Netra™ ft 1800 CMS API Developer's Guide

THE NETWORK IS THE COMPUTER™

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900 USA
650 960-1300 Fax 650 969-9131

Part No. 805-5870-10
February 1999, Revision A

Send comments about this document to: `docfeedback@sun.com`

Copyright 1999 Sun Microsystems, Inc., 901 San Antonio Road • Palo Alto, CA 94303 USA. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any. Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Parts of the product may be derived from Berkeley BSD systems, licensed from the University of California. UNIX is a registered trademark in the U.S. and other countries, exclusively licensed through X/Open Company, Ltd.

Sun, Sun Microsystems, the Sun logo, AnswerBook, Java, the Java Coffee Cup, and Solaris are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. Registered Excellence (and Design) is a certification mark of Bellcore.

The OPEN LOOK and Sun™ Graphical User Interface was developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 1999 Sun Microsystems, Inc., 901 San Antonio Road • Palo Alto, CA 94303 Etats-Unis. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Des parties de ce produit pourront être dérivées des systèmes Berkeley BSD licenciés par l'Université de Californie. UNIX est une marque déposée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company, Ltd.

Sun, Sun Microsystems, le logo Sun, AnswerBook, Java, le logo Jave Coffee Cup, et Solaris sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

L'interface d'utilisation graphique OPEN LOOK et Sun™ a été développée par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant également les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" ET AUCUNE GARANTIE, EXPRESSE OU IMPLICITE, N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Contents

1. The CMS Application Programming Interface 1

The Purpose of the CMS API 1

What the CMS API Provides 2

Use of the CMS API within Network Management Environments 3

Use of CMS API within the TMN Environment 3

2. The CMS API Architecture 5

The CMS Components 5

Object Types 5

Functionality 6

How the CMS Models the Platform 6

Object Definition Files 6

How the CMS API Communicates with the CMS 7

3. The CMS Server Object Model 11

Definition of a Managed Object 11

Attributes 11

Driver States 12

Behavior 15

Relationships 15

Constraints 16

4. The CMS API Object Model 17

CMS API Object Classes 20

Object 20

`session` Object 21

`managed` Object 21

`system` Object 21

`module` Object 22

`attribute` Object 22

`scope and filter` Object 23

`notification` Object 23

`event forwarding discriminator 23`

`exception forwarding discriminator 23`

`event` Object 24

`exception` Object 24

`notification record` Object 24

`event record` Object 25

`exception record` Object 25

`error` Object 26

CMS API Commands 26

`cms_open 26`

`cms_close 26`

`cms_request_notifications 27`

`cms_cancel_notifications 27`

`cms_get 27`

`cms_set 28`

5. Introduction to CMS API Functionality 29

Opening a CMS API Session 29

Registering Event and Exception Handlers 29

Obtaining the Class Names 30

Obtaining the Instance Identifiers of Modules of Interest 30

Obtaining the Attributes of a Module 31

Obtaining Attribute Values 31

Accessing the Definition of a Module 32

Setting the Value of an Attribute 32

Closing the Session 33

6. Connecting to and Disconnecting from the CMS 35

Connecting to the CMS 35

`session` Object 36

Disconnecting from the CMS 37

`session` Object 37

7. Creating a Model of the CMS Module Classes 39

Obtaining a List of the Platform Model Object Classes 39

`session` Object 39

`system` Object 39

`scope and filter` Object 40

Obtaining a List of Platform Model Object Classes with a Common Attribute 41

`scope and filter` Object 42

Obtaining a List of the Platform Model Object Instances 44

`session` Object 44

`system` Object 44

`scope and filter` Object 44

Obtaining a List of Platform Model Object Instances with a Common Attribute Value 47

`attribute` Object 47

`scope and filter` Object 47

Obtaining a List of Platform Model Object Instances that are Faulty 49

`scope and filter` Object 50

Obtaining Class Information About a Platform Object 51

`session` Object 51

`module` Object 52

`scope and filter` Object 52

`attribute` Object 52

Obtaining Instance Information About a Platform Object 55

`session` Object 55

`module` Object 55

`scope and filter` Object 56

`attribute` Object 56

Setting Attribute Values of a Platform Object Instance 57

`session` Object 57

`module` Object 57

`attribute` Object 57

`scope and filter` Object 58

Glossary 61

Figures

FIGURE 1-1 The TMN Layered Model 3

FIGURE 2-1 CMS Architecture 9

FIGURE 3-1 A State Diagram for an Abstract Device Object. 13

FIGURE 3-2 Event Trace for an Abstract Device Object. 14

FIGURE 4-1 The CMS API Object Model 19

Code Examples

CODE EXAMPLE 6-1 Opening a CMS API Session 36

CODE EXAMPLE 6-2 Closing a CMS API Session 37

CODE EXAMPLE 7-1 Returning the CMS Object Classes 40

CODE EXAMPLE 7-2 Returning the CMS Object Classes with a Common Attribute 42

CODE EXAMPLE 7-3 Returning a List of Object Instances of the Platform Model 45

CODE EXAMPLE 7-4 Returning the Instance Names with a Common Attribute 47

CODE EXAMPLE 7-5 Obtaining the Faulty Object Instances 50

CODE EXAMPLE 7-6 Obtaining Class Information 52

CODE EXAMPLE 7-7 Obtaining Instance Information 56

CODE EXAMPLE 7-8 Setting Attribute Values 58

Preface

This guide is intended for software engineers to enable them to develop a programming interface for applications that are required to interface to the configuration and fault tolerant services of the Netra ft platform.

How This Book Is Organized

The guide contains the following chapters:

Chapter 1 provides an overview of the features and purpose of the CMS API.

Chapter 2 describes the CMS API architecture.

Chapter 3 describes the CMS server object model.

Chapter 4 describes the CMS API object model.

Chapter 5 provides an introduction to how the CMS API functions.

Chapter 6 illustrates how to connect and disconnect from the CMS.

Chapter 7 describes how to model the CMS server platform.

Glossary contains a list of words and phrases used in the guide, and their definition.

Typographic Conventions

TABLE P-1 Typographic Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`** `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide.* These are called *class* options. You *must* be superuser to do this.
	Command-line variable; replace with a real name or value	To delete a file, type `rm` *filename*.

Shell Prompts

TABLE P-2 Shell Prompts

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

Related Documentation

TABLE P-3 Related Documentation

Application	Title	Part Number
Software Reference	*Netra ft 1800 Reference Manual*	805-4532-10
Driver Development	*Netra ft 1800 Developer's Guide*	805-4530-10
CMS Development	*Netra ft 1800 CMS Developer's Guide*	805-7899-10

Sun Documentation on the Web

The `docs.sun.com`sm web site enables you to access Sun technical documentation on the Web. You can browse the `docs.sun.com` archive or search for a specific book title or subject at:

`http://docs.sun.com`

Sun Welcomes Your Comments

We are interested in improving our documentation and welcome your comments and suggestions. You can email your comments to us at:

`docfeedback@sun.com`

Please include the part number of your document in the subject line of your email.

CHAPTER 1

The CMS Application Programming Interface

The Configuration Management System (CMS) is an essential component of the Netra™ ft 1800 fault tolerant platform. It provides configuration management for a telecommunications environment, addressing the needs for remote management and alarms.

The Application Programming Interface (API) to the Netra ft CMS enables the software developer to implement applications that interface to the configuration and fault management services of the fault tolerant platform.

The Purpose of the CMS API

The CMS API enables vendors to provide additional system management policies beyond those provided by the CMS. These policies can require the generation of statistical information on the health of the platform, or the generation of external event or alarm notifications based on certain CMS events. The CMS API also enables vendors to provide their own user interface for the system management of the platform.

The CMS API provides the software developer with the services required to integrate the Netra ft into network management systems using Telecommunication Managed Networks (TMN) or communications protocols. The software developer can also implement additional applications for local control of the configuration and fault management services of the Netra ft.

Currently, most computer systems used within telecommunications are replicated to provide redundancy, and the fault tolerance is managed by the software. The Netra ft 1800 manages its own fault tolerance using the CMS. To external management

entities, it appears as a virtual, highly-reliable machine. The CMS API enables management entities to manage the internal fault tolerance of the CMS in the same way as they can manage redundant systems.

The CMS API can be used directly to connect to network managers or agents written to provide CMIP, SNMP or other protocols.

What the CMS API Provides

The CMS API provides a basic set of library routines for:

- monitoring all or specific aspects of the platform's physical configuration and fault tolerance
- managing the platform's physical configuration and fault tolerance

The CMS API provides an event notification mechanism, which enables the developer to write applications that can gather statistics on the health of a platform. The CMS API also enables the developer to manipulate the model of the platform, which influences the behavior of the machine and enables vendor-specific policies to be implemented on top of the platform's generic behavior. The CMS API exports its model of the platform using generic constructs, enabling external utilities to create their own model of the machine.

The CMS API provides author access to configuration and fault management of the Netra ft system. The CMS API has been used in conjunction with the user commands `cmsfix`(1M), `xcmsfix`(1M), `cmsconfig`(1M) and `cmsfruinfo`(1M) to provide the local interfaces to the CMS. The developer can add to the base functionality of the Netra ft system by providing services to manage the system remotely.

The CMS API enables the application developer to:

- Traverse the module hierarchy of the Netra ft system
- Obtain the attribute definitions of Netra ft modules
- Interrogate the value of Netra ft module attributes
- Change the value of Netra ft module attributes
- Optimize the CMS API according to the application needs
- Register event and exception handlers

Use of the CMS API within Network Management Environments

The CMS API can be used to connect to network managers, or agents written to provide CMIP, SNMP or other protocols. It is not essential to export the complete CMS model of a platform to the network manger. It *is* important to define the level of management or monitoring required, and the capabilities of the standard used. From this, it is possible to define a simplified CMS model of the platform that can be mapped by the CMS API, application or agent to the particular management object model defined by the required standard.

Use of CMS API within the TMN Environment

As telecommunications companies look to network management systems to hide from the operator the complexity and inconsistency of the different management domains, they need to be able to move from today's vendor-proprietary world to a standard ISO/ITU-T environment. TMN is an ITUI-T standard that specifies world-wide management of telecommunications networks. It defines the functional areas of management as well as the interfaces between different parts of the network.

