Netra™ ft 1800
CMS 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. 805-7899-10
February 1999, Revision A

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

Please
Recycle

Adobe PostScript™

Contents

Figures

Tables

Code Samples

Preface

System management of the Netra ft 1800 is implemented by the Configuration Management System (CMS) comprising an object model database which is controlled by an automated state machine. Each object in the database represents a software abstraction of a Netra ft 1800 platform component, and is defined by a text-based definition file called a `cmsdef` file.

This guide provides an operational view of the Configuration Management System (CMS) of the Netra ft 1800. It explains the design, constraints and objectives of the Netra ft 1800 `cmsdef` model and is essential reading for anyone who needs to maintain or write a `cmsdef.`

The guide is also useful, in conjunction with the *Netra ft 1800 CMS API Developer's Guide* (Part Number 805-5870-10), for programmers who want to use the Application Programming Interface (API) to manipulate CMS objects.

How This Book Is Organized

Chapter 1 describes the CMS framework, shows how the CMS is used to model the system configuration, introduces the concept of *state*, and shows how the CMS constrains changes in state.

Chapter 2 describes the User and System Models.

Chapter 3 describes the FRU and Device State Models.

Chapter 4 provides a guide to writing a `cmsdef`.

Appendix A contains a sample `cmsdef` for a device.

Glossary is a list of words and phrases and their definitions.

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 API Development	*Netra ft 1800 CMS API Developer's Guide*	805-5870-10

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

System Management of the Netra ft 1800

This chapter provides an operational overview of the Netra ft 1800 Configuration Management System (CMS).

The CMS is an essential component of the fault tolerant platform. It provides configuration management for a telecommunications environment, addressing the needs for remote management and alarms.

The CMS provides the following functionality:

- enables the definition of an ideal configuration against which the system can be compared
- oversees control of fault tolerance and automatic recovery
- manages the redundancy of core components
- provides early diagnosis and notification of faults
- manages auto re-integration of degraded components
- assists hot replacement of degraded or faulty components
- provides the means for remote management

How Does the CMS Work?

The CMS achieves the required functionality of both the active systems management and the remote management interfaces by providing a framework for managing a software abstraction of the platform.

The Platform Model

The platform is represented as a directed graph of managed objects. The following types of object exist for a Netra ft platform:

- **Field Replaceable Units** (FRUs) represent the physical modules. These can be thought of as containers for devices. They can also represent environmental characteristics of the hardware modules such as temperature, fault LEDs, or the status of fans contained within them.
- **Devices** represent the logical realization of the physical functionality of hardware components, such as a disk or a tape. In general, a CMS device maps directly to a Solaris device driver instance. The CMS therefore provides a representation of the device driver which, in turn, provides a representation of the physical functionality of the hardware components.
- **Virtual devices** represent an aggregation of similar devices that present a single pseudo device interface to their user, such as two Ethernets that comprise a single ft Ethernet device.
- **Services** represent an aggregation of devices that present a service to the user. For example, a disk and an Ethernet device can make up an ftp service. The system, which is the logical realization of the Netra ft platform, is a specialized type of service.

Other types of object can exist.

Framework for Management

The framework provides an infrastructure for managing the objects using attributes. The entire configuration and manipulation of an object is controlled by setting its attribute values. Attribute values can be set by the user, the actions of an object, or by the platform component that the object represents.

For example:

- The user sets an attribute to specify the platform locator of an object (that is, the location attribute).

  This causes the object to interface with the platform to see if the managed platform component is present at the selected locator.
- The user sets an attribute to request a change in the object's operational state.

  This causes the object to interface to the managed platform component to cause it to enter the state requested by the user (usually through the driver control command `u4ftctl`).
- An action from an object sets an attribute in another object.

  This causes the other object to act on the attribute change.

- A managed platform component sets an attribute representing its state.

  This causes the object to update its state and trigger actions associated with the change of state, such as external notifications.

The CMS creates an interface for defining objects to represent platform components, and provides an interface by which object attributes can be read and set by the user, object, or by the platform component that is being represented. The configuration of the platform components and their interfaces is defined by the objects for the given platform.

The CMS provides services to the objects and user interface such as the ability to assess the impact of an attribute change on all the objects before permitting its update, limited scoping and filtering abilities for the user, and an asynchronous notification of unexpected changes in the objects.

The Objectives of the CMS

The objectives of the CMS are to provide configuration management for a Telco environment and address the needs for remote management and alarms. In support of this requirement, the CMS has a number of design objectives.

Providing a Simple User Interface for Hot Plugging

The CMS provides a simple, consistent user interface for manipulating FRUs and devices based on the principles of object management and the representation of object methods as attribute values. The CMS uses the same mechanisms to enable the user to specify redundant and hardware fault tolerant system configurations. The CMS hides the complexity of driver `ioctls` and maintenance bus utilities and presents a simple system image to the user.

Separating Policy From Mechanism

The CMS enables platform developers to move management of their platform components out of the kernel to user level object definitions.

This means that decisions can be made by the objects based on the system policies. The CMS also provides an event notification interface enabling remote managers to make decisions based on network policies.

Providing an Operational System State Blueprint

The CMS provides an infrastructure for platform developers to create objects that can be instantiated by the user. The user can also specify a required operational state for the system which the CMS will attempt to maintain and will report any changes from this operational state.

Components on Which the CMS Is Dependent

The CMS is dependent on the input received via the component interface. It assumes that device drivers are hardened and therefore follow the u4ft Nexus bus device state framework. It assumes that all state or association changes will be reported via the `u4FTlog:cms` pseudo device. It is also dependent on the hardware, that is, FRUs, providing a means of identification.

What Comprises the CMS?

The CMS comprises the following key functional areas:

- `cmsd`
- `cmsdef`
- `cmsconfig.rule`

`cmsd` – The CMS Core

The core of the CMS is `cmsd`(1M), a daemon process that models the actual configuration of the system based on the theoretical configuration specified in the system definition files. `cmsd` is started during the transition to run level 2 by means of the system startup script `S90CMS`.

The model held by `cmsd` is adjusted in response to events signalled to it by device drivers and other managed objects through the system logging device, `u4ftlog`. The model can also be adjusted by the system administrator by means of the user interfaces to the CMS.

When the model held by `cmsd` changes, `cmsd` executes shell command lines specified in the system definition files.

