowledge.

Target Audience

The users of the API fall into two broad categories:

- Administrators

- Developers

Unfortunately, accommodating the desires of consumers in these two categories within one interface is difficult. The first group desire task-based APIs which match directly onto well-understood and defined administrative activities. The second group desire detailed, operation-based interfaces which may be aggregated to better support unusual or niche administrative activities.

For any given subsystem, you can view existing command-line utilities (CLIs) and libraries (APIs) as expressions of the `rad` APIs which are required. The CLIs represent the task-based administrative interfaces and the APIs represent the operation-based developer interfaces.

The goal in using `rad` is to provide interfaces that address the lowest-level objectives of the target audience. If targeting administrators (task-based), this effort could translate to matching existing CLIs. If targeting developers, this effort could mean significantly less aggregation of the lower-level APIs.

Legacy Constraints

Many subsystems present incomplete interfaces to the world. Some CLIs contain processing capabilities that are not accessible from an existing API. This situation is another motivation for providing task-based administrative interfaces before introducing more detailed interfaces.

Such constraints must be considered in the `rad` API design. Consider migrating functionality from the CLI into the API to facilitate the creation of the new interface. Also consider presenting an interface which wraps the CLI and takes advantage of the existing functionality. Do not simply duplicate the functionality in the new `rad` interface, which would introduce redundancy and significantly increase maintenance complexity. One particular area where `rad` interface developers need to be careful is to avoid duplication around parameter checking and transformation. This duplication is likely to be a sign that existing CLI functionality should be migrated to an API.

`rad` modules must be written in C. Some subsystems, for instance, those written in other languages, have no mechanism for a C module to access API functionality. In these cases, `rad` module creators must access whatever functionality is available in the CLI or make a potentially significant engineering effort to access the existing functionality, for example, rewriting existing code in C, embedding a language interpreter in their C module, and the like.

Conservative Design

Designing a `rad` interface is very similar to designing a library interface. The same general principles of design apply: be conservative, start small, consider evolutionary paths and carefully consider commitment levels.

Once an interface is established, the use of versioning and considered, incremental improvements will expand the functionality.

Component Guidelines

This section presents specific design advice on the most significant components of a rad module. Naming is addressed separately in “Naming Guidelines” on page 92

API Guidelines

APIs are the primary deliverable of a rad module. They are a grouping of interfaces, events, methods and properties which enable a user to interact with a subsystem.

When exposing the elements of a subsystem consider carefully how existing functions can be grouped together to form an interface. Imperative languages, such as C, tend to pass structures as the first argument to functions, which provides a clear indicator as to how best to group functions into APIs.

Method Guidelines

Methods provide mechanisms for examining and modifying administrative state.

Consider grouping together existing native APIs into aggregated rad functions which enable higher order operations to be exposed.

Follow established good practice for RPC style development. rad is primarily for remote administration, and avoiding excessive network load is good practice.

Property Guideline

Make sure to define an <error> element with properties which can be modified.

Event Guidelines

The module is responsible for providing a sequence number. Monotonically increasing sequence numbers are recommended for use, since these will be of most potential use to any clients.

Consider providing mechanisms for allowing a client to throttle event generation.

Carefully design event payloads to minimize network load.

Don't try to replicate the functionality of network monitoring protocols such as SNMP.

Synchronous and Asynchronous Invocation

All method invocations in `rad` are synchronous. Asynchronous behavior can be obtained by adopting a design pattern that relies on the use of events to provide notifications. Refer to "Synchronization" on page 83 for more details.

Duplication

Do not duplicate code from existing CLIs. Instead, consider moving common code into a lower library layer that can be shared by `rad` and the CLI.

Client Library Support

`rad` modules are designed to have a language agnostic interface. However, you might want to provide additional language support through the delivery of a language-specific extension. This type of deliverables should be restricted in use. The main reason for their existence is to help improve the fit of an interface into a language idiom.

Naming Guidelines

When naming an API, interface, or "object" on page 19, module developers have broad leeway to choose names that make sense for their modules. However, some conventions can help avoid pitfalls that might arise when retrieving objects from the `rad` server.

Object Names

The domain portion of `rad` object names follows a reverse-dotted naming convention that prevents collisions in `rad`'s flat object namespace. This convention typically resembles a Java package naming scheme:

```
com.oracle.solaris.rad.zonemgr
com.oracle.solaris.rad.usermgr
org.opensolaris.os.rad.ips
...
```

To distinguish a `rad` API from a native API designed and implemented for a specific language, include a "`rad.`" component in the API name.