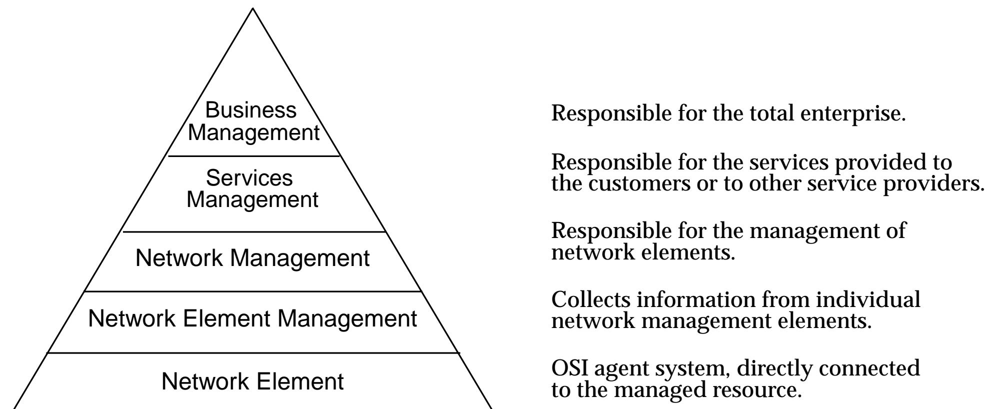


FIGURE 1-1 The TMN Layered Model

The network element layer consists of agents that communicate between the TMN management layers and the individual non-TMN network elements. The agent performs management operations on, and forwards notifications from, the managed objects that represent the physical network elements.

With an appropriate agent, the Netra ft system can be managed as a network agent in addition to providing its own internal configuration and fault management.

CHAPTER 2

The CMS API Architecture

The CMS Components

The CMS has the following components:

CMS server. The CMS server is responsible for modeling and interacting with the platform.

CMS API server. The CMS API server manages the per-client instances of the CMS API. It manages the client access to and control of information relating to the CMS platform model. Client notifications are managed by the CMS API server.

CMS API instance for each client. The CMS API provides a per-client instance for connection with the CMS.

Object Types

The CMS API defines the following object types as tools:

- `Session`
- `System`
- `Module`
- `Attribute`
- `Scope and filter`
- `Event` and `exception`
- `Event record` and `exception record`
- `Error`

Functionality

The CMS API functionality is provided through the methods performed by the objects listed above and through the following commands:

- `cms_open`
- `cms_close`
- `cms_get`
- `cms_set`
- `cms_request_notifications`
- `cms_cancel_notifications`
- `cms_has_event_occurred`

How the CMS Models the Platform

The CMS manages a platform by creating a model of it in terms of objects. The objects serve a dual purpose:

- To describe the platform to a user
- To provide a mechanism by which the CMS can manipulate the platform it is managing

An object is an abstraction of a physical component of the platform, or an abstraction of a component of functionality provided by the platform.

Object Definition Files

The CMS creates the model of the platform by using a blueprint object model which is prescribed by the CMS object definition files. These files define the structure of the objects on the platform – their identity, their relationship to other objects, their attributes and their behavior.

The CMS object definition files define:

- `Object` **class.**

  The `Object` class is a group of objects with similar properties (attributes), common behavior and common relationships to other objects.

  The CMS objects in the model created from the CMS object definition files are called object instances. An object instance is an actual realization of an object class.

- **Attributes** of the `Object` class.

  An *attribute* is a data value held by the objects in a class. An attribute is a pure data value, not an object. Each attribute name is unique within a class and all instances of objects of that class have an attribute of that name. The range of attribute values for instances in that class is defined by the class. However, each instance within that class can have a different value for that attribute.

- **Relationships** or **constituents** of the `Object` class.

  A *relationship* is a physical or conceptual connection between object instances. Relationships can only be one-to-one and can only be defined in one direction. The relationship is described by an attribute. An object class can specify the name of the relationship and the object instances that can be linked by that relationship.

The CMS object definition files also define the conditions that determine the state of an object of that class.

- A *condition* is a Boolean function of object values.
- A *state* is an abstraction of the attribute values and links of an object.

The state is defined only in terms of those attributes that affect the behavior of the object. The state specifies the response of the object to input events. A change of state caused by an event is called a *state transition*. The CMS object definition files define the object behavior in response to events that cause state transitions. The response of the object is likely to cause the CMS to interact with the entity it is modeling.

How the CMS API Communicates with the CMS

The CMS API is designed to be independent of the CMS platform model. To achieve this, the CMS API provides an interface that enables the platform meta-model to be read; that is:

- The object classes to be identified and their attribute and relationship definitions to be retrieved
- Object instances to be identified and their attribute values and relationships to be retrieved or set

Object class and object instance attributes and relationships are retrieved by passing a CMS API object, representing the chosen platform object target, into the `cms_get` function. After execution of the function, the CMS API object contains the requested information relating to the chosen target.

Alternatively, object instance attribute values are set by passing a CMS API object representing the chosen platform object target, and containing the attribute or relationship to be changed, into the `cms_set` function. After execution of the function, the CMS target takes on the attribute value of the CMS API object.

The CMS server contains representations of the physical components, such as devices or FRUs, within the machine and the higher-level objects which represent components of physical devices. These representations are known as *managed objects*. The CMS server creates the managed objects from system definition files. The CMS server interfaces to the system software that controls and monitors the physical components.

The CMS Server uses a state transition model paradigm. The managed objects are used to issue simple commands to the system software of their corresponding physical component, and responses and events from the system software are fed back to the managed object. The managed object will then change its state to represent the event. Other managed objects can have states which are dependent on the changed managed object; their states will also change.

The CMS API server manages the per-client instances of the CMS API. It manages the client access to the managed objects and the control of information relating to them. Client event and exception notifications are managed by the CMS API server.

The CMS API provides a per-client instance for management of the CMS. The CMS API provides tools for the client to access the managed objects, set event or exception forwarding discriminators and access information in an event or exception notification.

FIGURE 2-1 on page 9 provides an overview of the CMS architecture.

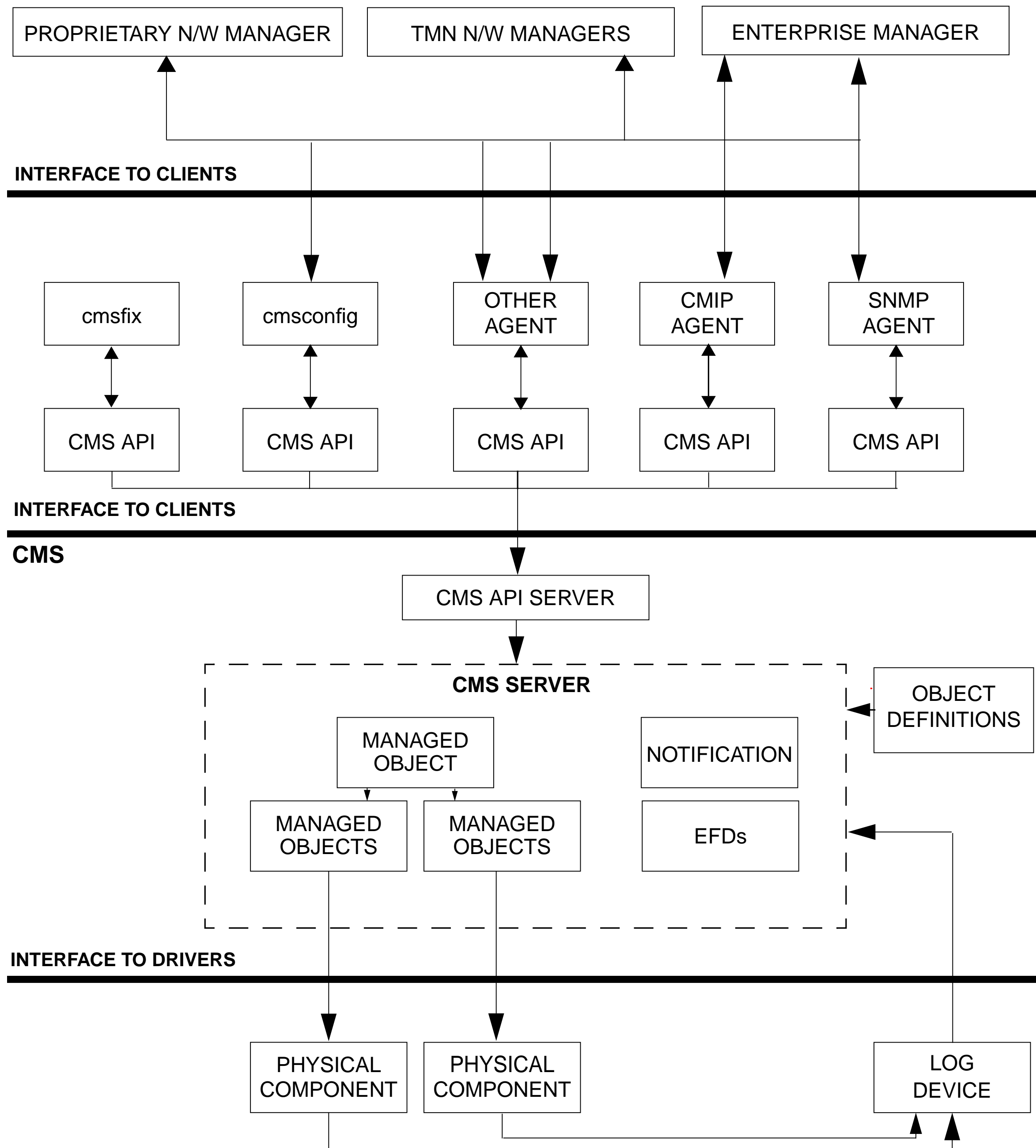


FIGURE 2-1 CMS Architecture

CHAPTER 3

The CMS Server Object Model

The object model describes the managed object classes known to the CMS server and their relationship to each another. There is no concept of a class hierarchy within the CMS. All module instance identifiers are unique and modules are not referenced by their containment relationships.

Definition of a Managed Object

Attributes

A managed object has attributes. The attributes can be:

- **System attributes** whose value can be set by a `system` object (for example, a device driver or a system daemon).