The model held by `cmsd` persists across a reboot. `cmsd` achieves this by writing all attributes that are not set to their default value to a private database. The database comprises flat files (`configfile` and `configfile-`) duplicated and checksummed for security. When cmsd restarts, it restores the values of all user attributes (that is, attributes that can be set using the user interfaces to the CMS) to the values defined in the database.

Commands specified in S90CMS then attempt to restore the actual system configuration to that specified by the user attributes. This is achieved by setting attribute values, whereupon the responses specified in the system definition effect the desired changes to the system configuration.

An application programming interface (API) is provided to enable third party applications to interface to the CMS. The CMS API comprises a C include file and a dynamically-linked library for use by applications. The CMS API communicates with cmsd by means of a UNIX domain socket. It maintains a local cache of cmsd's model of the state of the system. The local cache is initialized using a *snapshot* of the entire state of the system when the communication link is established. The local cache is then updated in real time by events received from cmsd when it modifies its model of the system state. The local cache is available to the application by means of a set of access functions. The CMS API also provides an asynchronous event and exception notification mechanism.

The CMS API provides functions that enable an application to modify the values of user attributes. Constraints are specified in cmsconfig.rule(4) to restrict such requests. This is intended to prevent user requests from jeopardizing the integrity of the system. For information about the function and use of the CMS API, refer to the *Netra ft 1800 CMS API Developer's Guide* (Part Number 805-5870-10).

The CMS core includes a single system definition for the null object. Objects that can theoretically reference other objects, but which currently do not, are configured to reference the null object.

cmsdef

A cmsdef file exists for each class of object managed by the CMS (for example, disk SCSI controller, motherboard). Each cmsdef specifies the following:

- The maximum number of instances for the class of object specified.
- The state model for the object.

  The state model of an object is expressed in terms of other attribute values and the state of objects related to it.
- The behavior of the object when state transitions occur.

  Typically, this behavior will be encapsulated in a state transition-specific shell script, which the CMS developer is responsible for specifying.
- The attributes of the object and the permitted values for these attributes.

  All CMS objects are managed by assigning values to attributes. This can cause the state of the object to change which, in turn, can trigger behavior associated with the state change. Behavior can also be triggered by attribute changes.

- Related objects.

  A single relationship type exists which is expressed using attributes. The relationship is used in ways that are intrinsic to the CMS and to `cmsconfig` and `cmsfix`, and in ways specified by the developer of the `cmsdef`.
- The owner of each attribute, either *user* or *system*.

  User attributes can be set using `cmsconfig` and `cmsfix`. System attributes can be set only by events received from the logging device – that is, from the CMS object itself or from the object it manages. User and system attributes differ in that changing user attributes is subject to more stringent controls than changing system attributes. The attributes of the user utilities `cmsconfig` and `cmsfix` can also be made invisible.

Refer to `cmsdef`(4) for a specification of `cmsdef`s and a general description of the way they are constructed, and to Chapter 4, “Writing `cmsdef`s” for more specific examples.

A `cmsdef` represents a software abstraction of a component of the Netra ft platform. In this sense, a software abstraction is a CMS software representation of a FRU or a device. This representation can model characteristics of the component, such as its device driver instance state or the power-on status of a FRU. This representation can also be used to elicit behavior from the component. For example, a user can manipulate the representation so that it attempts to enable or disable a device, or power on or power off a FRU.

A `cmsdef` exists for each of the following types of software abstraction:

- FRUs
- Devices
- Virtual devices
- Services

See also “The Platform Model” on page 2.

`cmsconfig.rule`

The single `cmsconfig.rule` file specifies the constraints applied when the user attempts to modify attribute values. Each constraint is expressed using a rule in the `cmsconfig.rule` file.

Rules can be specified using varying levels of generality. For example, system wide rules can be specified that apply to all objects. Since rules evaluate attributes, a consistent approach to the specification of `cmsdef` attributes is required. Conversely, rules can be specified for particular objects and attributes.

Certain rules are used to ensure consistency of the data model used by the CMS, particularly in the protection of relationship constituents. Other rules are used to ensure that the services supported by the system are not compromised by erroneous commands by the user.

Note that attempts to set an attribute to a value not specified in its set of permitted values contained in the associated `cmsdef` will fail. This behavior is built into the CMS and does not involve the specification of rules in `cmsconfig.rule`.

The man page `cmsconfig.rule`(4) gives the syntax of `cmsconfig.rule` and provides a simple description of how to specify the contents of the file.

CHAPTER 2

CMS User and System Models

The CMS is designed so that only those objects and attributes that are necessary for configuring, operating and dealing with any fault conditions in the Netra ft 1800 are visible to the user.

The CMS therefore provides two models of the system:

- A User Model that provides a simple interface for user management and maintenance tasks
- A System Model that is concerned with the automated functions and system management

The CMS implements these two model by means of hidden constituents and hidden system attributes. As users have only a limited view of the system, the CMS developer must ensure that users can diagnose from their user model the appropriate action to take to recover from any state the machine may be in.

CMS User Model

This section describes how the CMS models the user's view of the system and the functions that the user can perform.

Objects

The CMS user model comprises two classes of object:

- Field Replaceable Units (FRUs) that represent physical modules in specific locations
- Services that represent combinations of modules configured as subsystems

FRUs

TABLE 2-1 lists the eight types of FRU that can be found in the Netra ft 1800, together with the maximum number of instances that each FRU can have.

TABLE 2-1 FRU Objects in the CMS User Model

Object	Representing	Instances
`A-MBD`	Motherboard	1
`B-MBD`	Motherboard	1
`CAF`	Console, alarms and fans	2
`CPU`	CPUsets	2
`DSK`	Disk chassis	2
`HDD`	Disk drives	12
`PCI`	PCI cards	16
`PSU`	Power supply units	6
`RMM`	Removable media modules	2

Services

The user model contains the following services:

- `ft_network` (network subsystem)
- `sm` (console subsystem)
- `ft_alarm` (alarm subsystem)
- `ft_core` (processor subsystem)

Note – `ft_core` is considered to be a service but its behavior is similar to that of a FRU.

Relationships

The user model contains the following types of relationship.

FRU to FRU Relationships

Associations exist between the FRUs which represent physical dependencies, such as power management, and which are set up automatically by the CMS using the slot `location` attribute of the FRU.

