Remote Administration Daemon Module Developer's Guide

E61054-02
November 2020

Remote Administration Daemon Module Developer's Guide,

E61054-02

Copyright © 2012, 2020, Oracle and/or its affiliates.

Primary Author: Cathleen Reiher, Sharon Veach

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

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, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

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.

Oracle®, Java, and MySQL are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about 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 unless otherwise set forth in an applicable agreement between you and Oracle. 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, except as set forth in an applicable agreement between you and Oracle.

Copyright © 2012, 2020, Oracle et/ou ses affiliés.

Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf stipulation expresse de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, accorder de licence, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

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, la documentation du logiciel, les données (telles que définies dans la réglementation "Federal Acquisition Regulation") ou la documentation qui l'accompagne sont livrés sous licence au Gouvernement des Etats-Unis, ou à quiconque qui aurait souscrit la licence de ce logiciel pour le compte du Gouvernement des Etats-Unis, la notice suivante s'applique :

UTILISATEURS DE FIN DU GOUVERNEMENT É.-U. : programmes Oracle (y compris tout système d'exploitation, logiciel intégré, tout programme intégré, installé ou activé sur le matériel livré et les modifications de tels programmes) et documentation sur l'ordinateur d'Oracle ou autres logiciels OracleLes données fournies aux utilisateurs finaux du gouvernement des États-Unis ou auxquelles ils ont accès sont des "logiciels informatiques commerciaux", des "documents sur les logiciels informatiques commerciaux" ou des "données relatives aux droits limités" conformément au règlement fédéral sur l'acquisition applicable et aux règlements supplémentaires propres à l'organisme. À ce titre, l'utilisation, la reproduction, la duplication, la publication, l'affichage, la divulgation, la modification, la préparation des œuvres dérivées et/ou l'adaptation des i) programmes Oracle (y compris tout système d'exploitation, logiciel intégré, tout programme intégré, installé, ou activé sur le matériel livré et les modifications de ces programmes), ii) la documentation informatique d'Oracle et/ou iii) d'autres données d'Oracle, sont assujetties aux droits et aux limitations spécifiés dans la licence contenue dans le contrat applicable. Les conditions régissant l'utilisation par le gouvernement des États-Unis des services en nuage d'Oracle sont définies par le contrat applicable à ces services. Aucun autre droit n'est accordé au gouvernement américain.

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 un risque de 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 des applications dangereuses.

Oracle®, Java, et MySQL sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut être une marque appartenant à un autre propriétaire qu'Oracle.

Intel et Intel Inside sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées de SPARC International, Inc. AMD, Epyc, et le logo AMD sont des marques ou des marques déposées d'Advanced Micro Devices. UNIX est une marque déposée de The Open Group.

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 et excluent toute garantie expresse ou implicite quant aux contenus, produits ou services émanant de tiers, sauf mention contraire stipulée dans un contrat entre vous et Oracle. 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, sauf mention contraire stipulée dans un contrat entre vous et Oracle.

Contents

3 libadr Library

4 RAD Module Development

A zonemgr ADR Interface Description Language Example

Index

Using This Documentation

Product Documentation Library

Documentation and resources for this product and related products are available at http://www.oracle.com/pls/topic/lookup?ctx=E37838-01.

Feedback

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

1

Introduction to Developing a Remote Administration Daemon Module

Remote Administration Daemon

RAD provides programmable interfaces that enable developers and administrators to configure and manage Oracle Solaris system components. You can configure and manage system components using C, Java, Python, and REpresentational State Transfer (REST) APIs. RAD also enables developers to create custom interfaces using these APIs to manage the system components. RAD 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 such activities. For more information, see *Remote Administration Daemon Client Guide*.

A RAD *interface* defines how a client can interact with a system through a set of methods, attributes, and events using a structured namespace. The interface enables developers and administrators to configure and administer Oracle Solaris. Developers create these interfaces and program the access to them.

RAD uses a client-server design to support different types of clients such as clients written in different languages, clients running without privilege, and clients running remotely. In a client-server design, RAD acts as a server that services remote procedure calls and clients act as consumers.

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 resorting to a CLI-based implementation. By establishing a stream protocol, RAD enables the consumers to perform actions on any system or device over a range of secure transport options.

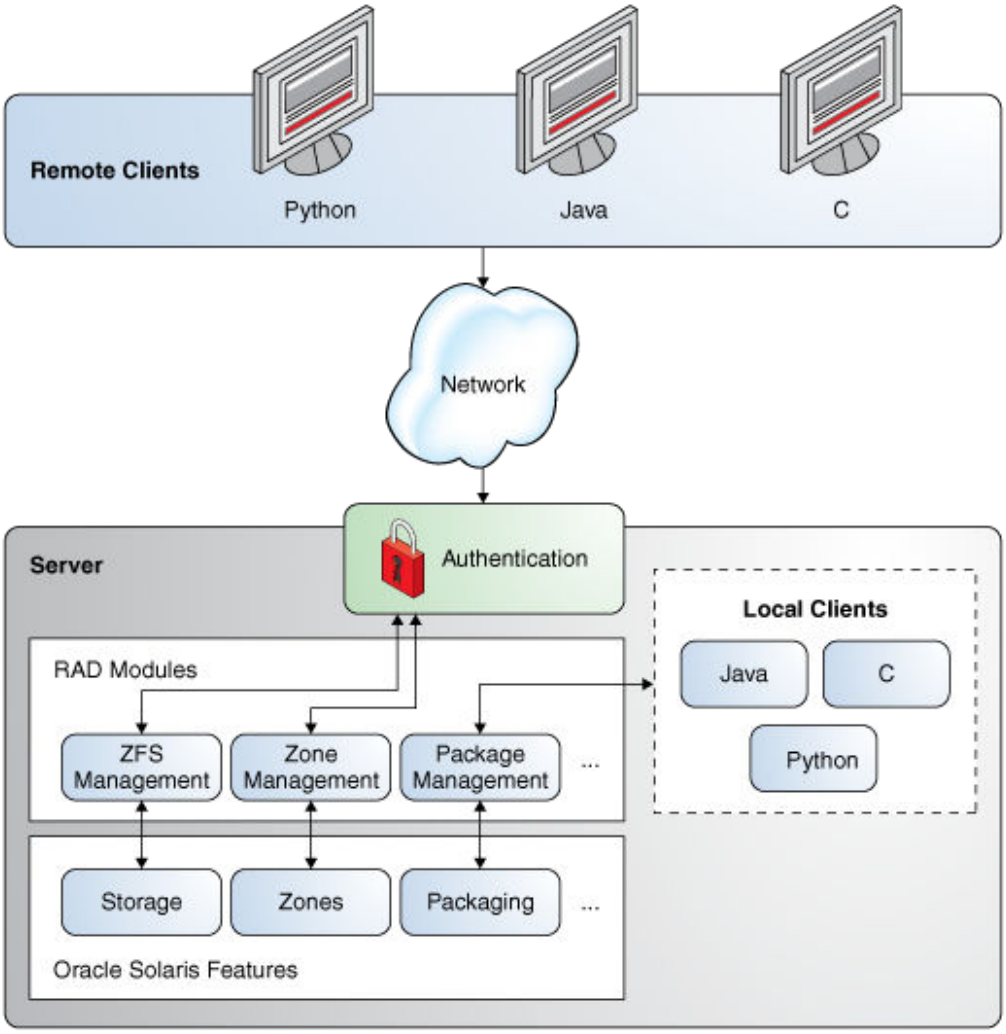
RAD offers the following benefits:

- Procedure calls in RAD are made against server objects in a browsable, structured namespace. This process permits a logical program progression.
- Procedure calls can be asynchronous. Depending on the protocol in use, a client might have multiple simultaneous outstanding requests.
- You can inspect and modify the interfaces exported by the server objects. This inspection facilitates interactive usage, debugging environments, and enables clients to use dynamically-typed languages such as Python.
- Using RAD interfaces, you can define properties and asynchronous event sources.