  System attributes include:

  - `_driver_state`
    The `_driver_state` is a property which is set by the driver of the managed object. The attribute is a driver state as described in “Driver States” on page 12.
  - `state`
    The attribute values and links held by the managed object are called its state. The state is derived from the condition attribute and other system and user attributes such as `_action`.

- **User attributes** whose value can be set by a non-system object (for example, an application such as `cmsfix`(1M) or `cmsconfig`(1M)).

  User attributes include:

  - `_action`
    The `_action` attribute indicates to the CMS the condition that the user requires the managed object to have.
  - `location`
    The `location` attribute specifies the physical location in which the managed object resides. The attribute is relevant to physical objects only.
  - `fault_acknowledged`
    The `fault_acknowledged` attribute signifies that the user has acknowledged a fault condition associated with the managed object. The local utilities `cmsfix`(1M) and `cmsconfig`(1M) provide mechanisms by which the user can acknowledge faults.
  - `description`
    The `description` attribute provides a pre-defined text description of the managed object class.
  - `user_label`
    The `user_label` attribute is used by the system administrator to specify a meaningful name, using free text, for a managed object instance. For example, this attribute can be used to relate a volume to a disk.

Driver States

The `_driver_state` attribute of the device object represents the state in which the device drivers believe their devices to be. The states are reported through the CMS device drivers interface and the device object's condition attribute is changed accordingly. The driver states of an abstract device object are shown below.

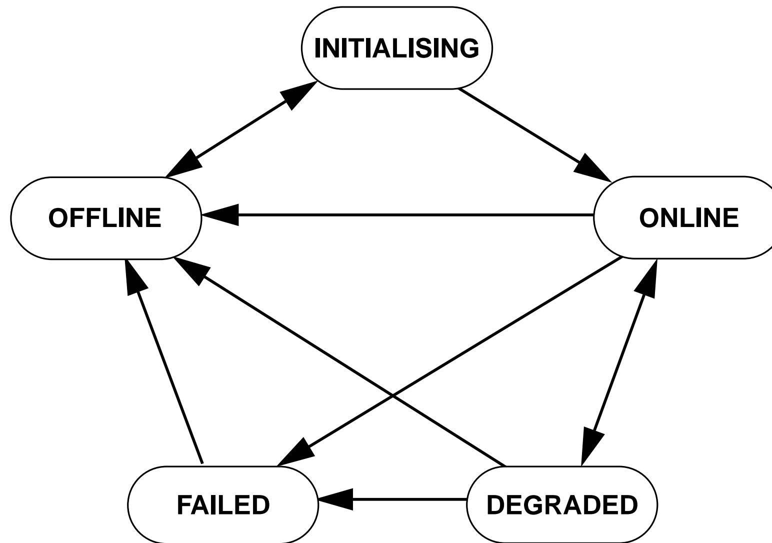


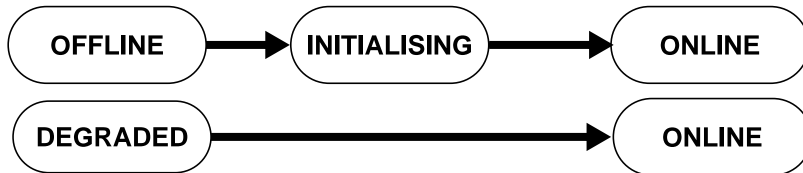
FIGURE 3-1 A State Diagram for an Abstract Device Object.

An abstract CMS device object has five core states:

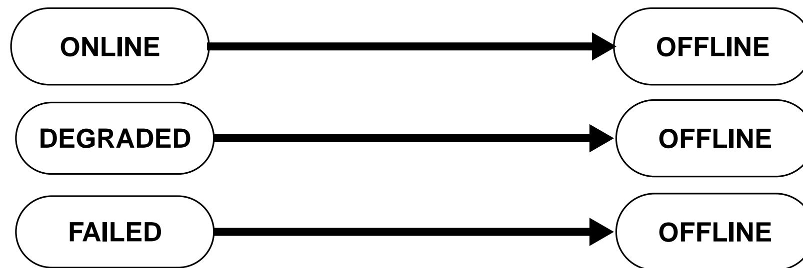
- `offline`
  The device object is not usable to the system.
- `initialising`
  The device object is attempting to come online (it is not usable to the system).
- `online`
  The device object is usable to the system.
- `degraded`
  The device object has a fault but is still usable to the system, possibly in a degraded mode of operation.
- `failed`
  The device object has a fault and is not usable to the system.

An event trace for typical scenarios for the state changes of an abstract device object are given in FIGURE 3-2 on page 14.

Scenario for typical user-controlled driver state changes



Scenario for turning off a device object



Scenario for typical device driver-controlled driver state changes

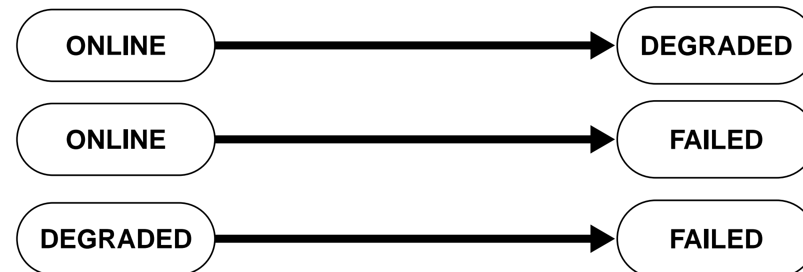


FIGURE 3-2 Event Trace for an Abstract Device Object.

Behavior

The behavior of a managed object is defined as a set of responses in the event of a derived state change or a specific attribute change. The responses are generated using a rule-based decision system within the CMS server. The responses are commands which are executable by a shell.

Responses are used to:

- Power on a module
- Enable access to a module
- Record the power on event in a module's EEPROM
- Power down a module
- Record information about module failures

Relationships

Relationships exist between managed objects, the nature of which depend on the objects being modeled. The types of relationship that can exist within the CMS include:

- `Physical containment`
- `Logical containment`
- `Controlled by`

Physical Containment

The `physical containment` relationship defines the physical modules that are contained within a chassis or container.

Logical Containment

The `logical containment` relationship defines the logical modules that are components of a fault tolerant container logical module.

Often, a logical module and a physical module are represented within the CMS by the managed object.

Occasionally, logical modules can share the same physical location.

Controlled By

The `controlled by` relationship defines when one module is controlled by another.

Constraints

Each managed object has a set of constraints on its behavior. These constraints are defined by rules that fall into three categories:

- Generic rules that apply to all object classes
- Rules that apply to a particular group of object classes
- Rules that apply to individual objects

The constraints are used to maintain the system's availability and protect it against user error.

CHAPTER 4

The CMS API Object Model

In order for the CMS API to be generic across platforms, the specific object model of a platform is not visible through the CMS API. Instead, the CMS API uses objects to represent objects in the platform. These objects can be used to:

- Get attribute and relationship definitions and values from the platform objects
- Set the values of attributes of platform objects
- Register for event or exception notifications
- Get attribute information returned on an event or exception notification

The CMS API objects are owned by the user. The user is responsible for their creation and deletion.

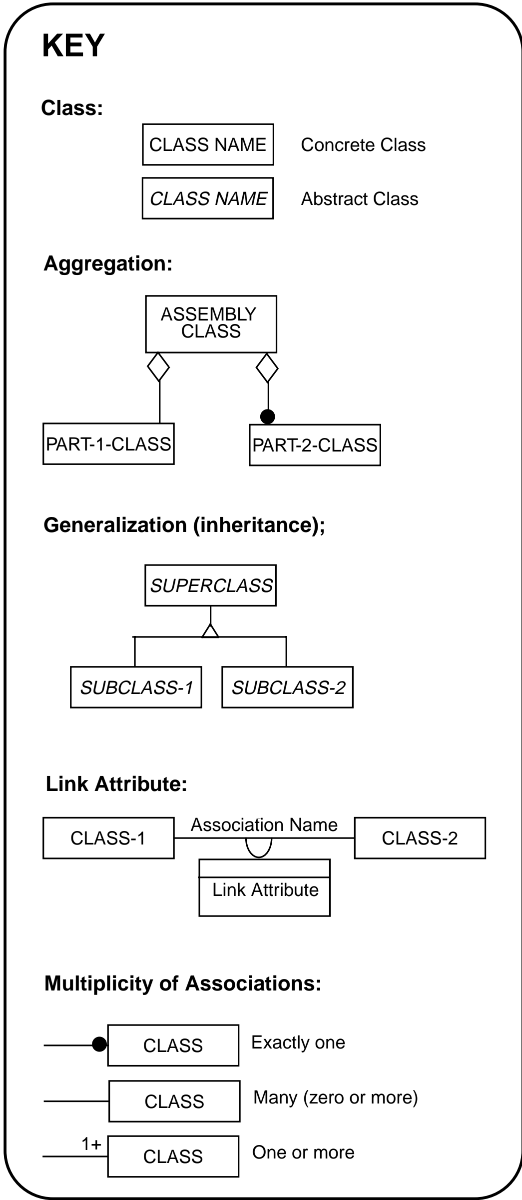
The CMS API defines the following object types:

- `Attribute`
- `Error`
- `Event`
- `Event record`
- `Exception`
- `Exception record`
- `Module`
- `Scope and filter`
- `Session`
- `System`

FIGURE 4-1 on page 19 shows the associations between the CMS API objects and the CMS server platform objects, event and exception forwarding discriminators, and `event` and `exception` notification objects. It also shows the relationships between the CMS API objects. FIGURE 4-1 uses Object Modeling Technique (OMT) notation.

The links indicate *generalization* (inheritance), *aggregation* (or containment) and *associations* between the managed objects. For example, the `notification` object is a generalization of an `event` and `exception` object. The `event` and `exception` objects inherit features from the `notification` object.

A module is an aggregation of attributes of one or more attribute objects. The contained `attribute` object can hold the attribute definition or the attribute value, and hence the `module` object's derived definition or value can be obtained.