These physical dependencies are encapsulated in the slot location names on the system chassis and, although obvious to users, it is unlikely that they will need to use these links explicitly. However, the relationships are used by the CMS to ensure the integrity of the system.

Service to FRU Relationships

The following relationships exist between services and FRUs:

- `ft_core` is the subsystem object for the CPU FRUs.
- `ft_network` is the multiplexer device for the CAF and PCI FRUs (network controllers)
- `sm` is the multiplexer device for the CAF (serial device)
- `ft_alarm` is the multiplexer device for the CAF (alarms device)

Managing Objects

A user can perform the following operations:

- Enable or disable a FRU
- Configure or unconfigure a FRU
- Modify the behavior of the FRU
- Configure or unconfigure links to devices
- Modify the behavior of a service

When a FRU is enabled or disabled, the CMS links or unlinks the service from the FRU. This does not change its configuration, but can cause the service to be brought online or taken offline.

Visible Attributes and Constituents

The user can see only the attributes and constituents of FRUs and services that comprise the User Model. Other attributes and constituents, including those to device objects, are hidden from the user.

For a detailed explanation these attributes and the values they can take, see Chapter 3, "States".

CMS System Model

The CMS System Model includes, in addition to the FRU and service objects, device objects and port objects. There is a one-to-one mapping between the CMS device objects and the Solaris device instances that are managed by the CMS.

The CMS System Model contains:

- A Device Hierarchy Tree that represents the Solaris Device Tree
- A Device Containment Tree that represents the location of the physical hardware of the device in relation to the system's FRUs
- Device Hot Plug Trees, which are trees with two roots, that represent the hot plug requirements of the Solaris Device Tree

The Device Hierarchy Tree is used to define the state dependencies between the devices (for example, a child device cannot be brought online until its parent is online, and, conversely, a parent device cannot be taken offline until its child device is offline).

The Device Containment Tree is used to define which FRU device hardware failures should be reported.

The Device Hot Plug Tree is used to define the group of devices to be managed in the hot plug operation of a particular FRU.

Hidden Attributes and Constituents

The constituents and devices used to create these trees, and a number of attributes that are used by the trees to perform their functionality, are hidden from the user.

For a detailed explanation of these attributes and the values they can take, see Chapter 3, "States".

CHAPTER 3

States

The primary purpose of an object's state is to trigger its own or another object's behavior. This is achieved by changing the value of the object's attributes, or when the value of a related object's state changes. The behavior triggered can be immediate, or take place after subsequent attribute changes or after the change of state of related object(s).

A secondary role of an object's state is to consolidate the values of the object's attributes, and those of any related object's state, into a single attribute.

CMS applications make use of an object's state. For example, `cmsfix` and the CMS API make use of the `state` attribute. They determine that a device is faulty if its state is `failed` and a FRU is faulty if its `faulty` attribute is set to `yes`.

State Generation

When any attribute in the CMS is modified, the CMS regenerates the state variables of every object. Any state variable change can result in behavior being triggered appropriate to the resultant state transition.

The relationships specified in the `cmsdef` are used to determine the order in which state change events are generated by the CMS. This only affects applications making use of the CMS API and which register for asynchronous event notification.

The CMS descends the constituent hierarchy so that the state variables of all constituent objects are regenerated before the dependent objects.

This ensures that events generated by the CMS are in the desired order (that is, events are delivered for constituents, then for dependents).

In principle, this behavior also means that behavior triggered by state changes is initiated in the desired order (that is, behavior of constituents is initiated before behavior of dependents). However, since the behaviors run asynchronously with respect to the CMS, there is no guarantee that behavior initiated in a particular order, according to the constituent relationships, will actually execute in that order.

The `cmsdef` developer must consider this issue when designing the `cmsdef`s and take action to prevent any dangerous consequences that could result.

Loops must not be created in the constituent hierarchy. Loops are not detected by the CMS and will cause the CMS to fail through infinite recursion. The `cmsdef` developer must ensure that erroneous user input does not cause the CMS to fail in this way, either by use of permitted attribute values, or by appropriate rules in `cmsconfig.rule`.

State Definition

The states of an object are defined in the corresponding `cmsdef` in the `state-function` section. A given state is assigned to an object if the corresponding function evaluates to `true`. The functions are evaluated in the order specified in the `cmsdef,` from top to bottom. The state corresponding to the first function that evaluates to `true` is used.

The state of all objects is evaluated when:

- the CMS is started
- a configuration change occurs and abdication takes place
- a system event is received from the logging device
- an attribute change occurs due to a user request via `cmsconfig` or `cmsfix`

Note – If a constituent has not yet been defined, its default value will usually be `null 0`, a valid object whose state is always `offline`. When writing state functions (and elsewhere), consider the constituent's default value – system attributes are reset to default values when the CMS starts.

Relationships

Relationships between objects in the CMS are specified using constituent attributes. An object's constituent attribute contains the identifier of the related object. The related object is considered the constituent of an object.

The CMS API refers to the constituent and dependent as child and parent respectively. The CMS API extends this convention further by referring to any dependent, or dependent of a dependent, as an ancestor. Any constituent, or constituent of a constituent, is referred to as a descendent.

A single relationship scheme, having certain built-in characteristics, exists throughout the CMS. Any relationship class specific behavior must then be added to the `cmsdef` by the developer.

Note that within `cmsdef` specifications, it is generally possible to identify the *constituents* of an object, but not possible to identify the *dependents* of an object. An indirect mechanism is available for traversing back through the relationship, but this cannot be used in the state definition. It is not permitted to set up circular relationships in the form B is a constituent of A, C is a constituent of B, and A is a constituent of C.

Attributes

This section describes the attributes used in the device state model and the FRU state model.

System Attributes are those attributes of the object that the device driver can report, for example, `_driver_state`, and the conditions of the module, that is, undefined, `offline`, `online`, `degraded`, `failed`. They cannot be set using `cmsconfig` and are reset to their default values when the CMS is started.

User Attributes are those that a user can change using `cmsconfig`. Their values persist across a CMS restart (that is, after a reboot). Attributes are displayed by `cmsconfig` in the order in which they are defined in the `cmsdef`. Thus it is important to make sure that the standard user attributes listed below are defined in the `cmsdef` in the order given, and before any other user attributes.