How RAD Works

In the RAD architecture, the clients can be local or remote. These clients can be written in C, Java, or Python. The following figure shows the architecture of RAD.

Architecture of RAD



RAD Functionality

RAD provides the following main functionalities:

- **Management and Configuration**
  - Two SMF services: `svc:/system/rad:local` and `svc:/system/rad:remote`
  - Structured and browsable namespace.
  - Inspectable, typed, and versioned interfaces.
  - Asynchronous event sources.
  - XML-based interactive data language (IDL) abstract data representation (ADR) that supports formal definitions of 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 by using GSSAPI in deployments where Kerberos is configured.
  - Implicit authentication by using `getpeerucred`(3C) when possible.
  - Non-local network connectivity is not available by default. RAD is preconfigured to use TLS.
  - Most operations are automatically delegated to lesser-privileged processes.
  - Defines two authorizations and two rights profiles 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 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.
  - Customizes the process attributes for each RAD module to conform to the Principle of Least Privilege.
- **Connectivity**
  - Local access by using `AF_UNIX` sockets.
  - Remote access by using TCP sockets.
  - Secure remote access by using TLS sockets.
  - Captive execution with access through a pipe.
  - Connection points are completely configurable at the command line or by using SMF.
- **Client support**
  - Java language binding provides access to all defined server interfaces.
  - Python language binding provides access to all defined server interfaces.
  - 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 enable the defining of new transports.

Designing RAD Components

The components 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.

RAD APIs

A RAD 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. APIs are versioned so that a client can specify which API version to use.

The users of the RAD APIs belong to two categories. administrators and developers. Accommodating both categories of consumers within one interface is difficult. Administrators require task-based APIs which match directly onto well-understood and defined administrative activities. Developers require 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. The CLIs represent the task-based administrative interfaces and the APIs represent the operation-based developer interfaces. The goal of using a RAD module is to provide interfaces that address the lowest-level objectives of the target audience. If you are targeting administrators (task-based), your goal could translate to matching existing CLIs. If you are targeting developers, your goal could mean significantly less aggregation of the lower-level APIs.

An API name defines the namespace, which identifies objects to the clients. APIs can have versions and a single RAD instance is capable of offering different major versions of APIs to different clients. RAD modules 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, you should consider 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. The structures provide a clear indication of where to group functions into APIs.

RAD API Versions

A version element is required for all APIs. For more information about API versions, see RAD Interface Versioning.

RAD API Namespace and Restricted Names

An API defines all top-level elements in a *namespace*. Names of components must be unique. Names must not begin with "`_rad`" because this string is reserved for the RAD toolchain.

Synchronous and Asynchronous Invocation in RAD

All method invocations in RAD are synchronous. Asynchronous behavior can be obtained by requiring events to provide notifications. For more information, see Synchronization in C for RAD.

Legacy Constraints for RAD APIs

Some CLIs contain processing capabilities that are not accessible from an existing API. Such constraints must be considered in the RAD API design.

Existing CLI functionality should be migrated to an API rather than duplicating the functionality in the new RAD interface. Duplication introduces redundancy and significantly increase maintenance complexity. One particular area where RAD interface developers need to be careful is around parameter checking and transformation.

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. Possible ways are to rewrite existing code in C or embed a language interpreter in their C module.

RAD 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. You should restrict the use of such extensions. Use them only to help improve the fit of an interface into a language idiom.

RAD API Design Examples

Combining these RAD tools to design an API can be a challenge. Several possible solutions for a particular problem are often available. The following User Management examples illustrate some best practices.

RAD User Management Example

> **Note:**
>
> This example does not reflect the user management modules in Oracle Solaris.

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

- `name` property of type `string`
- `shell` property of type `string`
- `admin` property of type `boolean`

The interface for managing this user might consist solely of a set of attributes corresponding to these properties. Alternatively, it could consist of a single attribute that is a structure containing fields that correspond to the properties. A possibly more efficient implementation

would read or write all properties 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 options. One option would be to modify the interface to use methods instead of attributes, and to add a "username" argument to the methods, for example:

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

This option is sufficient for a single user, as well as provides support to other global operations such as adding a user, deleting a user, getting a list of users and so on. This option could use a more appropriate name, for example:

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

However, suppose users have more properties and you want to perform more operations, such as, sending them email, giving them a bonus and so on. As the server functionality grows, the UserManagement's API becomes cluttered and the API will have code for both global and per-user operations. The need to specify a user and the associated errors for each per-user operation would start looking redundant.

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

A better alternative would be to create two interfaces that separate the global operations from the user-specific operations. 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)
rif()
sendEmail(message)
```

> **Note:**
>
> If `rif` 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 each object could 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 you do not want the RAD server to enumerate all objects when a client issues a `LIST` request. Pulling down a list of potentially thousands of objects on every `LIST` call does not benefit the majority of clients.

RAD Interface

An interface defines how a RAD client can interact with an object. Put another way, 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 subsystem has a data and state model tuned to the problems they are solving. Although the use of a common model across components offers major benefits, uniformity comes with trade-offs. A common model can be inefficient and create client complexity, thereby risking decreased developer adoption.

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. 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, a set of features a client may interact with, a set of derived types that are referenced by the features (optional), and a version. The features that are supported include:

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

RAD Interface Names

Each interface has a name. This name is used by the toolchain to construct identifier names when a generating code. When naming an API, interface, or object, module developers have broad leeway to choose illustrative names. However, some conventions can help avoid problems that might arise when retrieving objects from the RAD server.

RAD Object Naming Conventions

RAD object names follow naming conventions and should be easy for clients to recognize.

- 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.example.os.rad.ips
  ...
  ```

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

  ```
  name="com.oracle.solaris.rad.zonemgr"
  ```

- 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.

- Identifying an interface object is made simpler by adhering to a "`type=interface`" convention within the object name.

  ```
  property name="name" access="ro" type="integer"
  ```

A a typical API might look like the following:

```
<api  xmlns="https://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 would look like the following:

```
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 would look like the following:

```
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
```

RAD Component Naming Conventions

For consistency in the names of methods, interfaces, and property names across modules, use the following case guidelines.