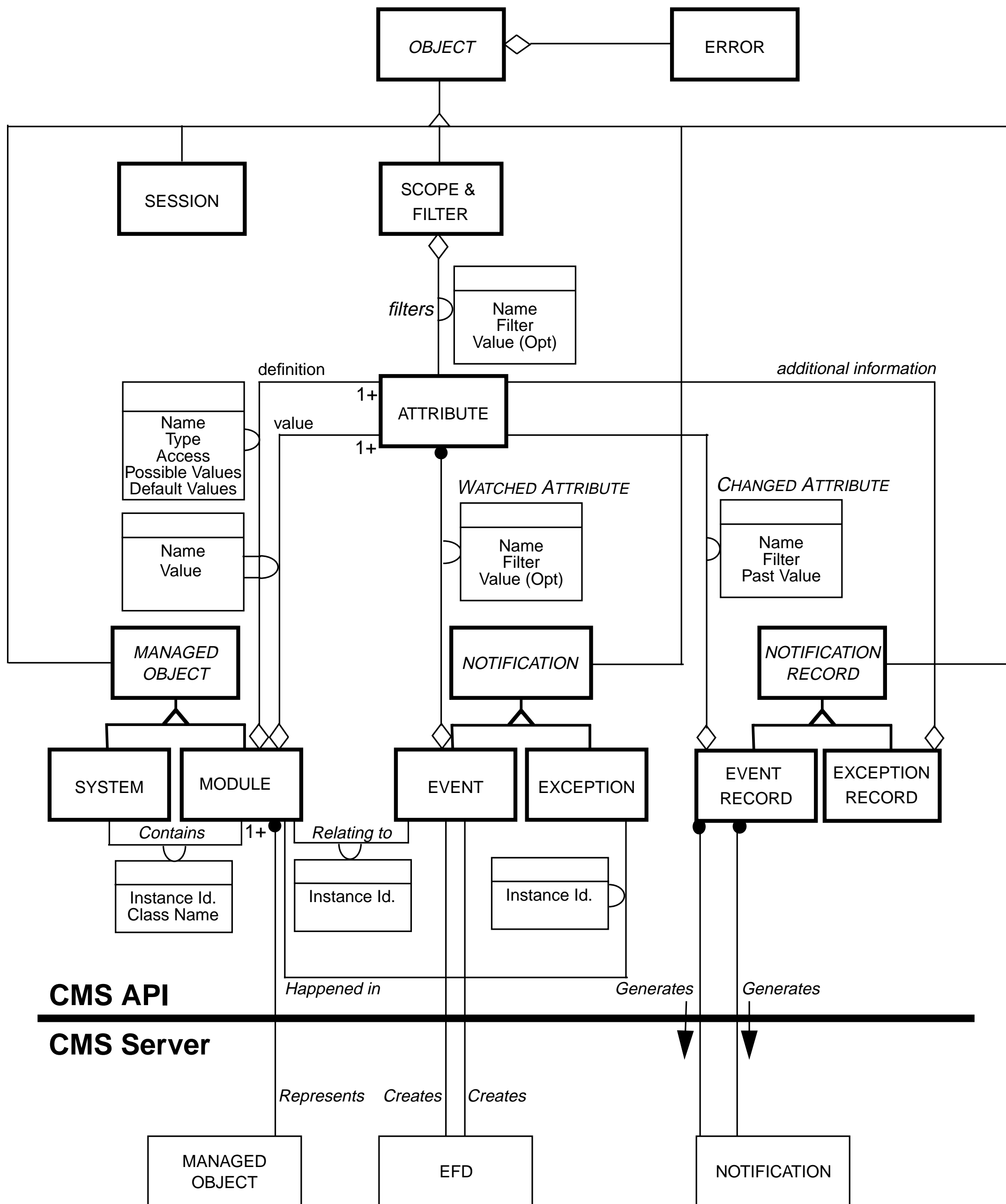


FIGURE 4-1 The CMS API Object Model

When one object references another, it is represented as an association. For example, the system object can contain the instance identifier or class name of modules, but not the actual modules themselves. The CMS API module objects represent the CMS server objects, but they are *not* the same entity. The module objects are used to take static snapshots of the managed objects they represent and are not dynamically updated when the state of a managed object changes. When the state of a managed object changes, the module object must be used to provide another static snapshot. The CMS API objects are defined in the following sections.

Note – The total number of instances of CMS API object class is defined by `CMS_maximum_number_instances`. The total number of attributes that a scope and filter or a module object can have is defined by `CMS_maximum_number_attributes`, and the length of any attribute type character string is defined by `CMS_maximum_value_length`.

CMS API Object Classes

All CMS API objects are pointers to `structs`. Therefore, access to all methods within a CMS API object is made by means of pointers to functions of the form

```
XXX_object->method()
```

Object

This abstract object is a generalization of all objects within the CMS API, with the exception of the `error` object which is used to access error information from a subclass object.

The CMS API objects form the following hierarchy:

- `Object`
  - `Session`
  - `Managed`
    - `System`
    - `Module`
  - `Attribute`
  - `Scope and filter`

- `Notification`
  - `Event`
  - `Exception`
- `Notification record`
  - `Event record`
  - `Exception record`

`session` Object

A subclass of the `Object` object.

The `session` object represents the client session with the CMS. It is used to:

- Open and close a client session with the CMS
- Read and interpret error codes returned from the CMS commands
- Optimize the performance of the CMS API depending on the needs of the application
- Specify the extent of rule checking when an application sets attribute values

---

Note – One `session` object should be used for each client connection to the CMS API.

`managed` Object

A subclass of the `Object` object.

The `managed` object is an abstract object which is generalization of the system and the module objects. A `managed` object provides access to information about the platform object module.

`system` Object

A subclass of the `managed` object.

The `system` object represents the object model of the platform. It is used to:

- Request the class names of CMS server objects in the platform model
- Request the instance names of CMS server objects in the platform model conforming to a specified scope or filter criterion

The `system` object can contain platform object class names or references (instance identifiers) to the platform object instances.

`module` Object

A subclass of the `managed` object.

The `module` object represents a CMS server platform object. It can be used to:

- Identify the physical platform object class or instance in which the client is interested
- Return attribute objects representing the attribute values of the CMS server platform object instances
- Return attribute objects representing the attribute definition of the platform object instance

The `module` object can contain one or more attribute objects representing attribute values of CMS server managed objects. It is referenced by the `system` object and `event record` object.

`attribute` Object

A subclass of the `Object` object.

The `attribute` object can be used:

- To represent an attribute of a CMS server platform object
- As a filter in a `scope and filter` object
- As the watched attribute in an `event` object
- As the changed attribute in an `event record` object

Using the `attribute` object, the client can

- Read the type, access permissions and value of an attribute
- Read the possible values and default value of an attribute
- Set an attribute to a new value
- Set a command filter
- Read the last value of a changed attribute
- Read the reason for a change

The `attribute` object forms part of a `scope and filter` object, an `event` object and an `event record` object.

`scope and filter` Object

A subclass of the `Object` object.

The `scope and filter` object defines the scope of the CMS API command. It is used in conjunction with the CMS server platform model's relationships to:

- Set the scope of a command to a platform object's dependents
- Set the scope of a command to a platform object's definitions
- Set the scope of a command to the platform model's definition
- Set the scope of a command to only platform object instances that are considered to be present, or those that are considered to be absent
- Set filters that apply to all attributes within the chosen platform object instances having a specified value or name, to select the returned set of objects or attributes

The definition of the presence of a platform object is specific to the platform.

Note – The filters used in the `scope and filter` object are `attribute` objects.

`notification` Object

A subclass of the `Object` object.

This abstract object is a generalization of `event` and `exception` objects.

The `notification` object is used for establishing and canceling event notifications relating to changes in the platform model, and exception notifications relating to events in the client's session.

`event forwarding discriminator`

The `event forwarding discriminator` (EFD) is an object that defines the criteria for determining the event notifications that are sent to the client.

`exception forwarding discriminator`

The `exception forwarding discriminator` (XFD) is an object that defines the criteria for determining the exception notifications that are sent to the client.

`event` Object

A subclass of the `notification` object.

The `event` object is used to create an EFD within the CMS server. An EFD defines the types of event of which the client wishes to be notified, and the type of information it wishes to receive in the event record. An event, in this context, is an attribute or relationship change within the CMS platform model.

Only one EFD can be used in a session. The client is notified of all events and receives the default set of information in the `event record` object.

Use the event object to:

- Set a client event handler
- Set a client-specific argument to be returned with the `event record` object

`exception` Object

A subclass of the `notification` object.

The `exception` object is used to create an XFD. An XFD defines the types of exception of which the client wishes to be notified. Only one XFD can be used in a session. The client is notified of all exceptions and receives the information in the `exception record` object.

Use the `exception` object to:

- Set a client exception handler
- Set a client-specific argument to be returned with the `exception record` object

`notification record` Object

A subclass of the `Object` object.

This abstract object is a generalization of an `event record` object and an `exception record` object.

When an event occurs, an `event record` object is passed to the client notification handler previously registered by the application.

When an exception occurs, an `exception record` object is passed to the exception notification handler previously registered by the application.

`event record` Object

A subclass of the `notification record` object.

The `event record` object returns information to the client by means of the event handler function relating to the event that has occurred. This record is generated by the EFD set up by an `event` object.

The record contains:

- A unique identifier for the EFD which generated this object
- The argument specified by the client when the event handler was registered
- the instance identifier of the CMS server platform object in which the event occurred
- An `attribute` object containing information relating to the platform object attribute that changed:
  - Name
  - Type
  - Value
  - Last value of the changed attribute
  - Reason for the event change (if available)

The `event record` object uses an `attribute` object to return information about the changed attribute.

`exception record` Object

A subclass of the `notification record` object.

The `exception record` object returns information to the client by means of the exception handler function relating to the exception that has occurred.

The record contains:

- A unique identifier for the EFD that generated this exception notification
- The argument specified by the client when the exception handler was registered
- A description of the type of exception that has occurred

`error` Object

The `error` object returns information to the client about failures that have occurred whilst using the CMS API. Each CMS object contains an `error` object.

- If an error occurs during an object method call, the object will have information relating to the failure in its error object.
- If a failure occurs during a call to a CMS API command, the session object's error object will contain information relating to the failure.