Hidden Attributes can be either system or user attributes, and are defined by the first character being '_'. These attributes are hidden from the users of `cmsconfig`. Attributes should be hidden if their display might confuse the user or when the user should be prevented from setting them.

FRU State Model

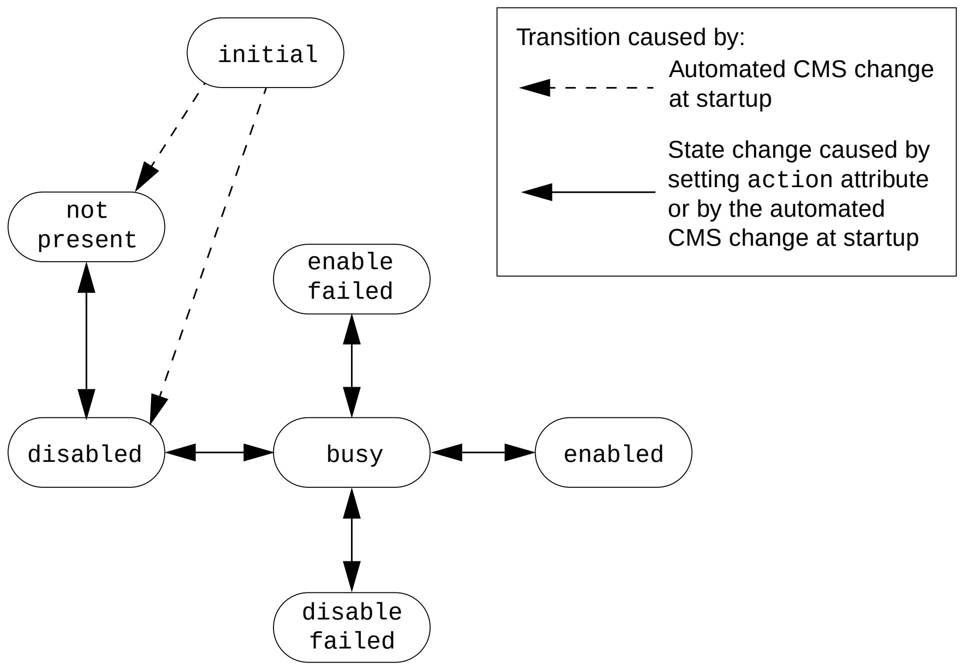
initial
not present
enable failed
disabled
busy
enabled
disable failed
Transition caused by:
Automated CMS change at startup
State change caused by setting action attribute or by the automated CMS change at startup

FIGURE 3-1 The CMS FRU State Model

FRU States

A FRU object can have the following states.

- `initial`

  The state of the FRU changes to `initial` at start up.

  The FRU will remain in this state until its `_initialised` attribute is set to `yes`.

- `not_present`

  A FRU is in this state when its `location` attribute has not been set. This means that the CMS is not aware of the FRU. A FRU that is in the `not_present` state will have no relationships with any other FRUs, devices or the `ft_core` object.

- `disabled`

  A FRU is in this state when the CMS believes that the FRU exists in a particular location, but the CMS is not currently managing the FRU. The FRU is configured but has not been enabled.

- `enabled`

  The FRU is in this state when the CMS believes that the FRU exists in a particular location and the CMS has attempted to enable the FRU for use.

- `enabled_failed`

  A FRU is in this state when the CMS believes that the FRU exists in a particular location, but the CMS has been unable to enable it for use. This could mean that the CMS has failed to read the FRU's EEPROM (perhaps because the FRU was not present at the location) or the FRU's EEPROM failed the validation for a FRU of that type. It could also mean that the CMS failed to apply power to the location.

- `disable_failed`

  The FRU is in this state when the CMS believes that the FRU exists in a particular location that has previously been `enabled`, but the CMS has been unable to disable the FRU. This is generally due to the CMS being unable to take offline some of the FRU's devices. In this state, the FRU will still be powered up.

- `busy`

  The FRU is in this state when the CMS in performing some action on the FRU.

FRU Relationships

The following relationship exists between the FRU object and the devices that reside in the FRU.

- `device_x`

  The attributes can either be fixed per class of FRU or read from the EEPROM when the user attempts to configure the FRU. The attribute can have one of the following values:

  ```
  null 0
  device instances
  ```

Generic FRU Hidden Attributes

The FRU object can have the following system attributes.

- `_show_by_default`

  This attribute defines whether a FRU should appear on the default `cmsconfig` menu. It has two permitted values:

  `no`
  `yes`

- `_initialised`

  This attribute is referenced at CMS start up. It has two permitted values:

  `no`
  `yes`

  If set to `no`, the FRU state is `initial`

- `_locked`

  This attribute is used to define when a FRU is performing an action. It has two permitted values:

  `no`
  `yes`

  If set to `yes`, the FRU state is `busy`.

- `_action`

  This attribute is used in state definitions to signify the last required state. It has two permitted values:

  `disable`
  `enable`

- `_enabled`

  This attribute is used in state definitions to signify when the FRU is enabled. It has two permitted values:

  `no`
  `yes`

- `_device_failed`

  This attribute is used by device objects to indicate that they have gone to the failed state. It takes a string value that returns the device name.

- `_device_soft_fault`

  This attribute is used by device objects to indicate that they have not managed to go `online` as a result of a software fault. It takes a string value.
- `_device_state`

  This attribute is used by device objects to indicate that they have changed state. It takes a string value.
- `_monitoring`

  This attribute is used to start or stop environmental monitoring of the FRU. It has one permitted value:

  `yes`

  If the attribute is not present or is not set to `yes`, no environmental monitoring is set up.
- `_power_control`

  This attribute is used to control the power to the FRU. It has one permitted value:

  `yes`

  If the attribute is not present or is not set to `yes`, power control is not applied to the FRU.
- `_device_del_loop`

  This attribute is used by the FRU to take its child devices offline sequentially. It has two permitted values:

  `disable`
  `enable`
- `_power_control_delay`

  This attribute is used to define a time delay between supplying power to a FRU and accessing it to allow time for firmware to be downloaded. It has two permitted values:

  `disable`
  `enable`

Generic FRU Visible Attributes

The FRU object can have the following user attributes.