Module
The base of the API or 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
```

RAD Feature Types

The common similarity between the three RAD feature types – methods, attributes, and events – is that they are named. Because the names of these three feature types exist in the same interface namespace, they must be unique. For example, you cannot have both a method and an attribute that is called `foo`. This exclusion avoids the majority of conflicts that could arise when mapping these interface features to a client environment. As in the API namespace, features must not begin with the reserved prefix "`_rad`".

> **Note:**
>
> Enforcing a common namespace for interface features is not 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 might cause a conflict.

RAD Methods

A RAD 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.

Each method argument has a name and a type. 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.

> **Note:**
>
> Method names cannot be overloaded.

The following are the guidelines for methods:

- 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.

RAD Property Attributes

A RAD 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

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.

An attribute's type value may 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.

Because methods exist in RAD, 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 offer a simpler interface.

The attribute mechanism has the following characteristics:

- Enforces symmetric access for reading and writing read-write attributes.
- Is easily and automatically translated to a form that is natural to the client language-environment.
- Communicates 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.

RAD Events

A RAD event is an asynchronous notification generated by RAD and consumed by clients. A client might 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.

The following are the guidelines for events:

- Provide a sequence number. Modules that provide a monotonically increasing sequence numbers are best, because such sequences are most useful to clients.
- Consider providing mechanisms for allowing a client to throttle event generation.
- Design event payloads to minimize network load.
- Do not duplicate the functionality of network monitoring protocols such as SNMP.

RAD Commitment Levels

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 levels independently.

Commitment levels suggest to API consumers 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.

RAD Interface Versioning

RAD interfaces are versioned for several 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 original API.
- 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.

RAD Version Numbering

RAD uses a `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.

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 RAD namespace acts as a gatekeeper by associating a name with each object, dispatching requests to the proper object, and providing meta-operations that enable the client to 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. You can see the namespace either as the place one goes to find objects or as 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.

RAD uses a structured namespace, as shown in RAD Object Naming Conventions. An object's name consists of a mandatory reverse-dotted domain combined with a non-empty set of key-value pairs. 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.

Some situations call for referring to groups of objects. In these situations, use a glob-style pattern or a `regex` style pattern. For more information, see Sophisticated RAD Searches in C in *Remote Administration Daemon Client User's Guide*.

Data Types Supported in RAD

All data returned, submitted to, or obtained from RAD APIs adhere to a strong typing system similar to that defined by XDR: External Data Representation Standard (https://www.rfc-editor.org/info/rfc4506). Strong typing simplifies defining interfaces that have precise semantics, and developing server extensions (which are written in C). Of course, the rigidity of the typing exposed to an API's consumer is primarily a function of the client language and implementation.

RAD 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.

RAD Derived Types

In addition to the base types, RAD supports the following derived types:

- Enumeration – 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 lack of interchangeability 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 bit-field values.
- Array – An ordered list of data items of a fixed type. Arrays do not have a predefined size.
- Structure – 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 enormous 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 mapped into simpler forms before consumption.

Optional Data in RAD

In some situations, data might 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` can be declared nullable. Additionally, only structure fields and certain API types can be nullable. Specifically, array data cannot be nullable because the array type in RAD is actually more like a list than an array.

RBAC Support for RAD

When a client application connects to a local or a remote RAD service, it initiates a new RAD slave process to execute remote procedure calls on behalf of the user and client. The RAD slave processes that are initiated by a normal user include basic privileges. However, if a RAD method wants to call a library function which requires `root` privileges, the user must authenticate as `root` to the RAD daemon before RBAC is added to RAD. This authentication requirement limits the utility of existing modules that have not been authenticated to. Also, all modules associated with a connection in legacy Oracle Solaris executed within a single slave process. Therefore, you could not associate module privileges at a process level.

Starting with Oracle Solaris 11.4, each module executes its own slave process, therefore you can apply process attributes for each module. This feature also ensures that each slave process can apply process attributes independently. To provide process attributes to a module, you create a rights profile that assigns privileges to a module. For details, see Creating a Rights Profile That Includes Privileged Commands in *Securing Users and Processes in Oracle Solaris 11.4*.

A possible `exec_attr` entry for a RAD User Security rights profile might display as follows:

```
RAD User Security:solaris:cmd:::/usr/lib/rad/module/mod_RADusermgr.so.1:privs=proc_zone
```

For more information, see the `privileges`(7) and `exec_attr`(5) man pages.

For more information RBAC, see Chapter 1, About Using Rights to Control Users and Processes in *Securing Users and Processes in Oracle Solaris 11.4*.

2

Abstract Data Representation for RAD

ADR Interface Description Language for RAD

RAD APIs use an XML-based IDL. The normative schema for this language are in the `/usr/share/lib/xml/rng/radadr.rng.1` file. The namespace name is `https://xmlns.oracle.com/radadr`.

ADR Definition Document for a RAD Module

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 definitions or interface definitions. Note that either a derived type or an interface must be defined.

Three derived types are available for definition and use by interfaces: a structured type that can be defined with a `struct` element, an enumeration type that can be defined with an `enum` element, and a dictionary type that can be defined with a `dictionary` 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.

The following is an example of an API.

Example 2-1 Skeleton RAD Module document

```
<api xmlns="https://xmlns.oracle.com/radadr" name="com.oracle.solaris.rad.example"
register="true">
  <summary>
    An API Example
  </summary>
  <doc>...</doc>