The `error` object contains:

- The class of error
- The error code
- A list of error or warning messages returned from `cmsconfig.rule`
- A text version of the error
- A more detailed text version of the error for developers

CMS API Commands

`cms_open`

The `cms_open` command opens a session with the CMS API Server. The command is used in conjunction with a `session` object, which specifies the CMS API session characteristics.

`cms_close`

The `cms_close` command closes the session with the CMS API Server. The command is used in conjunction with the `session` object opened with `cms_open`. The session is closed:

- On completion of the client's interaction
- When an unrecoverable error occurs
- When an abdication occurs

cms_request_notifications

The cms_request_notifications command registers event and exception notifications.

The command is used in conjunction with:

- An event object to register a client event handler and set up an EFD
- An exception object to register a client exception handler and set up an XFD

cms_cancel_notifications

The cms_cancel_notifications command is used to deregister event and exception notifications by deleting the EFD or XFD responsible for the notification generation.

The command is used in conjunction with:

- The event object that was used to create the EFD
- The exception object that was used to create the XFD

cms_get

The cms_get command is used to retrieve a description of the CMS server platform object model.

Use the command in conjunction with the system object to retrieve a list of platform object class names or a list of platform object instance names. A scope and filter object is used to specify the client's choice of class names or instance names. It is also used to specify scoping or filtering search criteria used to generate the list.

Use the command in conjunction with a module object to retrieve one or more attribute definitions or actual attribute values of the associated platform object. The client's choice of attribute definitions or values is specified using a scope and filter object. It is also used to specify the filtering search criteria required to generate the list of attribute objects returned.

cms_set

The cms_set command is used to modify an attribute of a CMS server platform object instance.

Use the command in conjunction with a module object to set one attribute value of the associated platform object. A scope and filter object is used to specify which attribute should be modified.

CHAPTER 5

Introduction to CMS API Functionality

Opening a CMS API Session

A CMS API session is opened with the `cms_open` command, using a `system` object to specify session characteristics.

See Chapter 6, "Connecting to and Disconnecting from the CMS" for further information on opening a CMS API session.

Registering Event and Exception Handlers

An application can register event and exception handlers which are invoked asynchronously when event and exception conditions occur.

The application uses `cms_request_notifications` in conjunction with an `event` object to register an event handler. It can then obtain information relating to the cause of the event by means of the event notification record passed to it.

The application uses `cms_request_notifications` in conjunction with an `exception` object to register an exception handler. It can then obtain information relating to the cause of the exception by means of the exception notification record passed to it.

An application can cancel the event and exception handlers with the cms_cancel_notifications command.

Obtaining the Class Names

The cms_get command is employed to obtain the class names defined in the system. When used in this context, the cms_get command uses a system object and a scope and filter object. The scope and filter object is used to specify the class names of the system.

On return from the cms_get command, the system object contains all the class names defined in the system. The system object provides methods by which the applications can access these class names.

Obtaining the Instance Identifiers of Modules of Interest

The cms_get command is employed to obtain the instance identifiers of the modules of interest. When used in this context, the cms_get command makes use of a system object and a scope and filter object.

The scope and filter object is used to specify the modules whose instance identifiers are required.

The scope specifies the depth of information returned from the object hierarchy. For example, the application can select all modules that relate to a particular module.

The filter specifies criteria by which to restrict the modules selected by the scope. The filters are based on the names and values of module attributes. Complex filter criteria can be created. For example, the application can obtain the instance identifiers of all modules that are faulty, or all modules whose location attribute is set to a particular value.

On return from the cms_get command, the system object contains the instance identifiers of the modules specified by the scope and filter object. The system object provides methods by which the application can access these instance identifiers. The module instance identifiers are then available to the application to obtain attributes of the module.

If the application already knows the instance identifiers of a particular module, it does not have to use the `cms_get` command, but can obtain the attributes of the module by specifying the known instance identifier.

Obtaining the Attributes of a Module

The application employs the `cms_get` command to obtain the attributes of a module. When used in this context, the `cms_get` command uses a `module` object and a `scope and filter` object.

The module object is used to specify the instance identifier of the module whose attributes are required.

If the instance identifier of the module is to be determined at run time according to scope and filter criteria, the `cms_get` command is used in conjunction with the `system` object and a `scope and filter` object to obtain the instance identifiers of the modules matching the scope and filter criteria. System object methods must then be used to iterate through the module instance identifiers thus obtained, as described above.

If the instance identifier of the module is already known (that is, it has been explicitly specified), it can be used directly with the `module` object employing a module object method. The `scope and filter` object can be used to define which attributes are returned and is specified in terms of `attribute` objects.

On return from the `cms_get` command, the `module` object contains the attribute objects specified by the `scope and filter` object.

Obtaining Attribute Values

The application employs the `cms_get` command to obtain the attributes of a module, as described in the section “Obtaining the Attributes of a Module” on page 31. On return, the `module` object contains `attribute` objects.

The application uses module object methods to access a specific `attribute` object. If the attribute name is known (that is, it has been specified explicitly) the object can be directly accessed using a module object method.

If the attribute name is not known, or it is necessary to access all attributes of the module, another module object method can be used to iterate through all the `attribute` objects contained in the `module` object. Attribute object methods can then be used to obtain the attribute value.

"Obtaining Instance Information About a Platform Object" on page 55 describes how to access a module's attributes in greater detail.

Accessing the Definition of a Module

The application employs the `cms_get` command to obtain the definition of a module. A module is defined by a set of attribute definitions. When used in this context, the `cms_get` command uses a `module` object and a `scope and filter` object.

The `module` object is used to specify the instance identifier of the module whose definition is required. The `scope and filter` object can be used to specify which attribute definitions are returned.

On return, the `module` object contains `attribute` objects. The application uses module object methods to access a specific attribute or iterate through all the attribute objects returned. Attribute object methods can be used to access the attribute definitions. The attribute definition includes the permitted possible values of an attribute. Attribute object methods are provided to enable the application to iterate through all the possible values.

Chapter 7, "Creating a Model of the CMS Module Classes" describes how to access the values and definition of an attribute in greater detail.

Setting the Value of an Attribute

The application employs the `cms_set` command to set the value of an attribute.

When used in this context, the `cms_set` command uses a `module` object and a `scope and filter` object. The `module` object is used to specify the instance identifier of the module whose attribute value is required. The `scope and filter` object is used to specify the attribute to be set and its new value. The `session` object is used to specify the extent of the `cmsconfig.rule` checking.

Closing the Session

The application employs the `cms_close` command to close the CMS API session. After the `cms_close` command has been invoked, the application can register a new session by invoking `cms_open` as described in the section "Opening a CMS API Session" on page 29.

The session must be closed using `cms_close` when an unrecoverable error or abdication occurs. Abdication could mean that any object model built up by the application is invalid. The application must close the session and rebuild its object model. The session object is used to obtain information relating to error conditions, and to determine their type.

"Disconnecting from the CMS" on page 37 describes how to close a CMS API session in more detail.

CHAPTER 6

Connecting to and Disconnecting from the CMS

This chapter describes how to connect and disconnect from the CMS by registering and closing a session, respectively.

Connecting to the CMS

A client connects to the CMS by registering a session via the CMS API. A session enables the client to access the CMS API commands for configuration and fault management. The CMS API provides two types of session for the client:

- A fully-featured session
- A fast, uncached session

The fully-featured session should be used by clients that intend to:

- Retrieve the platform object module
- Enable a user to modify attribute values of platform objects
- Make use of the event and exception notification mechanisms

The fast, uncached session is designed to be used in scripts when a session is created for modifying a single (or few) attribute(s) of a platform object without unnecessary overhead.

A client registers for a session within the CMS using the `cms_open` command in conjunction with a `session` object. The `session` object is used by the `cms_open` command to set the type of session and to return the client session's unique identifier.

Note – The application can have only one client session with the CMS at any one time. The same `session` object must be used when calling all other CMS API commands to identify the client session.

`session` Object

The `session` object is used by the client to specify whether the session is fully-featured or uncached. The session is fully-featured by default.

CODE EXAMPLE 6-1 Opening a CMS API Session

```
/*
 * ===============================================================
 * example1.c - opening a CMS API session
 * ---------------------------------------------------------------
 */

/* include files */
#include "example.h"

/* open_cms_session - opens the CMS API session, using the session
 * object
 */
void
open_cms_session(CMS_session_object *session_ptr)
{
    int ret;

    if ((ret = cms_create_session(session_ptr))!= CMS_success)
        cms_fatal((CMS_object) *session_ptr, ret);

    if ((ret = cms_open(*session_ptr))!= CMS_success)
        cms_fatal((CMS_object) *session_ptr, ret);
}

/* cms_fatal - prints contents of the CMS API Object's Error Object
 * to standard out and exits when an unrecoverable error occurs.
 */

void
cms_fatal(CMS_object parent, int code)
{
    CMS_error_object error = parent->error;
```

CODE EXAMPLE 6-1 Opening a CMS API Session *(Continued)*

```
    if (code == CMS_error_code_failure) {
        error = parent->error;
        printf(“CMS failure (%d):%s\n”,
            error->getCode(error),
            (char *) error->getMessage(error));
    } else {
             printf(“CMS failure (%d): %s\n”,
                    code,
                    cms_get_error_code_string(code));
    }
    exit(1);
}
```

Disconnecting from the CMS

Closing the session enables the CMS to clear the client workspace and free any client-held memory. To close the session with the CMS API, the application uses the `cms_close` command in conjunction with the `session` object. The `session` object must be the same as that used to open the session. After a session has been closed, the client will not be able to use any of the CMS API objects created during the session, other than the session object itself which should be destroyed by the user.

`session` Object

The `session` object is used to specify the unique client session identifier of the session to be closed.