- `action`

  `action` is the operation attribute enables the user to enable and disable the FRU, and preserves the user's desired FRU configuration following a reboot. It has two permitted values:

  `disable`
  `enable`

- `location`

  `location` is used by the CMS to specify the physical location (that is, slot name) of the FRU. The CMS uses the location as the origin for identifying objects considered to exist. All objects with a non-NULL location and their ancestors (that is, dependents and dependents of dependents) are considered to exist. If a `location` value is not set, the FRU is considered not to exist.

- `fault_acknowledged`

  The `fault_acknowledged` attribute is set when a user acknowledges that a fault has been seen on a particular FRU. It has two permitted values:

  `no`
  `yes`

- `description`

  This string attribute gives the user a test description of the class of device (maximum 33 characters). The attribute is not changed by `cmsconfig`.

- `user_label`

  This string attribute enables a user to set a test description for each instance of the FRU (maximum 33 characters).

- `part_number`

  This string attribute is set up the first time the FRU is enabled and is read from the EEPROM. Subsequently, each time the FRU is enabled, the value of the attribute is compared with the value in the EEPROM. If the values do not match, the FRU is not enabled. This function can be overridden by setting the value to ' '.

- `serial_number`

  This read-only string attribute is set up the first time the FRU is enabled and is read from the EEPROM.

- `software_fault`

  This read-only attribute is used to signify that a software fault has occurred. It has two permitted values:

  ```
  no
  yes
  ```

- `present`

  This read-only attribute is used to signify that the FRU cannot be accessed. The attribute is set when the FRU is enabled and its EEPROM has been read. The attribute can be set by the environmental monitor if it finds that the FRU is no longer present.

  It has two permitted values:

  ```
  no
  yes
  ```

Device State Model

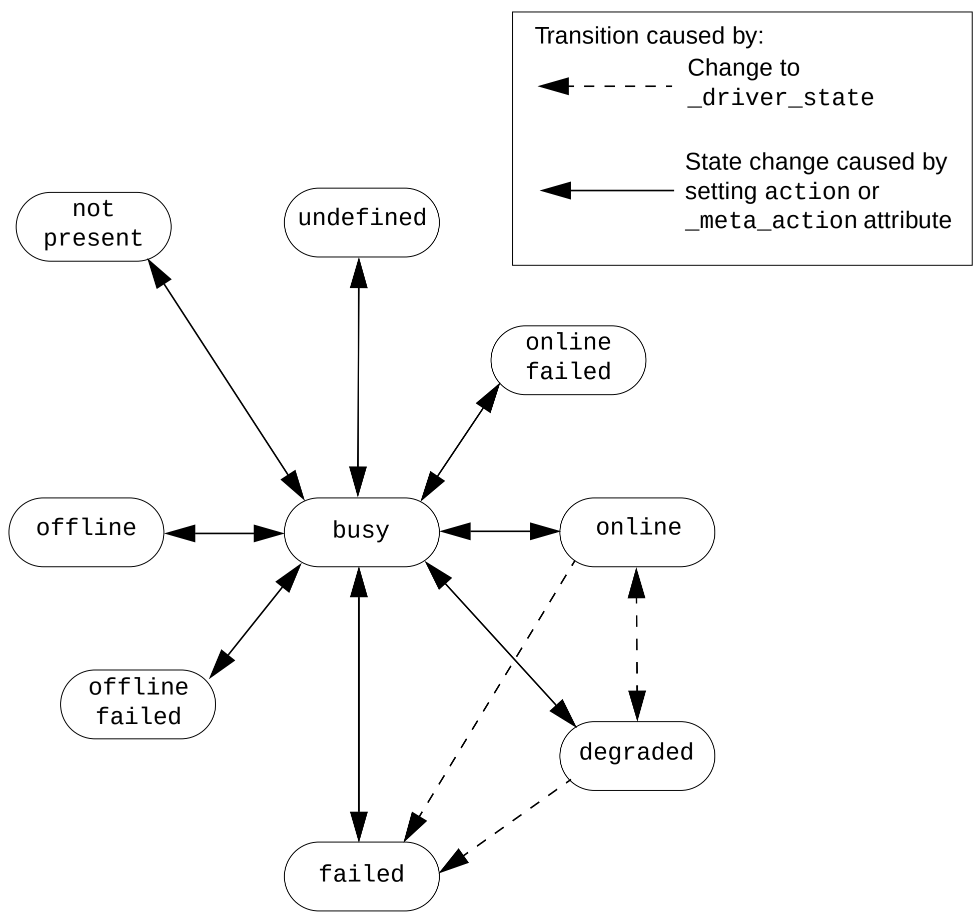

FIGURE 3-2 The Device State Model

A CMS device object is notified of its state by the driver of the device that the object is representing. The device driver implements the Nexus device driver framework. This specifies the permitted state transitions.

Device States

A CMS `device` object can have the following states.

- `undefined`

  A device is in this state when either the CMS has failed to tag or create the device, or when the CMS has deleted the device but has been unable to delete its parent.

- `offline`

  A device is in this state when its driver is offline but its FRU is enabled. This can occur only when a super-user is using the CMS to configure the device's `action` or `required_state` attributes manually.

- `online_failed`

  A device is in this state when its driver state is `offline` and the FRU has attempted to bring the device `online` but the request failed.

- `online`

  A device is in this state when its driver state is `online`.

- `degraded`

  A device is in this state when its driver state is `degraded`.

- `failed`

  A device is in this state when its driver state is `failed`.

- `offline_failed`

  A device is in this state when its driver state is not `offline` and its FRU has attempted to take the device `offline` but the request failed.

Device Relationships

The following relationships exist between the device and other software abstractions of Netra ft components.

- `controller`

  If the device requires a controller, this is the CMS object instance for that device. The attribute is set up by the FRU when appropriate and can have one of the following values

  ```
  null 0
  device instances
  ```

- `bridge`

  If the device operates under a bridge, this is the CMS object instance for that device. The attribute is set up by the FRU when appropriate and can have one of the following values:

  ```
  null 0
  bridge instances
  ```

- `location^`

  This is a reverse direction mapping of the FRU that has this device as a constituent. The attribute value is therefore a `FRU instance`

Generic Attributes of a Device Object

A CMS device object can have the following attributes. Where permitted values are specified, the first to be listed is the default value.