  <version/>

  <struct>...</struct>
  <struct>...</struct>
  <enum>...</enum>
  <interface>...</interface>
  <interface>...</interface>
</api>
```

The `xmlns` line is required to indicate the type of the XML document. The `name` attribute identifies the name of the API, the namespace within which all subsidiary interfaces are to be found. Additional attributes can 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` 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 used to create special types of

modules, such as protocol or transport modules. In general, it does not need to be specified, because the default generated function is enough for most purposes.

Documentation Definitions for RAD Modules

The documentation elements allow you to document the RAD module APIs and are defined by the schema specification in the `/usr/share/lib/xml/rng/radadr-doc.rng.1` file.

The two main documentation elements are:

`<summary />`
Container for inline elements

`<doc />`
Container for block elements

`<summary />` Element in RAD Modules

The `<summary />` element is a mandatory element. It represents a short text synopsis of the parent element. The `<summary />` element can only text data annotated with the inline elements described in the following list. The output of a `<summary />` element is running text with possible typographic modifications from the available inline elements:

`<code />`
Indicates small fragments of code.

`<emphasis />`
Emphasizes a phrase or word in italics.

`<strong />`
Emphasizes a phrase or word in bold.

`<link />`
Displays hypertext based on the following values for the hyperlink:

`url`
An external URL.

`interface, [method, property, event]`
An interface, method, property, or an event defined within the ADR document.

`enum, [value]`
An `enum` data type or an `enum` value defined within the ADR document.

`struct, [field]`
A `struct` data type or a `struct` field defined within the ADR document.

> **Note:**
>
> The text within the `<link>` element can be empty. If the value is empty, the text data is auto-generated based on the value of the method attribute.

<doc /> Element in RAD Modules

Use the <doc /> element to define larger blocks of content. The <doc /> element might contain inline elements, block elements or text data. The <doc /> element is displayed as separate blocks of data and can contain the following elements:

<heading />
Defines a section heading.

<para />
Defines a paragraph.

<list />
Defines a list of items. Items in the list are defined by an <item /> element. The <list /> element takes an optional attribute that defines the type of list to display using the ordered attribute for a numbered list or unordered attribute for a bullet list. The default list type is an unordered list.

<item />
Defines an item in a list. It might contain block or inline elements.

<example />
Displays a program listing. Available attributes are:

language
A mandatory attribute which defines the programming language. It can have any one of c, python, java, rest, or curl. The value must be in lowercase.

caption
An optional attribute, which provides a label for the example.

numbered
An optional attribute. Displays line numbers. Default is to omit line numbers.

<verbatim />
Defines a block of text in which line breaks and whitespace are preserved.

For more information about how to use the documentation definitions, see RAD Module Example.

Version Element in RAD Modules

A version element is required for all APIs.

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

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

This indicates that the module is starting at version 1.0.

Enumeration Definitions in RAD Modules

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 is not used elsewhere in the API definition, only in the server and various client environments. The symbolic name that is exposed in these environments are environment-dependent. An environment offering an explicit interface to RAD must 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` attribute 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 2-2 Defining Enumerations for a RAD Module

```
<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 is EOL -->
</enum>
```

Structure Definitions in RAD Modules

Similar to 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 or a derived type defined elsewhere in the API document, that type is defined with a `type` 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, or another nested `list` element.

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

> **Note:**
>
> The structure fields, methods return values, method arguments, attributes, error return values, and events have types, and in the IDL, use identical mechanisms for defining those types.

Example 2-3 Defining a `struct` for a RAD Module

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

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

Dictionary Definitions in RAD Modules

You can use dictionaries to add a data structure in which the key-value pair mappings can be stored and retrieved. The following example shows how to use the dictionary tag.

```
<dictionary>
  <key type="<key type>">
  <value type="<value type>">
</dictionary>
```

You can use the dictionary type similar to any other RAD type such as a field in a structure, a method argument or a return value, a property, an error payload, or as an event payload.

Example 2-4 Defining a Dictionary for RAD

This example shows how to define a dictionary with a key type of integer and value type of string as a read-write property.

```
...
  <property name="DictProp" access="rw" >
     <dictionary>
        <key type="integer" />
        <value type="string" />
     </dictionary>
  </property>
...
```

Values can be of any type except for `list` and `dictionary`. The value can be a derived type or a reference, use the `"type"` tag. However, the key must belong to any one of the following basic types:

- boolean
- integer
- unsigned integer
- long
- unsigned long
- float
- double
- time
- string

- name

Interface Definitions for a RAD Module

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 by using child elements of the `interface` element.

Interface Methods for a RAD Module

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 2-5 Defining a Method for a RAD Module

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

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

Interface Attributes for a RAD Module

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` as 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 2-6 Defining an Interface Attribute for a RAD Module

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

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

Interface Events for a RAD Module

Each event in an interface is defined by an `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 2-7 Defining an Event for a RAD Module

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

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

Including IDL Files in a Parent IDL File

ADR include feature allows you to include an XML-based IDL file within the parent IDL file. The following example shows how to include the `fragment.xml` file within the `parent.adr` file.

```
/*fragment.xml*/
<?xml version="1.0" encoding="UTF-8"?>
<fragment xmlns="https://xmlns.oracle.com/radadr">
<version major="1" minor="0"/>
    <para>
      Paragraph 1
    </para>
    <para>
      Paragraph 2
    </para>
</fragment>