CODE EXAMPLE 6-2 Closing a CMS API Session

```
/*
 * ===============================================================
 * example1.c - closing a CMS API session
 * ---------------------------------------------------------------
 */

/* include files */
#include “example.h”

/* close_cms_session - closes the CMS API session, using the
 * session object
 */
```

CODE EXAMPLE 6-2 Closing a CMS API Session *(Continued)*

```
void
close_cms_session(CMS_session_object *session_ptr)
{
    int ret;

    if ((ret = cms_close(*session_ptr)) != CMS_success)
        cms_fatal((CMS_object) *session_ptr, ret);

    (*session_ptr)->destroy(*session_ptr);
        *session_ptr = NULL;

}

/* cms_fatal - prints contents of the CMS API Object's Error Object
 * to standard out and exits when an unrecoverable error occurs.
 */

void
cms_fatal(CMS_object parent, int code)
{
    CMS_error_object error = parent->error;

    if (code == CMS_error_code_failure) {
        error = parent->error;
        printf("CMS failure (%d): %s\n",
            error->getCode(error),
            (char *) error->getMessage(error));
    } else {
             printf("CMS failure (%d): %s\n",
                    code,
                    cms_get_error_code_string(code));
    }
    exit(1);
}
```

CHAPTER 7

Creating a Model of the CMS Module Classes

Obtaining a List of the Platform Model Object Classes

The client can retrieve a list of all the object class names within the CMS Server Platform Model using the `cms_get` command in conjunction with the `session` object, the `system` object, and the `scope and filter` object.

`session` Object

This is used to:

- Specify the unique client session identifier
- Read the returned error and error string

`system` Object

This subclass of the `managed` object is used to read the returned class names.

`scope and filter` Object

This is used to scope the `cms_get` command to return class names for all the platform object classes.

Note – In the following example, the `system` object is cast to a managed object. This suppresses warning messages during compilation.

CODE EXAMPLE 7-1 Returning the CMS Object Classes

```
/*
 * ============================================================
 * get_classes_1.c - returns the object classes of the CMS
 * ------------------------------------------------------------
 */

/* include files */
#include “example.h”

/* get_cms_classes - prints the CMS class names */

void
get_cms_classes(CMS_session_object session)
{
    int                       ret;
    CMS_system_object         system;
    CMS_scope_filter_object   scope_filter;
    char        class_name[CMS_maximum_value_length+1];
    int                       number_classes;
    int                       i;

    if ((ret = cms_create_system(&system)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    if ((ret = cms_create_scope_filter(&scope_filter)) !=
                CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = scope_filter->setScope(scope_filter,
                CMS_scope_my_classes)) != CMS_success
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = cms_get(
```

CODE EXAMPLE 7-1 Returning the CMS Object Classes *(Continued)*

```
                session,
                (CMS_managed_object)system,
                scope_filter)) != CMS_success)
        cms_error(session, (CMS_object) session, ret);

    if ((ret = system->getNumberOfClasses(
                system, &number_classes)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    for (i = 0; i < number_classes; i++)
    {
        if ((ret = system->getClassName(
                system,i,class_name)) != CMS_success)
            cms_error(session, (CMS_object)system, ret);

        printf("%s\n",class_name);
    }
    scope_filter->destroy(scope_filter);
    scope_filter = NULL ;
    system->destroy(system);
    system = NULL ;
}
```

Obtaining a List of Platform Model Object Classes with a Common Attribute

The client can retrieve a list containing object class names only of those CMS server object classes that contain a particular attribute. This is achieved by applying a filter to the previous `cms_get` command.

`scope and filter` Object

This is used to filter the `cms_get` command to return class names for only those platform object classes that contain an attribute with the specified name.

CODE EXAMPLE 7-2 Returning the CMS Object Classes with a Common Attribute

```
/*
 * ============================================================
 * get_classes_2.c - returns the CMS Object Classes with attribute
 * ------------------------------------------------------------
 */

/* include files */
#include "example.h"

/* get_cms_classes_with_attribute - prints the CMS class names */
void
get_cms_classes_with_attribute(
        CMS_session_object      session,
        char                    *attribute_name)
{
    int                         ret;
    CMS_system_object           system;
    CMS_scope_filter_object     scope_filter;
    CMS_attribute_object        attribute;
    char    class_name[CMS_maximum_value_length+1];
    int                         number_classes;
    int                         i;

    if ((ret = cms_create_system(&system)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    if ((ret = cms_create_scope_filter(&scope_filter)) !=
                CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = cms_create_attribute(&attribute)) !=
                CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((attribute->setName(attribute,attribute_name)) !=
                CMS_success)
        cms_error(session, (CMS_object) attribute, ret);
```

CODE EXAMPLE 7-2 Returning the CMS Object Classes with a Common Attribute

```
    if ((attribute->setFilter(
                attribute,CMS_filter_name_equal)) !=
                CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((ret = scope_filter->setScope(
                scope_filter, CMS_scope_my_classes)) !=
                CMS_success)
    cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = scope_filter->addFilter(
                scope_filter, attribute)) != CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = cms_get(
                session,
                (CMS_managed_object)system,
                scope_filter)) != CMS_success)
        cms_error(session, (CMS_object) session, ret);

    if ((ret = system->getNumberOfClasses(
                system, &number_classes)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    for (i = 0; i < number_classes; i++)
    {
    if ((ret = system->getClassName(
            system,i,class_name)) != CMS_success)
        cms_error(session, (CMS_object)system, ret);

    printf("%s\n",class_name);
    }

    scope_filter->destroy(scope_filter);
    scope_filter = NULL;
    attribute->destroy(attribute);
    attribute = NULL;
    system->destroy(system);
    system = NULL;
}
```

Obtaining a List of the Platform Model Object Instances

The client can retrieve a list of all the object instances within the CMS server platform model using the `cms_get` command in conjunction with the `session` object, the `system` object, and a `scope and filter` object.

The CMS server creates all the instances for each class that could theoretically exist. The client can restrict the list to those object instances that are considered to be *present*. Object instances are considered present when they do not have a state of `not_present`.

The client can restrict the list to only those object instances that are present by means of a predefined `scope and filter` object.

`session` Object

This is used to:

- Specify the unique client session identifier
- Read the returned error and error string

`system` Object

This subclass of the managed object is used to read the returned instance names.

`scope and filter` Object

This is used to scope the `cms_get` command to return instance names for all the platform object instances that are considered present.

Note – In the following example, the `system` object is cast to a managed object. This prevents the issue of warning messages during compilation.

CODE EXAMPLE 7-3 Returning a List of Object Instances of the Platform Model

```
/*
 * ============================================================
 * get_instances_1.c - returns the object instance names
 * ------------------------------------------------------------
 */

/* include files */
#include "example.h"

/* get_cms_instances - prints the CMS instances names */

void
get_cms_instances(CMS_session_object session, char *class_name)
{
    int                        ret;
    CMS_system_object          system;
    CMS_scope_filter_object    scope_filter;
    CMS_attribute_object       attribute;
    char    instance_name[CMS_maximum_value_length+1];
    int                        number_instances;
    int                        i;

    if ((ret = cms_create_system(&system)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    if ((ret = cms_create_scope_filter(&scope_filter)) !=
                CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = scope_filter->setPreDefined(
                scope_filter, CMS_sf_modules_present)) !=
                CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if (class_name) {

        if ((ret = cms_create_attribute(&attribute)) !=
                    CMS_success)
```

CODE EXAMPLE 7-3 Returning a List of Object Instances of the Platform Model

```
            cms_error(session, (CMS_object) attribute, ret);

        if ((ret = attribute->setClassName(
                    attribute, class_name)) != CMS_success)
            cms_error(session, (CMS_object) attribute, ret);

        if ((ret = attribute->setFilter(
                    attribute,CMS_filter_name_equal))!=
                        CMS_success)
            cms_error(session, (CMS_object) attribute, ret);

        if ((ret = scope_filter->addFilter(
                    scope_filter, attribute)) != CMS_success)
            cms_error(session, (CMS_object) scope_filter, ret);
    }

    if ((ret = cms_get(
                session,
                (CMS_managed_object)system,
                scope_filter)) != CMS_success)
        cms_error(session, (CMS_object) session, ret);

    if ((ret = system->getNumberOfModules(
                system, &number_instances)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    printf("Number of instances is %d\n",number_instances);
    for (i = 0; i < number_instances; i++)
    {
        if ((ret = system->getInstanceIdentifier(
                    system,i,instance_name)) != CMS_success)
            cms_error(session, (CMS_object)system, ret);

        printf("%s\n",instance_name);
    }
    scope_filter->destroy(scope_filter);
    scope_filter = NULL ;
    system->destroy(system);
    system = NULL ;
}
```

Obtaining a List of Platform Model Object Instances with a Common Attribute Value

The client can retrieve a list containing object instance names restricted to those CMS server object classes that contain a particular attribute that has (or does not have) a particular value. This is achieved by applying a filter to the previous `cms_get` command.

`attribute` Object

This is used to create a filter attribute that filters object instances containing a particular attribute that either has or does not have a specified value.

`scope and filter` Object

This is used to filter the `cms_get` command to return instance names for only those platform object instances that are considered present and meet the filter criteria.

CODE EXAMPLE 7-4 Returning the Instance Names with a Common Attribute

```
/*
 * ==============================================================
 * get_instances_2.c - returns the Object instances that are
 * present
 * --------------------------------------------------------------
 */

/* include files */
#include “example.h”

/* get_cms_instances - prints the CMS instances names /*

void
get_cms_instances_attribute_value(
    CMS_session_object        session,
    char                      *attribute_name,
```

CODE EXAMPLE 7-4 Returning the Instance Names with a Common Attribute *(Continued)*

```
    char                        *attribute_value,
    int                         is_equal)
{
    int         ret;
    CMS_system_object           system;
    CMS_scope_filter_object     scope_filter;
    CMS_attribute_object        attribute;
    char        instance_name[CMS_maximum_value_length+1];
    int                         number_instances;
    int                         i;

    if ((ret = cms_create_system(&system)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    if ((ret = cms_create_scope_filter(&scope_filter)) !=
                CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = scope_filter->setPreDefined(
                scope_filter, CMS_sf_modules_present)) !=
                        CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = cms_create_attribute(&attribute)) != CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((attribute->setName(attribute,attribute_name)) !=
                CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((attribute->setValue(attribute,attribute_value)) !=
                CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((attribute->setFilter(
                attribute,
                is_equal?CMS_filter_value_equal:
                    CMS_filter_value_not_equal)) != CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((ret = scope_filter->addFilter(
                scope_filter, attribute)) != CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);
```

CODE EXAMPLE 7-4 Returning the Instance Names with a Common Attribute *(Continued)*

```
    if ((ret = cms_get(
                session,
                (CMS_managed_object)system,
                    scope_filter)) != CMS_success)
        cms_error(session, (CMS_object) session, ret);

    if ((ret = system->getNumberOfModules(
                system, &number_instances)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    for (i = 0; i < number_instances; i++)
    {
        if ((ret = system->getInstanceIdentifier(
                    system,i,instance_name)) != CMS_success)
            cms_error(session, (CMS_object)system, ret);

        printf("%s\n",instance_name);
    }
    scope_filter->destroy(scope_filter);
    scope_filter = NULL ;
    attribute->destroy(attribute);
    attribute = NULL ;
    system->destroy(system);
    system = NULL ;
}
```

Obtaining a List of Platform Model Object Instances that are Faulty

The client can retrieve a list containing object instance names of only those CMS server object instances that are considered to be *faulty*. The definition of when an object instance is considered faulty is specific to the platform.