- `_driver_state`

  The `_driver_state` attribute is set by `cmsd` when a message of class `state` is received from the logging device `u4ftlog` Its value is used to derive the state of the device. It can have one of the following values:

  ```
  undefined
  offline
  online
  failed
  degraded
  ```

- `_present`

  `_present` is a hidden attribute used to define the existence of the device. This attribute is set by the FRU and is used to derive the state of the device. It has two permitted values:

  ```
  no
  yes
  ```

- `FRU_name`

  `FRU_name` is a string attribute that identifies the name of the FRU in which the device resides. The attribute is set by the device when it leaves the `not_present` state.

- `info`

  The `info` attribute is a string that informs the user about the status of the device.

- `_locked`

  The `locked` attribute informs the user that the device is locked and will not accept any requests to change its attributes. It has two permitted values:

  `no`
  `yes`

  When the attribute is set to `yes`, the device state is `busy`.

- `action`

  `action` enables the super-user to manipulate the device directly when the value is set to `offline`. It has two permitted values:

  `offline`
  `online`

- `_meta_action`

  This attribute is used by the FRU to create a device and cause it to take its `required_state`, or to delete a device. It can have one of the following values:

  `null`
  `create`
  `delete`

- `required_state`

  This attribute enables the super-user to specify whether the device should be brought `online` when the FRU is enabled. It has two permitted values:

  `online`
  `offline`

- `description`

  This string attribute enables the user to specify a test description for the class of device.

- `user_label`

  This string attribute enables the user to set a test description for each instance of the device.

- `_device_name`

  This string attribute contains the name of the device driver if the CMS creates the device instance.

- `_parent_state`

  This attribute is set by `cmsd` when a message of class state is received from `u4ftlog`. Its value is used to derive the state of the device.

  The attribute can have one of the following values:

  `not_present`
  `busy`
  `undefined`
  `online_failed`
  `online`
  `offline`
  `offline_failed`
  `inaccessible`
  `degraded`
  `failed`

- `devpath`

  This string attribute contains the system name for the device. The attribute value is passed to the FRU to identify the device's functionality.

CODE EXAMPLE 3-1 System Attributes

```
# system attribute      values
# ----------------      ------
_driver_state           undefined, offline, online, degraded, failed
```

Note – Values appear in comma delimited lists, where the first value is the default. The default value is generally negative (for example, `offline` rather than `online`, `null` rather than `not null`). This ensures that objects are *inactive* by default and have to be deliberately brought into the *active* state by the CMS. System attributes are reset to default values when the CMS restarts.

Constraints

The following `cmsconfig` restrictions apply to device objects:

- If the state equals `not_present`, you can change only the `_locked` attribute.
- If `_busylock` equals `yes`, a warning is issued if you try to change any attribute.
- You cannot change the description.
- You cannot change the controller.
- You cannot change the bridge.
- If `_locked` equals `yes`, you cannot change any attribute.

Built-in States

In addition to the states specified in the `state-function` section of the `cmsdef`, two further states are built-in to the CMS:

- `unknown`

  The `unknown` state is assigned to the object at CMS startup. Once the previous values of attributes are recovered from disk, the CMS changes from the `unknown` state to the new state which is based on the recovered attribute values and the state functions specified in the `cmsdef`.

- `abdicate`

  The `abdicate` state is assigned to a new CMS when abdication occurs. Once the existing values of attributes are recovered from the abdicating CMS, the CMS changes from the `abdicate` state to the new state which is based on the recovered attribute values and the state functions specified in the `cmsdef`.

Although it is possible to specify transitions from the `unknown` or `abdicate` states in the `transition-response` section of the `cmsdef`, do so only when no other solution is possible. This is because, at CMS startup and abdication, every single object will be transitioning from the `unknown` or `abdicate` state which could result in the execution of a large number of response scripts. This could seriously impact system performance. If transitions are specified in the `transition-response` section, limit the number of response scripts that will execute at CMS startup or abdication.

Abdication

Abdication is a mechanism provided by `cmsd` that enables a hot upgrade to be made to the configuration of the system held by the CMS. On abdication, a new instance of `cmsd` is started using a new set of system definition files. The new instance of `cmsd` next obtains the actual system configuration from the old instance of `cmsd` which it then supersedes.

State Transition

The behavior triggered by state transitions is defined in the `cmsdef`. The behavior is specified in the form of shell commands with some extensions to enable the attribute values of that `cmsdef` to be available to the shell commands. This is achieved by a macro expansion which is performed once the behavior is triggered.

For example:

```
$state
$req_condition
$location
```

The shell commands are executed in a separate process that executes synchronously with respect to the CMS. It is possible for behaviors triggered by multiple and rapid state changes of a single object to run concurrently with no guarantee of the order of execution or that the commands will complete. The `cmsdef` developer must consider this issue when designing `cmsdef`s and provide locking to prevent two scripts from running concurrently.

Behaviors must also test the exit codes of the executed commands and take recovery action of the command fails. For example, this is necessary when taking a device offline as there could be a process holding the device open.

Behaviors that ignore the failure of the `offline` command could cause power to be removed from the FRU which would result in failure messages appearing in the log and possible software failure elsewhere. Error recovery must be built into the behavior to ensure that this does not happen.

Behavior on Attribute Changes

Shell commands are issued when attribute changes occur if they have been specified in the `cmsdef`. Each attributed of the object class is available as a macro, using the same mechanism as a conventional `cmsdef` file.

As before, behavior triggered by an attribute value change runs asynchronous with respect to the CMS and there is no guarantee that behavior will execute in order with respect to any other behavior, even that triggered by a subsequent change to the same attribute value.

The `cmsdef` developer must consider this issue when designing `cmsdef`s and take action to prevent any dangerous consequences that could result.

Take care where attributes involved in the state function also have behavior triggered on changes to the attribute. It is likely that this will result in two scripts being initiated – one for the state transition and one for the attribute change. Again, there is no guarantee as to the order in which these scripts will be executed.

Note – Surround all environment variables provided to the script by the CMS with double quotes. This is because the environment variables could comprise more than one word.

Changing Attribute Values

There are three sources of changes to attribute values within the CMS:

- System events received from the logging device
- CMS events received from the logging device
- User events received from the logging device

System Events

System events are received from the logging device. Typically, system events correspond to device state changes and appear in the log as 'S' class messages. Device state changes are mapped within the CMS to changes to the object's `condition` attribute.