/*parent.adr*/
 <?xml version="1.0" encoding="UTF-8"?>
 .
 .
 .
```

```
 .
<?include href="fragment.xml" major="1" minor="0"?>
```

The `<?include ?>` processing instruction has the following mandatory attributes:

`href`
Path to the included file.

`major`
Expected major version of the included fragment.

`minor`
Expected minor version of the included fragment

Version checking provides a warning in case of minor version mismatch and fail with an error in case of major version mismatch.

> **Note:**
>
> You cannot use the standard `xi:include` directive to include IDL files.

RAD Module Example

Example 2-8 Using Various RAD Module Elements

This example API demonstrates the use of various RAD module elements.

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

<api xmlns="https://xmlns.oracle.com/radadr"
name="com.oracle.solaris.rad.example">

 <summary>
    Example API
 </summary>

 <!-- A introductory paragraph to describe the API -->
 <doc>
   <para>
      This API defines PAM authentication methods that may be used to
authenticate
      a <strong>rad(8)</strong> client. <emphasis>NOTE: this is only an example
      and may not represent a working authentication interface!</emphasis>
   </para>
 </doc>

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

 <!-- An ADR enum type -->
 <!-- Each value in the enum has a short one-line description -->
 <enum name="BlockType" stability="private">
   <value name="CONV">
     <summary>
        conversation must continue
     </summary>
   </value>
   <value name="SUCCESS">
```

```
    <summary>
       authentication has succeeded
    </summary>
  </value>
  <value name="ERROR">
    <summary>
       authentication has failed
    </summary>
  </value>
</enum>

<!-- An ADR struct type -->
<!-- Each field in the struct has a short one-line description -->
<struct name="Block" stability="private">
  <field type="BlockType" name="type">
    <summary>
       the status of the conversation
    </summary>
  </field>
  <field name="messages" nullable="true">
    <summary>
       the messages to display to the user
    </summary>
    <list type="Message"/>
  </field>
</struct>

<!-- Another ADR enum type -->
<!-- Use of the verbatim tag to document the enum values -->
<enum name="MsgType" stability="private">
  <summary>Types of messages that may be returned from a PAM
   conversation </summary>
  <doc>
    <verbatim>
+-----------------+-------------------------------------+
| MsgType         | Action                              |
+-----------------+-------------------------------------|
| PROMPT_ECHO_OFF | Prompt the user for sensitive data, |
|                 | disabling echo of their response    |
| PROMPT_ECHO_ON  | Prompt the user for non-sensitive   |
|                 | data, echoing their response        |
| TEXT_INFO       | Print a general information message |
| TEXT_INFO       | Print an error message              |
+-----------------+-------------------------------------+
    </verbatim>
  </doc>
  <value name="PROMPT_ECHO_OFF" />
  <value name="PROMPT_ECHO_ON" />
  <value name="ERROR_MSG" />
  <value name="TEXT_INFO" />
</enum>

<!-- Another ADR struct type -->
<struct name="Message" stability="private">
  <field type="MsgType" name="style">
    <summary>
       this message's type
    </summary>
  </field>
  <field type="string" name="message">
    <summary>
```

```
        the message text
      </summary>
    </field>
  </struct>

  <interface name="Authentication" stability="private">
    <doc>
      <!-- A paragraph that describes the interface -->
      <para>
        The <code>Authentication</code> interface implements a PAM exchange to
        authenticate <strong>rad(8)</strong> clients.  Handles to this type of
        object can be retrieved from the RAD server using an object name built
        with:
      </para>

      <!-- An ordered list - items are numbered -->
      <list type="ordered">
        <item>
          the "<code>com.oracle.solaris.rad.pam</code>" domain name
        </item>
        <item>
          a key named "<code>type</code>" paired with a value of
          "<code>Authentication</code>"
        </item>
      </list>

      <!-- A link to the login() method in the Authentication interface -->
      <!-- A link to the Block struct -->
      <para>
        The <link interface="Authentication" method="login">login()</link> method
        begins a PAM conversation to authenticate as a user. It returns a list
        of <link struct="Block">Block</link> objects encapsulating the status of
        the conversation, the messages that should be displayed, and the input
        that should be collected.
      </para>

      <para>
        At each step, when the requested input has been collected, it is
        submitted using <link interface="Authentication"
       method="submit">submit()</link>.  This method also returns a list of
       <link struct="Block">Block</link> objects, allowing the conversation to
        continue indefinitely until authentication is complete.
      </para>

      <!-- A link to a struct field, and an enum value -->
      <para>
        When either of the two returns a <link struct="Block">Block</link> whose
       <link struct="Block" field="type">type</link> is <link enum="BlockType"
       value="SUCCESS">SUCCESS</link>, authentication has succeeded and <link
       interface="Authentication" method="complete">complete()</link> should be
        called to close the conversation.
      </para>

      <para>
        A typical algorithm for walking through this conversation might be:
      </para>

      <!-- A program listing, with a caption -->
      <example caption="Authentication interface" language="python">
import rad.connect as radcon
import rad.auth as rada
```

```
# Create a connection
rc=radcon.connect_tls("host")
# Get a native-looking python object that throws RAD exceptions
auth = rada.RadAuth(rc)
# login with username and password
auth.pam_login("jdoe", "*******")
print rc
rc.close()
print rc
</example>

     <para>
        This example uses the rad.auth module which makes simplifying
        assumptions for a default Solaris install.
     </para>
   </doc>

   <!-- User Identity -->
   <property name="user" type="string" access="ro" nullable="true" stability="private">
     <summary>
        gets the username of the connected user
     </summary>
   </property>

   <!-- PAM Authentication -->
   <method name="login" stability="private">
     <summary>
        begins a PAM conversation to authenticate as the specified user
     </summary>
     <result type="Block"/>
     <error/>
     <argument type="string" name="locale"/>
     <argument type="string" name="username"/>
   </method>

   <method name="submit" stability="private">
     <summary>
        continues a PAM conversation with information collected from the
        previous step
     </summary>
     <result type="Block"/>
     <error/>
     <argument name="responses">
       <list type="secret"/>
     </argument>
   </method>

   <method name="complete" stability="private">
     <summary>
        completes the PAM conversation with the RAD server
     </summary>
   </method>
 </interface>
</api>
```

radadrgen Processing Tool

radadrgen is the ADR IDL processing tool that generates API-specific language bindings for the RAD server and various client environments. See the radadrgen(1) man page for details about its options.

3

libadr Library

Data Management in libadr

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 3-1 ADR Base and Array Types

ADR type	C adr_type_t	C array adr_type_t
string	adr_t_string	adr_t_array_string
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_reference	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 mandatory immutable types, allocation implies initialization, and their allocation routines take the value of `adr_data_t` as arguments. 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 in `libadr`

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

`adr_data_new_string` 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, ...);
```

`adr_data_new_fstring` 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);
```

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

Allocating `boolean` in `libadr`

```
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 in `libadr`

```
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 in `libadr`

```
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 in `libadr`

```
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 in `libadr`

```
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. The client or server implementations might take additional steps to protect the content of the character array data, for example, zeroing buffers after use.

Allocating Names in `libadr`

```
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 the 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 in `libadr`

```
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 in `libadr`

```
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 in `libadr`

```
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. This is required in the case of structured types with non-nullable fields. 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 the reference is needed longer duration, the caller should take a hold on the `adr_data_t` (see adr_data_t Type). 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, 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.

- `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.

Manipulating the Structure of an `adr_data_t` Type

The primary interface for accessing the `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.

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. Another, less frequently used 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 3-1 Error Handling Without adr_data_verify

```
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, the entire non-truncated function can be reduced to the method shown in the following example.

Example 3-2 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 that structure fields can 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 3-3 Creating Names With `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 3-4 Creating Names With `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 an 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 in `libadr`

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.

Dictionary Support in `libadr`

The `libadr` functions that are supported for dictionary are as follows:

- `adr_data_t *adr_data_new_dictionary(adr_type_t *type)`
- `boolean_t adr_dictionary_contains(adr_data_t *dict, adr_data_t *key)`
- `adr_data_t *adr_dictionary_get(adr_data_t *dict, adr_data_t *key)`
- `adr_data_t *adr_dictionary_keys(adr_data_t *dict)`
- `int adr_dictionary_map(adr_data_t *dict,int (*func)(adr_data_t *, adr_data_t *, void *), void *data)`
- `adr_data_t *adr_dictionary_put(adr_data_t *dict, adr_data_t *key, adr_data_t *value)`
- `adr_data_t *adr_dictionary_remove(adr_data_t *dict, adr_data_t *key)`
- `unsigned int adr_dictionary_size(adr_data_t *dict)`
- `adr_data_t *adr_dictionary_values(adr_data_t *dict)`

For more information, see Dictionary Definitions in RAD Modules.

Example 3-5 Using `libadr` for Dictionary Operations

This example shows the dictionary operations by using the `libadr` functions. For information about defining dictionaries, see Defining a Dictionary for RAD.

```
/*Create a new dictionary with a key type of integer and a value type of string*/
adr_data_t *ex_dict = adr_data_new_dictionary(&t__dict_integer_string);
```

```
/*Put a key value pair [ 1 : "value1" ] in the empty dictionary*/
(void) adr_dictionary_put(ex_dict, adr_data_new_integer(1),
adr_data_new_string("value1"));


/*Get the value for the key (1)*/
adr_data_t *value = adr_dictionary_get(ex_dict, adr_data_new_integer(1));


/*Replace the value for the key (1)*/
adr_data_t *old_value = adr_dictionary_put(ex_dict, adr_data_new_integer(1),
adr_data_new_string("value2"));


/*Remove the value for the key (1)*/
adr_data_t *value = adr_dictionary_remove(ex_dict, adr_data_new_integer(1));
```

API Management in libadr

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

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

Running radadrgen

You can use radadrgen to generate server bindings for both C and Python languages. See the radadrgen(1) man page for details about its options.

Generating Server Bindings for C in libadr

This section provides examples for generating server bindings for C.