The client restricts the list to only those object instances that are faulty, using a predefined `scope and filter` object with the original `cms_get` command.

`scope and filter` Object

This is used to filter the `cms_get` command to return instance names for only those platform object instances that are considered *present* and *faulty.*

CODE EXAMPLE 7-5 Obtaining the Faulty Object Instances

```
/*
 * ================================================================
 * get_instances_3.c - returns the Object Instances that are faulty
 * ----------------------------------------------------------------
 */

/* include files */
#include "example.h"

/* get_cms_faulty_instances - prints the CMS instances names /*

void
get_cms_faulty_instances(CMS_session_object session)
{
    int                       ret;
    CMS_system_object         system;
    CMS_scope_filter_object   scope_filter;
    char       instance_name[CMS_maximum_value_length+1];
    int                       number_instances;
    int                       i;

    if ((ret = cms_create_system(&system)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    if ((ret = cms_create_scope_filter(&scope_filter)) !=
               CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = scope_filter->setPreDefined(
               scope_filter, CMS_sf_modules_faulty)) !=
                       CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = cms_get(
               session,
               (CMS_managed_object)system,
                       scope_filter)) != CMS_success)
        cms_error(session, (CMS_object) session, ret);
```

CODE EXAMPLE 7-5 Obtaining the Faulty Object Instances *(Continued)*

```
    if ((ret = system->getNumberOfModules(
                system, &number_instances)) != CMS_success)
        cms_error(session, (CMS_object) system, ret);

    printf("Number of instances is %d\n",number_instances);
    for (i = 0; i < number_instances; i++)
    {
        if ((ret = system->getInstanceIdentifier(
                    system,i,instance_name)) != CMS_success)
            cms_error(session, (CMS_object)system, ret);

        printf("%s\n",instance_name);
    }
    scope_filter->destroy(scope_filter);
    scope_filter = NULL ;
    system->destroy(system);
    system = NULL ;
}
```

Obtaining Class Information About a Platform Object

The client can retrieve the definitions of the attributes contained by the CMS server platform object class using the `cms_get` command in conjunction with the `session` object, the `module` object, `attribute` objects and a `scope and filter` object.

`session` Object

This is used to:

- Specify the unique client session identifier
- Read the returned error and error string

`module` Object

This subclass of the `managed` object is used to:

- Specify the instance of platform object to be interrogated
- Return a group of attribute objects representing the attribute definitions of the platform object class

`scope and filter` Object

This is used to:

- Scope the `cms_set` command to get the definition of the platform object class
- Select for retrieval only those definitions with specific attributes

`attribute` Object

This is used to:

- Read the returned definition of each requested attribute
- Specify what filtering is required (if any)

CODE EXAMPLE 7-6 Obtaining Class Information

```
/*
 * get_definitions _1.c
 *
 *
 */

/* include files */
#include "example.h"

void
print_attribute_defintion(CMS_attribute_object a)
{
    int n;
    int i;

    printf("Name:    \t%s\n",a->getName(a));
    printf("Type:    \t%s\n",
                a>getType(a)==CMS_attribute_type_string?
            "string":"invalid");
    printf("Access: \t%s\n",
                a->getAccess(a)==CMS_attribute_access_system?
```

CODE EXAMPLE 7-6 Obtaining Class Information *(Continued)*

```
            “system”:
            a->getAccess(a)==CMS_attribute_access_user?
            “user”:”invalid”);
    printf(“Default:\t%s\n”,a->getDefaultValue(a));

    n = a->getNumberPossibleValues(a);
    printf(“Values: \t”);
    for (i = 0; i < n ; i++) {
    printf(“%s%s”,a->getPossibleValue(a,i), i+1!=n?”, “:” “);
    }
    printf(“\n”);
}

/* get_cms_instances - prints the CMS instances names
 */
void
get_cms_attribute_defintions(
    CMS_session_object        session,
    char                      *class_name,
    char                      *attribute_name)
{
    int                       ret;
    CMS_module_object         module;
    CMS_scope_filter_object   scope_filter;
    CMS_attribute_object      attribute;
    CMS_attribute_object      returned_attribute;
    char    instance_name[CMS_maximum_value_length+1];
    int                       number_attributes;
    int                       i;

    if ((ret = cms_create_module(&module)) != CMS_success)
    cms_error(session, (CMS_object) module, ret);

    if ((ret = cms_create_scope_filter(&scope_filter)) !=
            CMS_success)
    cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = scope_filter->setScope(
        scope_filter, CMS_scope_my_definition)) !=
            CMS_success)
    cms_error(session, (CMS_object) scope_filter, ret);

    if (attribute_name) {
    if ((ret = cms_create_attribute(&attribute)) !=
            CMS_success)
        cms_error(session, (CMS_object) attribute, ret);
```

CODE EXAMPLE 7-6 Obtaining Class Information *(Continued)*

```
    if ((attribute->setName(attribute,attribute_name))!=
            CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((attribute->setFilter(
            attribute, CMS_filter_name_equal))!=
            CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((ret = scope_filter->addFilter(
            scope_filter, attribute)) != CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);
    }

    if ((ret = module->setClassName(module,class_name)) !=
            CMS_success)
    cms_error(session, (CMS_object) module, ret);

    if ((ret = cms_get(
        session,
        (CMS_managed_object)module,
        scope_filter)) != CMS_success)
    cms_error(session, (CMS_object) session, ret);

    if (attribute_name) {
    if ((ret = module->getAttributeWithName(
            module,
            attribute_name,
            &returned_attribute)) != CMS_success)
        cms_error(session, (CMS_object) module, ret);

    print_attribute_defintion(returned_attribute);

    attribute->destroy(attribute);
    attribute = NULL ;
    }
    else {

        if ((ret = module->getNumberOfAttributes(
                module, &number_attributes)) != CMS_success)
            cms_error(session, (CMS_object) module, ret);

        for (i = 0; i < number_attributes; i++ ) {
            if ((ret = module->getAttributeAtNumber(
                module,
                i,
                &returned_attribute)) != CMS_success)
```

CODE EXAMPLE 7-6 Obtaining Class Information *(Continued)*

```
            cms_error(session, (CMS_object) module, ret);
            print_attribute_defintion(returned_attribute);
            printf("\n");
        }
    }
    scope_filter->destroy(scope_filter);
    scope_filter = NULL ;
    module->destroy(module);
    module = NULL ;
}
```

Obtaining Instance Information About a Platform Object

The client can retrieve the value of one or more attributes contained by the CMS server platform object class using the `cms_get` command in conjunction with the `session` object, the `module` object, `attribute` objects and a `scope and filter` object.

`session` Object

This is used to:

- Specify the unique client session identifier
- Read the returned error and error string

`module` Object

This subclass of the managed object is used to:

- Specify the instance of platform object to be interrogated
- Return a group of attribute objects representing the attribute definitions of the platform object class

`scope and filter` Object

This is used to:

- Scope the `cms_get` command to return the definition of the platform object class
- Retrieve definitions for specific attributes only; the attributes can be selected using filters

`attribute` Object

This is used to:

- Read the returned definition of each requested attribute
- Specify the filtering required (if any)

CODE EXAMPLE 7-7 Obtaining Instance Information

```
/* include files */
#include "example.h"

void
print_attribute_defintion(CMS_attribute_object a)
{
    int n;
    int i;

    printf("Name:   \t%s\n",a->getName(a));
    printf("Type:   \t%s\n",
        a->getType(a)==CMS_attribute_type_string?
            "string":"invalid");
    printf("Access: \t%s\n",
        a->getAccess(a)==CMS_attribute_access_system?
            "system":
            a->getAccess(a)==CMS_attribute_access_user?
            "user":"invalid");
    printf("Default:\t%s\n",a->getDefaultValue(a));

    n = a->getNumberPossibleValues(a);
    printf("Values: \t");
    for (i = 0; i < n ; i++) {
        printf("%s%s",a->getPossibleValue(a,i), i+1!=n?", ":" ");
    }
    printf("\n");
```

Setting Attribute Values of a Platform Object Instance

The client can set the value of an attribute contained by the CMS server platform object class using the `cms_set` command in conjunction with the `session` object, the `module` object, an `attribute` object and a `scope and filter` object.

`session` Object

This is used to:

- Specify the unique client session identifier
- Read the returned error and error string

`module` Object

This subclass of the `managed` object is used to specify the platform object instance to be modified.

`attribute` Object

This is used to specify the name and value of the attribute of the platform object instance that forms the filter.

scope and filter Object

This is used to scope the cms_set command to the attribute name of the platform object instance specified by the platform object class.