Alternatively, device property updates can be sent to the CMS by the device. Device property updates are mapped within the CMS to changes to the object's attribute whose name is identical to the device property. If there is no such attribute, the property update is ignored. The CMS attempts to restrain its response to device property updates by counting and discarding updates for a device property while it is attempting to acquire the value for that device property. When the property value is acquired, the count of property updates received for that device property is reported to the log. This mechanism is intended to prevent the CMS from being saturated by property updates.

System events are not subject to validation according to `cmsconfig.rule` and are therefore always actioned.

CMS Events

CMS events are received from the logging device and are used to notify the CMS of system attribute changes for objects that do not have a corresponding device driver. Typically, CMS events are generated by the use of `u4ftctl`(1M) from within `cmsdef` responses.

CMS events appear in the log in the following format:

```
[time] <u4ftcmd#0> C <level> <source> <module> <instance> <attribute> <value>
[message]
```

- <*level*> is not currently used by the CMS and should be set to 0.
- <*source*> is the software object originating the CMS event (for example, `pmft`).
- <*module*> <*instance*> is the module and instance of the CMS object (for example, `ioset 0`).
- <*attribute*> is the attribute name.
- <*value*> is the attribute value.

CMS events are not subject to validation according to `cmsconfig.rule` and are therefore always actioned.

User Events

User requests are received from `cmsconfig`, `cmsfix`, `xcmsfix` and other CMS applications. User requests modify attributes specified as user attributes in the `cmsdef`. User requests are always validated according to `cmsconfig.rule` and are actioned if the consequence of the request does not violate a rule.

Validation of User Events

The CMS validates user requests by assessing the consequence of the change according to `cmsconfig.rule`. This is achieved by identifying attribute and state changes that would occur if the change were actioned. The CMS does this by duplicating itself, executing the request in the duplicate, but not actually executing any scripts as a result of attribute and state changes that are caused by the request.

The resultant attribute and state changes are sent back to the requesting application by the duplicated CMS. The application is then responsible for determining if `cmsconfig.rule` is violated.

- If `cmsconfig.rule` is violated, the application is responsible for abandoning the user request and issuing suitable error messages to the user.
- If `cmsconfig.rule` is not violated, the application issues the real request and the CMS actions it.

For the `cmsconfig.rule` mechanism to work effectively, the state of dependent objects must be partially or entirely derived from related objects. For example, devices that would become failed if power to the related FRU were removed must have their state directly related to the FRU in some way.

The application does not consider the object whose attribute is being changed when evaluating `cmsconfig.rule`. This is because it is assumed that the user must be aware of the consequences of a change to that object.

APPENDIX **A**

Sample cmsdefs

This appendix provides sample cmsdefs for imaginary device called flop and flip.

flop has four instances and can have up to two child devices of the type flip.

CODE EXAMPLE A-1 cmsdef for Device flop

```
#
# %W%   %E% SMI
#
# Copyright (c) 1998, by Sun Microsystems, Inc.
# All rights reserved.
#
# slot cmsdef file
#

module          max_number_in_system
------          --------------------
flop            4

     user_attribute      values
     --------------      ------
     action             offline, online        startup=norestore
     required_state      online, offline
     description         "flop device"
     user_label          string
     _locked            no, yes                startup=norestore

     system_attribute    values
     ----------------    ------
     FRU_name            string
     info                string
     busylock            no, yes
     _meta_action        null, create, delete
```

CODE EXAMPLE A-1 `cmsdef` for Device `flop` *(Continued)*

```
#
     _action             offline, online, report_state
startup=norestore
     _driver_state       undefined, offline, online, degraded,
failed
     _present            no, yes
     _parent_state       not_present, busy, undefined,\
                         online_failed, online, offline\
                         offline_failed, inaccessible,  \
                         degraded,failed
     _device_name        "u4flop"
     _type_flop_device   yes
     _bsh                yes
     _no_delete          yes

     constituent         values
     -----------         ------
     dev0               null 0, flip 0, flip 1, flip 2, flip 3,

     dev1                null 0, flip 4, flip 5, flip 6, flip 7

     state               function
     -----               --------
     not_present         if (_present == no)
     busy                if (_locked == yes)
     undefined           if (_driver_state == undefined)
     online_failed      if (_driver_state == offline && _action\
                            == online )
     online             if (_driver_state == online && _action\
                            != offline)
     offline            if (_driver_state == offline && _action\
                            != online)
     offline_failed     if (_driver_state != offline && _action\
                            == offline)
     inaccessible        if (_parent_state != online && \
                              _driver_state == online)
     degraded            if (_driver_state == degraded)
     failed              if (_driver_state == failed)

    class           response
    -----           --------

    generic_script \
        function _cms_my_report_device_state { \
                cmsmreport $name $number _parent_state $state \
                        $dev0->name $dev0->number \
                        $dev1->name $dev1->number; \
```

CODE EXAMPLE A-1 `cmsdef` for Device `flop` *(Continued)*

```
#
        };\
        function _cms_my_create_device { \
             : add your own create device script if appropriate ;\
                : return 0 for success, 1 otherwise ;\
                return 1;\
        };\
        function _cms_my_tag_device { \
               : add your own tag device script if appropriate ;\
                : return 0 for success, 1 otherwise
                return 1; \
        } ;\
        function _cms_my_report_device_failed { \
                _cms_report_device_failed $location^location \
                             "$location^name $location^number";\
                esac;\
        };

    transition      response
    ----------      --------

    not_present -> busy \
        _cms_report $name $number FRU_name "$location^name
$location^number" ;

    busy -> undefined \
        _cms_my_report_device_state ;

    busy -> online_failed \
        _cms_my_report_device_state ;

    busy -> online \
        _cms_my_report_device_state ;

    busy -> degraded \
        _cms_my_report_device_state ;

    busy -> failed \
        _cms_my_report_device_state ;\
        _cms_my_report_device_failed;

    busy -> offline \
        _cms_my_report_device_state ;

    busy -> inaccessible \
        _cms_my_report_device_state ;
```