Example 3-6 Generating Server Bindings for C

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

The C library argument to radadrgen 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`.

When `radadrgen` is used in the Using Various RAD Module Elements, the result is two files. One, `api_example_impl.c`, holds the implementation of the `GrabBag` interface and data types it depends on, and should 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 3-7 Showing a `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 3-8 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.

Generating Server Bindings for Python in libadr

The Oracle Solaris 11.4 code that is generated is currently compatible with Python 2.7 and Python 3.5. However, Python updates might affect that compatibility, so you should specify the exact version of Python. Furthermore, do not add new Python 2.7 code. This version of Python is near the end of support by the community.

This section provides examples for generating server bindings for Python 3.5.

Example 3-9 Generating Server Bindings for Python 3.5

This example shows how to generate Python server bindings for the API Snake, which has the interface, Cobra, with one method, multiply.

```
<api xmlns="https://xmlns.oracle.com/radadr"
    name="com.oracle.solaris.rad.snake">

 <summary>
     Snake API
 </summary>

 <doc>
      <para>
        This is a testing module for Python Server Modules.
      </para>
</doc>

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

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

   <method name="multiply" stability="private">
   <doc>
     Multiply two numbers.
    </doc>
    <result type="integer"/>
    <argument name="first" type="integer"/>
    <argument name="second" type="integer"/>
</method>

</interface>

</api>
```

The following example shows how to generate Python server bindings for snake.adr in the build sub-directory.

```
$ radadrgen -l python35 -s server -d build snake.adr
```

When you run radadrgen in this example, the following files are generated:

- api.snake.h and api.snake_impl.c C files – These files are generated so that the Python module can reuse the existing RAD marshalling and dispatching framework.

- `com/oracle/solaris/rad/snake_iface.py` Python file – This file contains an abstract definition of an interface, which should be extended to provide a concrete implementation.

Example 3-10 Showing a `radadrgen`-Generated Interface From the Python File

```
"""
Snake API

This is a testing module for Python Server Modules.


"""
try:
    import modapi
except ImportError:
    import rad.server.modapidoc as modapi
from rad.server import RADInstance
from rad.client import *
import abc
import six

dom = "com.oracle.solaris.rad.snake"
vers = (1, 0)

@ClassStability("private")
class Cobra(six.with_metaclass(abc.ABCMeta, RADInstance)):
    """"""

    _rad_type = "Cobra"
    _rad_domain = dom
    _rad_version = vers
    _rad_singleton = None

    __metaclass__ = abc.ABCMeta

    def __init__(self, name = None, user = None, freef = None, dynamic = False):
        super(Cobra, self).__init__(_rad_moddata, name, user, freef, dynamic)

    @MethodStability("private")
    @abc.abstractmethod
    def multiply(self, first, second):
        """Multiply two numbers."""
        pass
```

The methods in the example are decorated to indicate that they are abstract and to indicate their RAD stability level. You can write the Python code to implement the abstract interface.

Example 3-11 Implementing Interfaces Generated by `radadrgen`

```
from . import snake_iface
import rad.client as radcli
import rad.server as radser

class Cobra(snake_iface.Cobra):

    def __init__(self, name = None, user = None, freef = None):
        super(Cobra, self).__init__(name, user, freef)

    def multiply(self, first, second):
        return (first * second)
```

```
def rad_init():
    radser.rad_log(radser.rad_log_lvl.RL_DEBUG,
        "Initializing: %s" % "com.oracle.solaris.rad.snake")
    Cobra._rad_insert_singleton(radser.rad_container)
    return 0

def rad_fini(handle):
    radser.rad_log(radser.rad_log_lvl.RL_DEBUG,
        "Finalizing: %s" % "com.oracle.solaris.rad.snake")
    return 0
```

In this example, the `Snake` module is implemented and is exporting the `rad_init` and `rad_fini` functions, which provides the capability for the module to setup or remove an execution environment. In this example, you are logging and creating a singleton instance to represent the `Snake` module.

The RAD daemon converts the ADR (RAD's native data representation format) types to or from the Python types. For basic types, the conversion is straightforward mapping. For derived types, the conversion is more complex, but essentially follows the same process as illustrated above for interfaces. `radadrgen` generates the required types in the binding. The various generated components are identical to any such components in the Python client binding.

4
RAD Module Development

C APIs for RAD

This section describes the APIs that are available for C language.

Entry Points in C for RAD

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 points return a `conerr_t`. If the access is successful, they should return `CE_OK`.

If the access fails due to a system error or a module defined error, they should return the respective error codes. For more information about the error codes, see Error Codes in C for RAD.

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.

Error Codes in C for RAD

RAD distinguishes errors as system errors and module defined errors.

System Errors in C for RAD

If the access fails due to a system error, the entry points should return one of the following error codes:

- `CE_SYSTEM` – An operation fails due to a system error. This error code should not have payload.
- `CE_NOTFOUND` – The retrieve operation fails because the object does not exist. This error code should not have payload.
- `CE_EXISTS` – The create operation fails because the object already exists. This error code should not have payload.
- `CE_PRIV` – An operation fails due to insufficient privileges. This error code should not have payload.
- `CE_NOMEM` – An operation fails due to insufficient memory. This error code should not have payload.

RAD Module Defined Errors in C for RAD

If the access fails due to an expected error as described in the API definition, the entry points should return `CE_OBJECT`. If an expected error occurs and an error payload is defined, it may be set in *error.

> **Note:**
>
> Do not use the `CE_MISMATCH` and the `CE_ILLEGAL` error codes. If there is any data type mismatch error or an illegal access error, return the `CE_OBJECT` error code with a payload describing the illegal arguments.

Global Variables in C for RAD

The following are the RAD global variables in the C language:

`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 in C for RAD

The following are the RAD module registration functions in the C language:

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 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 in C for RAD

The following are the RAD instance management functions in the C language:

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 in C for RAD

The following are the RAD container actions in the C language:

conerr_t rad_cont_insert(rad_container_t *container, adr_name_t *name, rad_instance_t *instance);
conerr_t rad_cont_insert_singleton(rad_container_t *container, adr_name_t *name, rad_object_t *object);
Creates an instance, rad_instance_t, using the supplied name and object and then inserts it into container. If the operation succeeds, CE_OK is returned.

void rad_cont_remove(rad_container_t *container, adr_name_t *name);
Removes the instance from the container.

```
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);
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);
```

Registers a dynamic container instance manager. This is 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.

Logging in C for RAD

The following are the RAD logging functions in the C language:

`void rad_log(rad_logtype_t type, const char * format, ...);`
Logs 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`
Logs a memory allocation failure with log level `RL_FATAL`.

`rad_logtype_t rad_get_loglevel`
Returns the logging level.

Using Threads in C for RAD

The following are thread functions in the C language for RAD:

`void *rad_thread_arg(rad_thread_t *tp);`
Returns the `arg` referenced by the thread `tp`.