With the goal of storing objects with names consumers would expect, APIs, and the domains of the objects defined within them, should share the same name. This practice makes the mapping between the two easily identifiable by both the module consumer and module developer.

With the same goal of simplicity, identifying an interface object is made easier by adhering to a "type=interface" convention within the object name.

Applying both conventions, a typical API will look like the following example.

```
    <api  xmlns="http://xmlns.oracle.com/radadr"
       name="com.oracle.solaris.rad.zonemgr">
      <version major="1" minor="0"/>
      <interface name="ZoneInfo"> <!-- Information about the current zone  -->
      <property name="name" access="ro" type="integer"/>

          ...

        </interface>
</api>
```

Within the module, the API appears as follows:

```
int
_rad_init(void)

    {
    ...
          adr_name _t *zname = adr_name_vcreate(MOD_DOMAIN, 1, "type", "ZoneInfo");
          conerr_t cerr = rad_cont_insert_singleton(&rad_container, zname, &interface_ZoneInfo_svr);
          adr_name_rele(zname);


         if (cerr != CE_OK) {
             rad_log(RL_ERROR, "failed to insert module in container");
             return(-1);
         }
        return (0);
}
```

On the consumer side (Python), the API appears as follows:

```
import rad.connect as radcon
import rad.bindings.com.oracle.solaris.rad.zonemgr as zonemgr

# Create a connection and retrieve the ZoneInfo object
with radcon.connect_unix() as rc:
    zinfo = rc.get_object(zonemgr.ZoneInfo())
    print zinfo.name
```

Case

In an effort to normalize the appearance of like items across development boundaries, and to minimize the awkwardness in generated language-specific interfaces, several case strategies have been informally adopted.

Module

The base of the API/domain name. For a module describing an interface *domain.prefix.base*`.adr`, module spec files should be named *base*`.adr`, and the resulting shared library `mod_`*base*`.so`.

Examples:

- `/usr/lib/rad/interfaces/zonemgr/version/1/zonemgr.adr`
- `/usr/lib/rad/module/mod_zonemgr.so`

API

Reverse-dotted domain, all lowercase.

Examples:

- `com.oracle.solaris.rad.usermgr`
- `com.oracle.solaris.rad.zonemgr`

Interface, struct, union, enum

Non-qualified, camel case, starting with uppercase.

Examples:

- `Time`
- `NameService`
- `LDAPConfig`
- `ErrorCode`

Enum value and fallback

Non-qualified, uppercase, underscores.

Examples:

- `CHAR`
- `INVALID_TOKEN`
- `REQUIRE_ALL`

Interface property and method, struct field, event

Non-qualified, camel case, starting with lowercase.

Examples:

- `count`
- `addHostName`
- `deleteUser`

API Design Examples

Combining the tools described so far in this document to construct an API with a known design can be a challenge. Several possible solutions for a particular problem are often available. The examples in this section illustrate the best practices described in previous sections.

Note - This is only an example. This means it does not reflect the user management modules that is in Oracle Solaris.

User Management Example

Object/interface granularity is subjective. For example, imagine an interface for managing a user. The user has a few modifiable properties:

TABLE 7-1 Example User Properties

Property	Type
name	string
shell	string
admin	boolean

The interface for managing this user might consist solely of a set of attributes corresponding to the above properties. Alternatively, it could consist of a single attribute that is a structure containing fields that correspond to the properties, possibly more efficient if all properties are usually read or written together. The object implementing this might be named as follows:

```
com.example.users:type=TheOnlyUser
```

If instead of managing a single user you need to manage multiple users, you have a couple of choices. One option would be to modify the interface to use methods instead of attributes, and to add a "user" argument to the methods, for example:

```
setUserAttributes(username, attributes) throws UserError
attributes getUserAttributes(username) throws UserError
```

This example is sufficient for a single user, and provides support to other global operations such as adding a user, deleting a user, getting a list of users and so on. You might want to give it a more appropriate name, for example:

```
com.example.users:type=UserManagement
```