CODE EXAMPLE A-1 `cmsdef` for Device `flop` *(Continued)*

```
#
    busy -> offline_failed \
        _cms_my_report_device_state ;

    online -> failed \
        _cms_my_report_device_state ;\
        _cms_my_report_device_failed;

    degraded -> failed \
        _cms_my_report_device_state ;\
        _cms_my_report_device_failed;

    offline_failed -> online \
        _cms_my_report_device_state ;

    online_failed -> offline \
        _cms_my_report_device_state ;

    busy -> not_present \
        _cms_my_report_device_state ;

    attribute        response
    ---------        --------

    action \
        my_parent_name=$_type_gate^name;\
        my_parent_number=$_type_gate^number; \
        _cms_device_action $$my_parent_name $$my_parent_number;

    _parent_state \
        _cms_device_parent_state ;

    _meta_action \
        my_parent_name=$_type_gate^name;\
        my_parent_number=$_type_gate^number; \
        _cms_device_meta_action $$my_parent_name
$$my_parent_number;

    _driver_state \
        my_parent_name=$_type_gate^name;\
        my_parent_number=$_type_gate^number; \
        _cms_device_driver_state $$my_parent_name
$$my_parent_number;

    _action \
        _cms_device__action "$location^name $location^number";
```

The following points apply to the device `flop`:

- The device does not bring its parent `online` or take it `offline`.
- Its parent has an attribute called `type_gate`.
- The device is not deleted when a FRU is disabled.
- The device is brought `online` and taken `offline` by its children and, therefore, does not report its state to a FRU, but to its children.
- The device reports failures to the FRU which is its container and which has an attribute `location`.

The following code example is the `cmsdef` for the corresponding child device `flip`.

CODE EXAMPLE A-2 `cmsdef` for Child Device `flip`

```
# %W%   %E% SMI
#
# Copyright (c) 1998, by Sun Microsystems, Inc.
# All rights reserved.
#
# flip cmsdef file
#

module          max_number_in_system
------          --------------------
flip            8

     user_attribute      values
     --------------      ------
     action             offline, online       startup=norestore
     required_state      online, offline
     description         "flip device"
     user_label          string
     _locked            no, yes               startup=norestore

     system_attribute    values
     ----------------    ------
     FRU_name            string
     FRU_info            string
     info                string
     busylock            no, yes
     _meta_action        null, create, delete
     _action             offline, online, report_state
startup=norestore
     _driver_state       undefined, offline, online, degraded,
failed
     _device_name        "sd"
     _present            no, yes
     _parent_state       not_present, busy, undefined,
online_failed, online, \
```

CODE EXAMPLE A-2 `cmsdef` for Child Device `flip` *(Continued)*

```
# %W%   %E% SMI
                         offline, offline_failed, inaccessible,\
                         degraded, \
                         failed

     state               function
     -----               --------
     not_present         if (_present == no)
     busy                if (_locked == yes)
     undefined           if (_driver_state == undefined)
     online_failed      if (_driver_state == offline && _action\
                            == online )
     online              if (_driver_state == online && _action\
                            != offline)
     offline            if (_driver_state == offline && _action\
                            != online)
     offline_failed     if (_driver_state != offline && _action\
                            == offline)
     inaccessible        if (_parent_state != online && \
                            _driver_state == online)
     degraded            if (_driver_state == degraded)
     failed              if (_driver_state == failed)

    class           response
    -----           --------

    generic_script \
        function _cms_my_report_device_state { \
                _cms_report_device_state $location^name
$location^number; \
        }; \
        function _cms_my_create_device { \
             : add your own create device script if appropriate ;\
                : return 0 for success, 1 otherwise ;\
                return 1;\
        };\
        function _cms_my_tag_device { \
               : add your own tag device script if appropriate ;\
                : return 0 for success, 1 otherwise
                return 1; \
        } ;\
        function _cms_my_report_device_failed { \
                _cms_report_device_failed $location^location \
                        "$location^name $location^number"; \
        };\
        function _cms_my_post_online_function { \
                u4ft_findPath $name $number "$tag"; \
```

CODE EXAMPLE A-2 `cmsdef` for Child Device `flip` *(Continued)*

```
# %W%   %E% SMI
        };

    transition      response
    ----------      --------

    not_present -> busy \
        _cms_report $name $number FRU_name "$location^name \
        $location^number" ;

    busy -> undefined \
        _cms_my_report_device_state ;

    busy -> online_failed \
        _cms_my_report_device_state ;

    busy -> online \
        _cms_my_report_device_state ;

    busy -> degraded \
        _cms_my_report_device_state ;

    busy -> failed \
        _cms_my_report_device_state ;\
        _cms_my_report_device_failed;

    busy -> offline \
        _cms_my_report_device_state ;

    busy -> inaccessible \
        _cms_my_report_device_state ;

    busy -> offline_failed \
        _cms_my_report_device_state ;

    online -> failed \
        _cms_my_report_device_state ;\
        _cms_my_report_device_failed;

    degraded -> failed \
        _cms_my_report_device_state ;\
        _cms_my_report_device_failed;

    offline_failed -> online \
        _cms_my_report_device_state ;

    online_failed -> offline \
```

CODE EXAMPLE A-2 `cmsdef` for Child Device `flip` *(Continued)*

```
# %W%   %E% SMI
        _cms_my_report_device_state ;

    busy -> not_present \
        _cms_my_report_device_state ;


    attribute        response
    ---------        --------

    action \
        my_parent_name=$_type_flop_device^name; \
        my_parent_number=$_type_flop_device^number; \
        _cms_device_action $$my_parent_name $$my_parent_number;

    _parent_state \
        _cms_device_parent_state ;

    _meta_action \
        my_parent_name=$_type_flop_devicer^name; \
        my_parent_number=$_type_flop_device^number; \
        _cms_device_meta_action $$my_parent_name
$$my_parent_number;

    _driver_state \
        my_parent_name=$_type_flop_device^name; \
        my_parent_number=$_type_flop_device^number; \
        _cms_device_driver_state $$my_parent_name
$$my_parent_number;

    _action \
        _cms_device__action "$location^name $location^number";

    FRU_info \
        [[ "$location^_dev1" = "$name $number" ]] && \
            _cms_report $location^name $location^number Funct_0\
                  "$FRU_info";\
        [[ "$location^_dev2" = "$name $number" ]] && \
            _cms_report $location^name $location^number Funct_1\
                  "$FRU_info";
```

The following points apply to the device `flip`:

- The device brings its parent `online` and takes it `offline`.
- Its parent has an attribute called `_type_flop_device`.
- The device is deleted when a FRU is disabled.
- The device reports its state to a FRU. As it has no children, it also reports its failure to the FRU.
- The FRU has a system attribute for the device as part of its user interface.

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^{2}C	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