`void rad_thread_ack(rad_thread_t *tp, rad_moderr_t error);`
Acknowledges the thread referenced by `tp`. This process enables the controlling thread, from which a new thread was 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.
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.

`rad_moderr_t rad_thread_create(rad_threadfp_t fp, void *arg);`
Creates 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);`
Creates a thread to run `fp`. `arg` is stored and passed into `fp`.

Synchronization in C for RAD

The following are synchronization functions in the C language for RAD:

`void rad_mutex_init(pthread_mutex_t *mutex);`
Initializes a `mutex.abort` on failure.

`void rad_mutex_enter(pthread_mutex_t *mutex);`
Locks a `mutex.abort` on failure.

`void rad_mutex_exit(pthread_mutex_t *mutex);`
Unlocks a `mutex.abort` on failure.

`void rad_cond_init(pthread_cond_t *cond);`
Initializes a condition variable, `cond.abort`, on failure.

Subprocesses in C for RAD

The following are subprocesses in the C language for RAD:

`exec_params_t *rad_exec_params_alloc`
Allocates a control structure for executing a subprocess.

`void rad_exec_params_free(exec_params_t *params);`
Frees a subprocess control structure, params.

`void rad_exec_params_set_cwd(exec_params_t *params, const char *cwd);`
Sets the current working directory, `cwd`, in a subprocess control structure, `params`.

`void rad_exec_params_set_env(exec_params_t *params, const char **envp);`
Sets the environment, `envp`, in a subprocess control structure, `params`.

`void rad_exec_params_set_loglevel(exec_params_t *params, rad_logtype_t loglevel);`
Sets the RAD log level, `loglevel`, in a subprocess control structure, `params`.

`int rad_exec_params_set_stdin(exec_params_t *params, int fd);`
Sets the stdin file descriptor, `fd`, in a subprocess control structure, `params`.

`int rad_exec_params_set_stdout(exec_params_t *params,int fd);`
Sets the stdout file descriptor, `fd`, in a subprocess control structure, `params`.

`int rad_exec_params_set_stderr(exec_params_t *params, int fd);`
Sets 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);`
Uses 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);`
Uses 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);`
Uses 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 in C for RAD

The following are the RAD utilities in the C language:

`void *rad_zalloc(size_t size);`
Returns a pointer to a zero-allocated block of size bytes.

`char *rad_strndup(char *string, size_t length);`
Creates and returns a duplicate of string that is of size `length` bytes.

`int rad_strccmp(const char * zstring, const char * cstring, size_t length);`
Compares two strings, up to a maximum size of `length` bytes.

`int rad_openf(const char *format, int oflag, mode_tmode, ...);`
Opens a file with access mode `oflag`, and mode `mode`, whose path is specified by calling `sprintf` on `format`.

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

Locales in C for RAD

The following are the RAD locale functions in the C language:

`int rad_locale_parse(const char *locale, rad_locale_t **rad_locale);`
Updates `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);`
Frees a locale, `rad_locale`, previously obtained with `rad_locale_parse`.

Transactional Processing in C for RAD

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

Asynchronous Methods and Progress Reporting in C for RAD

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.

For 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 ask the server for a progress report.

Python APIs for RAD

This section describes the APIs that are available for the Python language.

`rad.server`
RAD Server module.

`RADContainer`
RADContainer Container base class. Represents a container into which instances are inserted.

`RADInstance`
Instance base class. All the generated interfaces inherit from the `RADInstance` class. Thus, the interfaces inherit a set of useful behaviours. All the inherited attributes are prepended with `_rad` to both prevent name collisions and clearly indicate that these attributes are protected.

`RADException`
RAD exception base class. Represents an exception, which will be propagated back to the client as a `CE_OBJECT` exception. If an invocation fails, the error is declared in the ADR. See Using the RADException Python Class.

`RADExistsException`
Exception when an object already exists.

`RADIllegalException`
Exception when an illegal object is provided to the client.

`RADNotFoundException`
Exception when an object is not found.

The following functions must be provided by an implementation module:

- `rad_reg`
- `rad_init`
- `rad_fini`

`rad.server` Python Module

The `rad.server` Python module includes functions and attributes. The following are `rad.server` functions.

`rad_log`
Provides log information.

`_rad_create`
Creates `rad` session

`_rad_delete`
Deletes a `rad` session.

`rad_locale_get`
Provides the locale information.

`rad_locale_free`
Frees the locale.

`rad_locale_parse`
Parses the locale.

`rad_instance_rele`
Frees the instance.

`rad_instance_hold`
Holds the instance.

The following are `rad.server` attributes.

`rad_container`
Variable pointing to the RAD container that the module must be using.

`rad_log_lvl`
One of `RL_DEBUG`, `RL_NOTE`, `RL_WARN`, `RL_ERROR`, `RL_CONFIG`, `RL_FATAL`, `RL_PANIC`.

`RADInstance` Python Class

`RADInstance` is an encapsulation of a RAD instance in the server module. `RADInstance` includes one method and one property:

- `_rad_notify(self, event, payload)` – This `RADInstance` method sends an event *event* with payload *payload* to subscribed clients
- `_rad_name` – This `RADInstance` property is the RAD name of the given instance

`RADContainer` Python Class

`RADContainer` contains live RAD instances. `RADContainer` includes the following methods:

`insert(self, inst)`
Adds instance into the container. This API is rarely used directly and typically called by *Subclass-of-RADInstance* `.__init__(self, name, user, freef, dynamic)` when *name* is not *None* and *dynamic* == *False*.

`insert_singleton(self, radinstsubcls, name)`
Creates a new `RADInstance` subclass instance and insert it to container under name *name*.

`remove(self, inst)`
Removes instance *inst* from the container.

`register(self, klass, listf, lookupf, user)`
Registers `RADInstance` subclass *klass* for dynamic listing and looking up.

`find_instance(self, name)`
Finds instance by name *name*.

`list(self, pat)`
Gets a list of instance RAD names matching pattern *pat*.

`RADException` Python Class

This following example shows how to use `RADException`.

Example 4-1 Using the `RADException` Python Class

The ADR type definition is as follows:

```
<struct name="pair" stability="private">
   <field name="first" type="integer"/>
   <field name="second" type="integer"/>
</struct>
```

The ADR method definition is as follows:

```
<method name="raiseError" stability="private">
  <doc>
  Raise an exception to test exception handling.
  </doc>
  <error type="pair"/>
</method>
```

In the method definition, you are specifying that an exception must be raised when the initialization of the struct `pair` fails.

The implementation is as follows:

```
def raiseError(self):
    raise radser.RADException(snake_iface.pair(3, 6))
```

In this example, you are raising a `RADException` and providing a payload that matches the definition in the ADR document.

The exceptions that occur as a consequence of "other" errors such as divide by zero are propagated back to the client as a CE_SYSTEM error representing the general RAD failure code for systemic failure.

RAD Namespace Objects

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.

