Sun™ Management Center 2.1 Developer Environment Reference Manual

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. 806-3167-11
December 1999, Revision A

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

NOTICE: USE OF THIS SUN (TM) MANAGEMENT CENTER 2.1 DEVELOPER ENVIRONMENT REFERENCE MANUAL AND ALL OTHER RELATED DOCUMENTATION AND SOFTWARE UTILITIES REQUIRES THAT THE USER OBTAIN A SUN MANAGEMENT CENTER 2.1 DEVELOPER ENVIRONMENT RIGHT TO USE LICENSE FROM SUN MICROSYSTEMS, INC. THE STANDARD BINARY CODE LICENSE FOR USE OF THE SUN (TM) MANAGEMENT CENTER 2.1 SOFTWARE IN A PRODUCTION ENVIRONMENT PROVIDES NO RIGHTS TO USE THESE MATERIALS FOR DEVELOPMENT PURPOSES.

Contents

Figures

Tables

Procedures and Examples

Preface

The *Sun Management Center 2.1 Developer Environment Reference Manual* provides instructions on how to use the Sun Management Center™ development environment. These instructions are designed for programmers with knowledge of object-oriented programming languages.

Audience

The audience of this document are programmers who already have a knowledge of object-oriented language and Java. This document does not explain object-oriented fundamentals. Moreover, this document does not explain some concepts in great detail since the assumption is that the document is for programmers already familiar with them.

The audience is one who is exposed to the Sun Management Center product. Hence, many terms and concepts applicable to the product are not explained here. For more information on those, refer to the *Sun Management Center 2.1 User's Guide*. Third-party clients, such as application programmers and system administrators, should note that the code examples in this document are mainly presented here for reference.

Contents in this Manual

Refer to Chapter 3, which provides a quick preview of the contents and information contained in this document.

Access to Up-to-date Information on the Developer Environment

Once the product image is installed, refer to:

`/opt/SUNWsymon/sdk/docs/index.html` file

for the most up-to-date information on where the following files reside:

- Developer Environment code examples
- Client API classes in the `Javadocs`

Using UNIX Commands

This document does not contain information on basic UNIX® commands and procedures, such as shutting down the system, booting the system, and configuring devices. See one or more of the following for this information:

- *Solaris Handbook for Sun Peripherals*
- AnswerBook™ online documentation for the Solaris™ software environment
- Other software documentation that you received with your system

Shell Prompts

TABLE P-1 Shell Prompts

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

Typographic Conventions

TABLE P-2 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:`
`<abc>` or *<abc>*	These are both acceptable formats that define variables. They are only pertinent to this document and do not denote Sun's standard usage.	`<abc>` *<abc>*
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.*