However, suppose there were many more properties associated with the user and many more operations you would want to do with a user, for example, sending them email, giving them a bonus and so on. As the server functionality grows, the UserManagement's API grows increasingly cluttered. It would accumulate a mixture of global operation and per-user operations, and the need for each per-user operation to specify a user to operate on, and specify the errors associated with not finding that user, would start looking redundant.

```
username[] listUsers()
addUser(username, attributes)
giveRaise(username, dollars) throws UserError
fire(username) throws UserError
sendEmail(username, message) throws UserError
setUserAttributes(username, attributes) throws UserError
attributes getUserAttributes(username) throws UserError
```

A cleaner alternative would be to separate the global operations from the user-specific operations and create two interfaces. The UserManagement object would use the global operations interface:

```
username[] listUsers()
addUser(username, attributes)
```

A separate object for each user would implement the user-specific interface:

```
setAttributes(attributes)
attributes getAttributes()
giveRaise(dollars)
fire()
sendEmail(message)
```

Note - If fire operates more on the namespace than the user, it should be present in UserManagement where it would need to take a username argument.

Finally, the different objects would be named such that the different objects could be easily differentiated and be directly accessed by the client:

```
com.example.users:type=UserManagement
com.example.users:type=User,name=ONeill
com.example.users:type=User,name=Sheppard
...
```

This example also highlights a situation where the rad server may not want to enumerate all objects when a client issues a `LIST` request. Listing all users may not be particularly expensive, but pulling down a list of potentially thousands of objects on every `LIST` call will not benefit the majority of clients.

♦ ♦ ♦ APPENDIX A

zonemgr ADR Interface Description Language

The following example describes the APIs used in zonemgr ADR Interface Description Language. This is only an example.

Note - This is only an example and may not reflect the actual implementation of zonemgr APIs in Oracle Solaris.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>

<api xmlns="http://xmlns.oracle.com/radadr" name="com.oracle.solaris.rad.zonemgr">
  <version major="1" minor="0"/>
  <enum name="ErrorCode">
    <value name="NONE" value="0"/>
    <value name="FRAMEWORK_ERROR"/>
    <value name="SNAPSHOT_ERROR"/>
    <value name="COMMAND_ERROR"/>
    <value name="RESOURCE_ALREADY_EXISTS"/>
    <value name="RESOURCE_NOT_FOUND"/>
    <value name="RESOURCE_TOO_MANY"/>
    <value name="RESOURCE_UNKNOWN"/>
    <value name="ALREADY_EDITING"/>
    <value name="PROPERTY_UNKNOWN"/>
    <value name="NOT_EDITING"/>
    <value name="SYSTEM_ERROR"/>
    <value name="INVALID_ARGUMENT"/>
    <value name="INVALID_ZONE_STATE"/>
  </enum>
  <struct name="Result" stability="private">
    <field typeref="ErrorCode" name="code" nullable="true"/>
    <field type="string" name="str" nullable="true"/>
    <field type="string" name="stdout" nullable="true"/>
    <field type="string" name="stderr" nullable="true"/>
  </struct>
  <struct name="ConfigChange">
    <field type="string" name="zone"/>
  </struct>
  <struct name="StateChange">
    <field type="string" name="zone"/>
    <field type="string" name="oldstate"/>
    <field type="string" name="newstate"/>
  </struct>
  <enum name="PropertyValueType">
    <value name="PROP_SIMPLE"/>
    <value name="PROP_LIST"/>
    <value name="PROP_COMPLEX"/>
  </enum>
  <struct name="Property">
    <field name="name" type="string"/>
    <field name="value" type="string" nullable="true"/>
    <field name="type" typeref="PropertyValueType" nullable="true"/>
    <field name="listvalue" nullable="true">
      <list type="string"/>
    </field>
    <field name="complexvalue" nullable="true">
      <list type="string"/>
    </field>
  </struct>
  <struct name="Resource">
    <field type="string" name="type"/>
    <field name="properties" nullable="true">
      <list typeref="Property"/>
    </field>
    <field name="parent" type="string" nullable="true"/>
  </struct>
  <interface name="ZoneManager">
    <method name="create">
      <result typeref="Result"/>
      <error typeref="Result"/>
      <argument name="name" type="string"/>
      <argument name="path" type="string" nullable="true"/>
      <argument name="template" type="string" nullable="true"/>
    </method>
    <method name="delete">
      <result typeref="Result"/>
      <error typeref="Result"/>
      <argument name="name" type="string"/>
    </method>
    <method name="importConfig">
      <result typeref="Result"/>
      <error typeref="Result"/>
      <argument name="noexecute" type="boolean"/>
      <argument name="name" type="string"/>
      <argument name="configuration">
        <list type="string"/>
      </argument>
    </method>
    <event typeref="StateChange" name="stateChange"/>
  </interface>
  <interface name="ZoneInfo">
    <property name="brand" access="ro" type="string"/>
    <property name="id" access="ro" type="integer"/>
    <property name="uuid" access="ro" type="string" nullable="true">
      <error typeref="Result"/>
    </property>
    <property name="name" access="ro" type="string"/>
    <property name="isGlobal" access="ro" type="boolean"/>
  </interface>
  <interface name="Zone">
    <name key="name" primary="true"/>
    <name key="id"/>
```