CODE EXAMPLE 7-8 Setting Attribute Values

```
/*
 * ================================================================
 * set_classes.c - sets the attribute values of an object instance
 * ----------------------------------------------------------------
 */

/* include files */
#include "example.h"

/* get_cms_attribute_values - prints the CMS instances names */

void
set_cms_attribute_value(
    CMS_session_object        session,
    char                      *class_name,
    int                       instance_number,
    char                      *attribute_name,
    char                      *attribute_value)
{
    int                       ret;
    CMS_module_object         module;
    CMS_scope_filter_object   scope_filter;
    CMS_attribute_object      attribute;
    char       instance_name[CMS_maximum_value_length+1];
    int                       number_attributes;
    int                       i;

    if ((ret = cms_create_module(&module)) != CMS_success)
        cms_error(session, (CMS_object) module, ret);

    if ((ret = cms_create_scope_filter(&scope_filter)) !=
               CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    if ((ret = scope_filter->setScope(
               scope_filter, CMS_scope_me)) != CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);
```

CODE EXAMPLE 7-8 Setting Attribute Values *(Continued)*

```
    if ((ret = cms_create_attribute(&attribute)) != CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((attribute->setName(attribute,attribute_name)) !=
                CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((attribute->setValue(attribute,attribute_value)) !=
                CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((attribute->setFilter(
                attribute, CMS_filter_name_equal))!=CMS_success)
        cms_error(session, (CMS_object) attribute, ret);

    if ((ret = scope_filter->addFilter(
                scope_filter, attribute)) != CMS_success)
        cms_error(session, (CMS_object) scope_filter, ret);

    sprintf(instance_name,"%s %d",class_name,instance_number);
    if ((ret = module->setInstanceIdentifier(
                module,instance_name)) != CMS_success)
        cms_error(session, (CMS_object) module, ret);

    if ((ret = cms_set(
                session,
                (CMS_managed_object)module,
                scope_filter)) != CMS_success)
        cms_error(session, (CMS_object) session, ret);

    attribute->destroy(attribute);
    attribute = NULL;
    scope_filter->destroy(scope_filter);
    scope_filter = NULL ;
    module->destroy(module);
    module = NULL ;
}
```

Glossary

abstract class A class having no instances but specifying the common characteristics of its subclasses.

ASIC Application-Specific Integrated Circuit.

ASR Automatic System Recovery: reboot on system hang.

attribute A data value held by instances of a class. The class defines the unique attribute names that each instance of the class contain, but each instance within that class can have a different value for any particular attribute name.

behavior A set of responses of a managed object in the event of a derived state change or a specific attribute change.

BMX+ Crossbar switch *ASIC*.

bridge The interface between the *CPUsets* and the I/O devices.

CAF Console, Alarms and Fans *module*.

class A group of objects with similar characteristics that respond to the same set of commands, have the same attributes and respond in the same way to each command.

CMS Configuration Management System. The software that records and monitors the *modules* in the system. Users access the CMS via a set of utilities which they use to add and remove modules from the system configuration and *enable* and *disable* modules that are in the system configuration.

component An identifiable part of a *module*.

condition A Boolean function of object values.

configure (*CMS*) Notify the CMS that a *module* is present in a specified location.

constituent (*CMS*) An object that provides part of the functionality of another object. An object references its constituents.

CPUset A *module* containing the system processors and associated components.

craft-replaceable A *module* which clearly indicates when it is faulty and can be *hot-replaced* by a trained craftsperson.

DIMM Dual Inline Memory Module.

disable (*CMS*). Bring offline and power down a *module*.

DMA Direct Memory Access.

DRAM Dynamic Random Access Memory.

DSK Disk chassis *module*.

DVMA Direct Virtual Memory Access. A mechanism to enable a device on the PCI bus to initiate data transfers between it and the CPUsets.

ECC Error Correcting Code.

EEPROM Electrically Erasable Programmable Read Only Memory.

EFD Event Forwarding Discriminator

EMI Electro-magnetic Interference.

enable (*CMS*) Power up and bring online a *module* that is already *configured* into the system.

engineer-replaceable A *module* which may not indicate that it is faulty and which may require special tools for diagnosis and replacement. The Netra ft 1800 does not have any engineer-replacable modules.

ESD ElectroStatic Discharge.

EState Error limitation mode.

fault tolerant A system in which no single hardware failure can disrupt system operation.

fault-free No faults are evident in the operating system, or application software, or in external systems, except in the case of certain high demand real-time uses.

faulty module A *module* one or more of whose devices have gone into the degraded or failed states, as indicated to the *CMS* via the *hot-plug* device driver framework.

FPGA Field Programmable Gate Array.

front-replaceable The ability to replace a *module* from the front of the system.

FRU Field Replacable Unit. Another name for a *module*, used within the *CMS*.

generalization See *inheritance*.

HDD Hard Disk Drive.

hardened Specially engineered to be resistant to hardware and some causes of software failure. Applies to device drivers.

health features Features that can indicate that a fault is about to occur.

hot plug The ability to insert or remove a *module* without causing an interruption of service to the operating platform.

hotPCI An implementation of the PCI bus designed to minimize the probability that a fault on a *module* will corrupt the bus, and so to ensure that the system control mechanism runs without interruption

hot-replaceable A *module* that can be replaced without stopping the system.

I^2C Inter Integrated Circuit

inheritance A relationship between classes organized into a hierarchy whereby a class lower in the hierarchy receives (inherits) the methods and attributes of the class from which it was derived.

IOMMU Input/Output Memory Management Unit

LED Light-Emitting Diode.

location A slot where a *module* can be inserted. Each location has a unique name and is clearly marked on the chassis.

lockstep The process by which two *CPUsets* work in synchronization.

losing side The side of a *split* system which has a new identity when rebooted.

managed object A representation of the physical components within the machine and the higher level objects that represent components of physical devices.

MBD Motherboard.

Mbus Maintenance bus.

module An assembly that can be replaced without requiring the base machine to be returned to the factory. A module is a physical assembly that has a module number which is stored in the software on the machine, generally in the *EEPROM* of the physical assembly

PCI Peripheral Component Interconnect.

PCIO PCI-to-Ebus2/Ethernet controller *ASIC*.

PRI Processor re-integration. The process by which the two CPUsets come into *lockstep* to function as a fault tolerant system. *Re-integration* is preferred.

PROM Programmable Read Only Memory.

PSU Power Supply Unit.

RAS Reliability, Availability and Serviceability.

RCP Remote Control Processor.

relationship A physical or conceptual connection between object instances, described by an attribute.

RMM Removable Media Module.

RS232 An EIA specification that defines the interface between DTE and DCE using asynchronous binary data interchange.

SC_UP+ System controller *ASIC*.

side One *CPUset* and its associated *modules*, capable of running as a standalone system. A side is one half of a *fault tolerant* system or one of two systems in a *split* system.

SPF Single Point of Failure.

split system A system whose two *sides* run as separate systems.

state transition A change of state caused by an event.

stealthy PRI Stealthy processor re-integration. Processor re-integration (*PRI*) which is completed without user intervention.

subclass A class derived from an existing class (its *superclass*) an which inherits its protocol. An instance of a subclass can therefore respond to the same commands and has the same attributes as its superclass. It can also contain additional attributes and respond to additional commands.

subsystem (*CMS*) A fault tolerant configuration of *modules* defined in the CMS.

superclass The class from which a *subclass* inherits its functionality.

system attribute (*CMS*) An attribute of a CMS object that is written only by the CMS.

surviving side The side of a *split* system which retains the identity of the previous *fault tolerant* system.

TLB Translation Lookaside Buffer. The hardware which handles the mapping of virtual addresses to real addresses.

TMN Telecommunication Managed Networks

U2P UPA-to-PCI bridge (U2P) *ASIC*.

UPA UltraSPARC Port Architecture.

user attribute An attribute whose value can be set by a non-system object.

XFD Exception Forwarding Discriminator

Index

SYMBOLS

_action, 11, 12
_driver_state, 11, 12

A

aggregation, 17
associations, 17, 20
attribute, 22, 47, 52, 56, 57
attribute values
  obtaining, 31
  setting, 32
attributes, 11
  system, 11
  user, 12

B

behavior, 15
  constraints, 16

C

class names, 30
CMS API
  commands, 6, 26
  instances, 5
  object model, 17
  object types, 5, 17
  server, 5
CMS server, 5
CMS_maximum_number_attributes, 20
CMS_maximum_number_instances, 20
CMS_maximum_value_length, 20
cmsconfig(), 2, 12
cmsfix(), 2, 12
cmsfruinfo(), 2
commands, 6
  cms_cancel_notifications, 27, 30
  cms_close, 26
  cms_get, 7, 27, 30
  cms_open, 26, 29
  cms_request_notifications, 27, 29
  cms_set, 7, 28
condition, 7
connecting, 35
containment, 17

D

description, 12
disconnecting, 37

E

error, 26
event, 24
event forwarding discriminator, 8, 17
event forwarding discriminator, 23
event handlers, 29

event record, 25
exception, 24
exception forwarding discriminator, 8, 17
exception forwarding discriminator, 23
exception handlers, 29
exception record, 25

F

fault management, 2
fault_acknowledged, 12

G

generalization, 17

I

inheritance, 17
instance identifiers, 30

L

location, 12

M

managed, 21
managed objects, 8, 11, 20
module, 22, 52, 55, 57
module attributes
  obtaining, 31
module definition, 32

N

network management environments, 3
notification, 23
notification record, 24

O

Object, 20
object
  attributes, 7
  class, 6
  relationships, 7
object definition files, 6
object states
  degraded, 13
  failed, 13
  initialising, 13
  offline, 13
  online, 13
object types, 5, 17, 20
  attribute, 22, 47, 52, 56, 57
  error, 26
  event, 24
  event forwarding discriminator, 23
  event record, 25
  exception, 24
  exception forwarding discriminator, 23
  exception record, 25
  hierarchy, 20
  managed, 21
  module, 22, 52, 55, 57
  notification, 23
  notification record, 24
  Object, 20
  scope and filter, 23, 40, 42, 44, 47, 50, 52, 56, 58
  session, 21, 36, 37, 39, 44, 51, 55, 57
  system, 21, 39, 44
obtaining
  attribute values, 31
  class names, 30
  instance identifiers, 30
  module attributes, 31
  platform model object classes, 39
    with common attribute, 41
  platform model object instances, 44
    faulty, 49
    with common attribute, 47
  platform object
    class information, 51
    instance information, 55

P

platform model, 6
  object classes, 39
    with common attribute, 41
  object instances, 44
    faulty, 49
    with common attribute, 47

R

registering a session, 35

S

`scope and filter`, 23, 40, 42, 44, 47, 50, 52, 56, 58
server object model, 11
session
  closing, 37
  fully-featured, 35
  registering, 35
  uncached, 35
`session`, 21, 36, 37, 39, 44, 51, 55, 57
setting
  attribute values, 32
  platform object attribute values, 57
state, 7
`state`, 11
state transition, 7, 8
`system`, 21, 39, 44

U

user commands, 2
`user_label`, 12

X

`xcmsfix()`, 2