RAD Static Objects

The `rad_modapi.h` header file 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`. For example:

```
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 Dynamic Handlers

A module can register a dynamic handler for each interface that is implemented by the module. This registration allows efficient searching within a module by limiting a listing to a matching subset of the instances that the module is managing. Note that you can register a single dynamic handler for a module's entire namespace. Additionally, when you register a dynamic handler, you need to specify a lookup function pointer.

The following example shows the use of dynamic handlers in the zones module.

Example 4-2 Using Dynamic Handlers in the Zones Module

```
cerr = rad_cont_register_dynamic(rad_container, aname,
&modinfo, zone_listf, zone_lookupf, NULL);
```

RAD Module Linkage

Modules are registered with the RAD daemon in the `_rad_reg` function. Registration is automatically generated from the information contained within the IDL that defines 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` function in the module is called by the RAD daemon just prior to unloading the module.

Example 4-3 Initializing and Registering a RAD Module

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

int
_rad_init(void)
{
    adr_name _t *uname = adr_name_vcreate("com.example.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);
}
```

A

zonemgr ADR Interface Description Language Example

The example in this appendix shows some APIs used in the zonemgr ADR Interface Description Language. It does not reflect the actual full implementation of the zonemgr APIs in Oracle Solaris.

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

<api xmlns="https://xmlns.oracle.com/radadr"
  name="com.oracle.solaris.rad.zonemgr"
  description="API for Zones administration">

  <summary>
    API for Zones administration
  </summary>

  <doc>
    <para>
      This API provides functionality for the configuration and
      administration of Zones subsystem.
    </para>
  </doc>

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

  <enum name="ErrorCode">
    <summary>Errors</summary>
    <value name="NONE" value="0">
      <summary>No error</summary>
    </value>
    <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">
    <summary>An error occurred for the given operation</summary>
    <doc>
      <example language="python" caption="Retrieve an error information from the
structure.">
        ...
        try:
```

```
            test0.cancelConfig()
        except rad.client.ObjectError as e:
            result = e.get_payload()
            print("Result.code = %s" % result.code)
      </example>
    </doc>
    <field type="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">
    <summary>The payload of a configChange event</summary>
    <field type="string" name="zone"/>
  </struct>

  <struct name="StateChange">
    <summary>The payload of a stateChange event</summary>
    <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" type="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">
    <summary>A zone resource</summary>
    <doc>
      <para> This structure is used for storing information about an individual
        zone configuration resource.
      </para>
    </doc>
    <field type="string" name="type"/>
    <field name="properties" nullable="true">
      <list type="Property"/>
    </field>
    <field name="parent" type="string" nullable="true"/>
  </struct>

  <interface name="ZoneManager">
    <summary>Manage zones on this system</summary>
    <doc>
      Create and delete zones. Changes in the state of zones
      can be monitored through the StateChange event.
```

```
  </doc>

  <method name="create">
    <summary>Create a zone</summary>
    <result type="Result"/>
    <error type="Result"/>
    <argument name="name" type="string"/>
    <argument name="path" type="string" nullable="true"/>
    <argument name="template" type="string" nullable="true"/>
  </method>

  <method name="delete">
    <summary>Delete a zone</summary>
    <result type="Result"/>
    <error type="Result"/>
    <argument name="name" type="string"/>
  </method>

  <method name="importConfig">
    <summary>Import a zone</summary>
    <result type="Result"/>
    <error type="Result"/>
    <argument name="noexecute" type="boolean"/>
    <argument name="name" type="string"/>
    <argument name="configuration">
      <list type="string"/>
    </argument>
  </method>

  <event type="StateChange" name="stateChange"/>
</interface>

<interface name="ZoneInfo">
  <summary>Report on the zone in which this instance is executing</summary>
  <doc>
    Information about the current zone can be accessed.
  </doc>
  <property name="brand" access="ro" type="string"/>
  <property name="id" access="ro" type="integer"/>
  <property name="uuid" access="ro" type="string" nullable="true">
    <error type="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"/>
  <summary>Operations that affect a single zone</summary>
  <doc>
    Represents an individual zone. All zone configuration and
    administrative actions are represented in this interface.
    Changes of the zone configuration can be monitored through the
    configChange event.
  </doc>

  <property name="auxstate" access="ro" nullable="true">
    <list type="string"/>
    <error type="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 type="Result"/>
</property>
<property name="name" access="ro" type="string"/>
<property name="state" access="ro" type="string"/>
<method name="cancelConfig">
  <error type="Result"/>
</method>
<method name="exportConfig">
  <result type="string"/>
  <error type="Result"/>
  <argument name="includeEdits" type="boolean" nullable="true"/>
  <argument type="boolean" name="liveMode" nullable="true"/>
</method>
<method name="update">
  <error type="Result"/>
  <argument name="noexecute" type="boolean"/>
  <argument name="commands">
    <list type="string"/>
  </argument>
</method>
<method name="editConfig">
  <error type="Result"/>
  <argument type="boolean" name="liveMode" nullable="true"/>
</method>
<method name="commitConfig">
  <error type="Result"/>
</method>
<method name="configIsLive">
  <result type="boolean"/>
</method>
<method name="configIsStale">
  <result type="boolean"/>
  <error type="Result"/>
</method>
<method name="addResource">
  <error type="Result"/>
  <argument name="resource" type="Resource"/>
  <argument name="scope" type="Resource" nullable="true"/>
</method>
<method name="reloadConfig">
  <error type="Result"/>
  <argument type="boolean" name="liveMode" nullable="true"/>
</method>
<method name="removeResources">
  <error type="Result"/>
  <argument name="filter" type="Resource" nullable="false"/>
  <argument name="scope" type="Resource" nullable="true"/>
</method>
<method name="getResources">
  <result>
    <list type="Resource"/>
  </result>
  <error type="Result"/>
  <argument name="filter" type="Resource" nullable="true"/>
  <argument name="scope" type="Resource" nullable="true"/>
</method>
<method name="getResourceProperties">
  <result>
```

```
    <list type="Property"/>
  </result>
  <error type="Result"/>
  <argument name="filter" type="Resource" nullable="false"/>
  <argument name="properties" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="setResourceProperties">
  <error type="Result"/>
  <argument name="filter" type="Resource" nullable="false"/>
  <argument name="properties" nullable="false">
    <list type="Property"/>
  </argument>
</method>
<method name="clearResourceProperties">
  <error type="Result"/>
  <argument name="filter" type="Resource" nullable="false"/>
  <argument name="properties" nullable="false">
    <list type="string"/>
  </argument>
</method>
<method name="apply">
  <result type="Result"/>
  <error type="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="attach">
  <result type="Result"/>
  <error type="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="boot">
  <result type="Result"/>
  <error type="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="clone">
  <result type="Result"/>
  <error type="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="detach">
  <result type="Result"/>
  <error type="Result"/>
  <argument name="options" nullable="true">
    <list type="string"/>
  </argument>
</method>
<method name="halt">
  <result type="Result"/>
  <error type="Result"/>
  <argument name="options" nullable="true">
```

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

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

Index

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

V