```
<property name="auxstate" access="ro" nullable="true">
  <list type="string"/>
  <error typeref="Result"/>
</property>
<property name="brand" access="ro" type="string"/>
<property name="id" access="ro" type="integer"/>
<property name="uuid" access="ro" type="string" nullable="true">
  <error typeref="Result"/>
</property>
<property name="name" access="ro" type="string"/>
<property name="state" access="ro" type="string"/>
<method name="cancelConfig">
  <error typeref="Result"/>
</method>
<method name="exportConfig">
  <result type="string"/>
  <error typeref="Result"/>
  <argument name="includeEdits" type="boolean" nullable="true"/>
  <argument type="boolean" name="liveMode" nullable="true"/>
</method>
<method name="update">
  <error typeref="Result"/>
  <argument name="noexecute" type="boolean"/>
  <argument name="commands">
    <list type="string"/>
  </argument>
</method>
<method name="editConfig">
  <error typeref="Result"/>
  <argument type="boolean" name="liveMode" nullable="true"/>
</method>
<method name="commitConfig">
  <error typeref="Result"/>
</method>
<method name="configIsLive">
  <result type="boolean"/>
</method>
<method name="configIsStale">
  <result type="boolean"/>
  <error typeref="Result"/>
</method>
<method name="addResource">
  <error typeref="Result"/>
  <argument name="resource" typeref="Resource"/>
  <argument name="scope" typeref="Resource" nullable="true"/>
</method>
<method name="reloadConfig">
  <error typeref="Result"/>
  <argument type="boolean" name="liveMode" nullable="true"/>
</method>
<method name="removeResources">
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="false"/>
  <argument name="scope" typeref="Resource" nullable="true"/>
</method>
<method name="getResources">
  <result>
    <list typeref="Resource"/>
  </result>
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="true"/>
  <argument name="scope" typeref="Resource" nullable="true"/>
</method>
<method name="getResourceProperties">
  <result>
    <list typeref="Property"/>
  </result>
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="false"/>
  <argument name="properties" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="setResourceProperties">
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="false"/>
  <argument name="properties" nullable="false">
    <list typeref="Property"/>
  </argument>
</method>
<method name="clearResourceProperties">
  <error typeref="Result"/>
  <argument name="filter" typeref="Resource" nullable="false"/>
  <argument name="properties" nullable="false">
    <list type="string"/>
  </argument>
</method>
<method name="apply">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="attach">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="boot">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="clone">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="detach">
```

```
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="halt">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="install">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="mark">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="move">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="rename">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="ready">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="reboot">
  <result typeref="Result"/>
  <error typeref="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="savecore">
  <result typeref="Result"/>
  <error typeref="Result"/>
```

```
      <argument name="options" nullable="true">
        <list type="string"/>
      </argument>
    </method>
    <method name="shutdown">
      <result typeref="Result"/>
      <error typeref="Result"/>
      <argument name="options" nullable="true">
        <list type="string"/>
      </argument>
    </method>
    <method name="suspend">
      <result typeref="Result"/>
      <error typeref="Result"/>
      <argument name="options" nullable="true">
        <list type="string"/>
      </argument>
    </method>
    <method name="uninstall">
      <result typeref="Result"/>
      <error typeref="Result"/>
      <argument name="options" nullable="true">
        <list type="string"/>
      </argument>
    </method>
    <method name="verify">
      <result typeref="Result"/>
      <error typeref="Result"/>
      <argument name="options" nullable="true">
        <list type="string"/>
      </argument>
    </method>
    <event typeref="ConfigChange" name="configChange"/>
  </interface>
</api>
```