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
```

Related Documentation

TABLE P-3 Related Documentation

Product Family	Title	Part Number
Enterprise Servers	The following are related documents for Enterprise Servers:	Respective Part #s:
	1. *Sun Management Center 2.1 Software User's Guide*	1. 806-3166-10
	2. *Sun Management Center 2.1 Software Release Notes*	2. 806-3168-10
	3. *Sun Management Center 2.1 Software CD Installation Guide*	3. 804-6849-10
	4. *Sun Management Center 2.1 Developer Environment Release Notes*	4. 806-3169-10
Midrange Servers	*Sun Management Center Supplement for Sun Enterprise Midrange Servers*	806-0649
Workgroup Servers	*Sun Management Center Supplement for Workgroup Servers*	806-1183
Workstations	*Sun Management Center Supplement for Workstations*	806-1184
Microsoft Documentation	*Windows Interface Guidelines for Software Design*	Check with Microsoft, Inc.
Third-Party Documentation	*Tcl and the Tk Toolkit* by John K. Ousterhout	Addison-Wesley, 1994

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.

PART I Introduction to Developer Environment

This volume includes the following sections:

CHAPTER 1

Sun Management Center and the Developer Environment

This chapter covers the following topics:

Note – This document also contains a lot of examples. The examples provided in this document are purely for reference. After you install the Sun Management Center Developer Environment, you can find key examples in the following directory: `/opt/SUNWsymon/sdk/examples/doc_samples`.

Sun Management Center Framework

Sun Management Center is an open, extensible, and standards-based server management solution that facilitates enterprise-wide management of Sun server products and their subsystems or components.

The Sun Management Center framework is comprised of the console layer, the server layer, and the agent layer. The major functionality of each layer is described in the following sections:

- Sun Management Center Console
- Sun Management Center Server
- Sun Management Center Agent

The components that comprise the Sun Management Center product are shown in the following illustration:

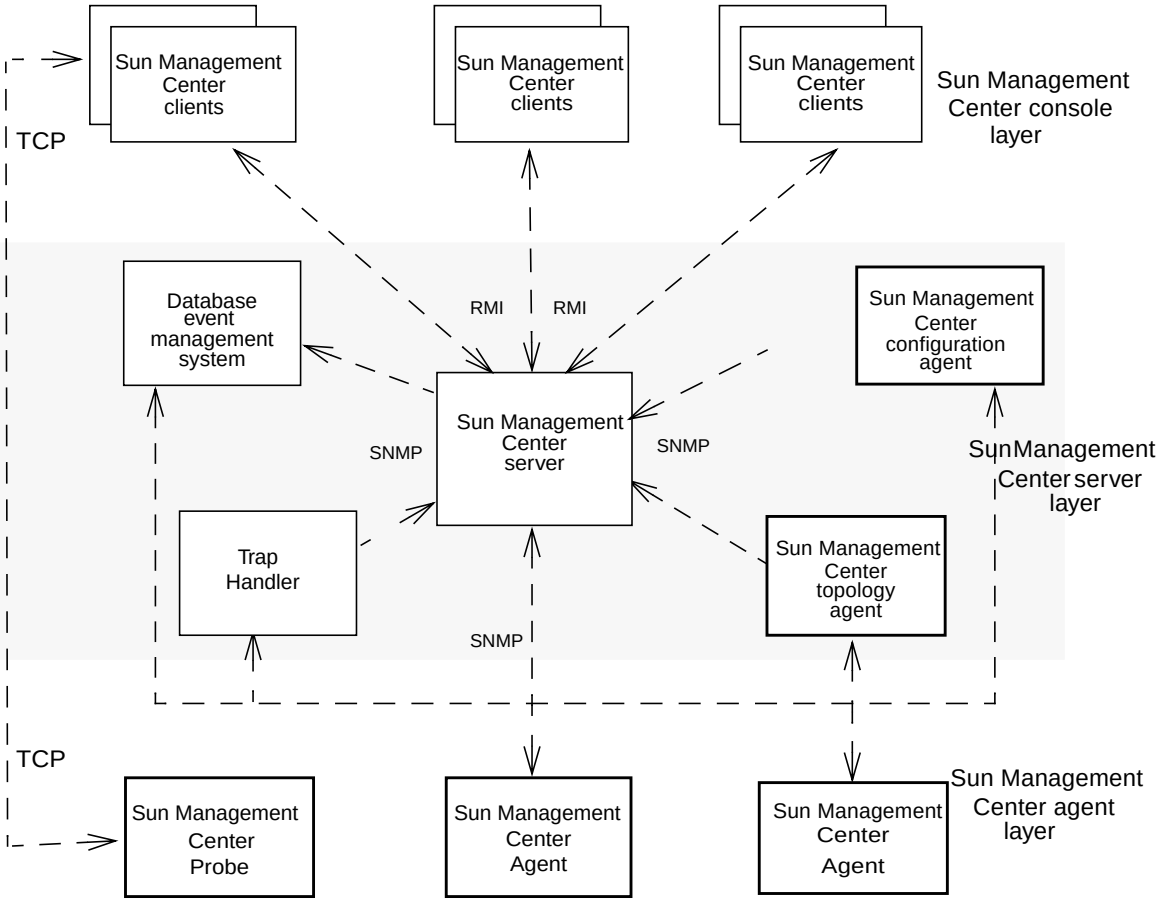

FIGURE 1-1 Sun Management Center Components

The above figure and its components are described in the following sections.

Sun Management Center Console

The console and its associated Graphical User Interface (GUI) clients are the principle means for the user to interact with the Sun Management Center software product and to accomplish management tasks. The console interacts with the Sun Management Center server to get the data, configuration and image files necessary to present the views of the managed system(s).

Sun Management Center Server

The Sun Management Center Java server acts as a request broker between the agent and the console. The Sun Management Center server layer consists of a Java server and Sun Management Center server helper agents that handle specialized tasks like topology management, event management, configuration management, and trap handling. For more information on the Sun Management Center server, refer to the *Sun Management Center 2.1 Software User's Guide*.

Sun Management Center Agent

The Sun Management Center agent consists of a set of building blocks for system management that are called modules. Sun Management Center agents are dynamically configurable, intelligent, and autonomous. Sun Management Center agents in the Sun Management Center agent layer run on managed nodes to monitor and manage entities, such as hardware and operating systems, applications, other Sun Management Center agents, and legacy SNMP agents. Sun Management Center agents also support ad-hoc probe requests from other Sun Management Center components.

Sun Management Center agents can dynamically load and unload modules. They can also disable the monitoring functions of a loaded module when not required. You can perform these functions using the Sun Management Center console. Each module is capable of modeling and managing a specific set of data items.

When the Sun Management Center agent is installed, it loads a default set of modules (for example, MIB-II, standard host system monitoring, and such). You can load additional modules from the Sun Management Center console once the agent is running. Modules are automatically reloaded whenever the agent is restarted. You can also unload and disable modules from the console.

Sun Management Center Developer Environment

Sun Management Center Developer Reference Manual is the software development kit that provides Application Programming Interfaces (APIs) and tools to help users and third party developers, such as ISVs, plan, design, develop, and integrate third-party applications, tools, and customized solutions based on the framework provided by the Sun Management Center product.

Using the information in this document, you can perform the following functions:

- Install the Sun Management Center Developer Environment.
- Build modules.
- Build consoles.
- Write rules.
- Use the Client API.
- Conform to internationalization and GUI guidelines.
- Package your product.

Note – To access commands, procedures and information on the above tasks, refer to the section, “Introduction to the Reference Manual” on page 11.

CHAPTER 2

Sun Management Center Developer Environment Installation

This chapter covers the following topics:

Note – We recommend that you run the Sun Management Center 2.1 Developer Environment and Sun Management Center 2.1 Production Environment on separate servers.

Uninstalling Previous Versions of Sun Management Center Software

If you have Sun Enterprise SyMON™ 2.*x* component layers already installed on your system, the install script allows you to uninstall the previously installed packages. You can either:

- Uninstall immediately and proceed with a fresh installation, or
- Quit the current session and uninstall the necessary packages automatically by using the `es-uninst` command, and then proceed with the installation.

If you have Sun Management Center 2.1 component layers already installed on your system, the install script prompts you to manually uninstall the previously installed packages.

Refer to the *Sun Management Center 2.1 Software User's Guide* for more details.

Caution – Be aware that when you answer `y` to the uninstall prompt, your current Sun Enterprise SyMON 2.*x* or Sun Management Center 2.1 packages will be uninstalled. However, the administrative domain information, custom alarm settings, and so on are retained in the `/var/opt/SUNWsymon` directory and may be used by the new installation.

Sun Management Center Developer Environment Licensing

You must already have a valid license for the Sun Management Center Developer Environment prior to installation. For information on purchasing a license, refer to the following web site:

```
http://www.sun.com/sunmanagementcenter
```

Note – The Sun Management Center Developer Environment right-to-use license requirement is based upon the number of users per environment, whereas the license requirement for the Sun Management Center Production Environment is based upon managing or monitoring more than one object.

Installing the Sun Management Center Developer Environment From CD

For complete instructions on how to install the Sun Management Center Developer Environment from CD-ROM, refer to the *Sun Management Center 2.1 Software CD Installation Guide*. You should also review the documents listed in TABLE 2-1 before you begin the installation process.

TABLE 2-1 Related Installation Documents

Document	Description
`INSTALL.README`	An overview of both the developer environment and production enviroment installation steps, located in the following directory: `/cdrom/cdrom0/INSTALL.README`
Sun Management Center 2.1 Developer Environment Release Notes	Installation issues related to the Sun Management Center developer environment.
Sun Management Center Software Release Notes	Installation issues related to the Sun Management Center production environment.
Sun Management Center 2.1 Software User's Guide	Pre-installation requirements, details on online help installation, and options for starting the software.
`http://www.sun.com/sunmanagementcenter`	Any late-breaking news about Sun Management Center developer environment installation.

The Sun Management Center Developer Environment can be installed in any location on your system where the minimum required disk space is available. The default location for package installation is `/opt`.

Code Examples and Client API

Once the product image is installed, the most up-to-date information on where the following files reside:

- Developer Environment code examples
- Client API in the `javadocs`

will be detailed in the following HTML file:

`/opt/SUNWsymon/sdk/docs/index.html`

CHAPTER 3

Introduction to the Reference Manual

This chapter covers the following topics:

The Different Parts of this Manual

The Sun Management Center Developer Environment documentation provides information on the following general topics, divided into three parts:

- Introduction to Developer Environment—page 1

  This part provides information on module development for the Sun Management Center agents.

- Programmer's Reference to Console Integration and Client API—page 239

  This part presents information on console bean integration with the Sun Management Center console. It also provides the Client API reference material.

- Additional Material—page 309

  This part presents tools and utilities for module developers, bean developers, and others. It also includes troubleshooting and a set of appendices with related information.

A Glossary and an Index are included.

This document also contains a lot of examples. The examples provided in this document are provided purely for reference. The client API examples will be placed in a directory from which you can copy and run them for your purposes. The location of the examples directory will be provided in the following file:

```
opt/SUNWsymon/sdk/docs/index.html
```

The procedures described in the following sections allow you to work within the broad areas covered in this document. They also introduce scenarios to help you understand the usage of the Developer Environment from a functional perspective.

Two main types of information are included:

- Scenarios that include tasks that are commonly attempted by most users.
- Procedures that help step through some of the major functionality of the product.

Accessing Information in this Manual

To access the information you need, you can refer to the “Contents” on page iii or the “Index” on page 513. However, you can also review the major topics identified below and proceed to specific sections and chapters. This section includes pointers for the following major functionality:

1. **Installing Sun Management Center Developer Environment**

   See the following chapter:

   - Chapter 2

2. **Building Modules**

   See the following chapters:

   - Chapter 4
   - Chapter 5
   - Chapter 6
   - Chapter 7
   - Chapter 8
   - Chapter 9

3. **Writing Rules**

   See the following chapter:

   - Chapter 8

4. **Building Consoles**

   See the following chapter:
   - Chapter 13

5. **Using Client API**

   See the following chapter:
   - Chapter 14

6. **Conforming to Internationalization & GUI Guidelines**

   See the following chapters:
   - Chapter 15
   - Chapter 16

7. **Packaging Your Product**

   See the following chapter:
   - Chapter 17

8. **Working within the Agent Interactive Mode**

   See the following chapter:
   - Chapter 11

Note – This document also includes a troubleshooting section and several appendices.

Building Modules

This section describes the steps to build a simple module:

1. Name Module Definition Files.
2. Specify Module Parameters.
3. Create a Data Model.
4. Realize the Data Model.
5. Add Alarm Checks.
6. Install Module Files.
7. Load a Module.
8. Log Data and To Activate Debug Mode.

9. Write a Module from an existing SNMP MIB.

10. Publish an SNMP Interface.

▼ Name Module Definition Files

- **Select a unique name for the module that must be used in naming module definition files.**

▼ Specify Module Parameters

1. **Decide on the module requirements.**
2. **Specify the standard or mandatory set of parameters.**
3. **Specify any optional parameters that are required for the module.**
4. **Specify any parameters for internationalization.**

   When done, all the parameters go into the following file:

   <*module*><-subspec>-m.x

▼ Create a Data Model

1. **Identify the components and properties that must belong to the module.**
2. **Define the data model structure.**

   Create the hierarchical structure of the managed object classes and properties. For each of the nodes in the MIB tree for the module:

   i. **Add the structural primitive.**

   ii. **Assign data and alarm and rule type primitives to properties.**

   iii. **Add the node description and units.**

   iv. **Add the qualifiers for internationalization.**

   When done, place the contents into the following file:

   <*module*>-models-d.x

▼ Realize the Data Model

1. **Determine the data acquisition method to use:**
   - UNIX programs and shell scripts: <*module*>-d.flt, <*module*>-d.sh
   - Tcl/TOE Code: <*module*>-d.prc
   - C-code libraries and Tcl/TOE command extensions: pkg<*module*>.so, lib<*module*>.so
   - Binary extensions and packages.
2. **Incorporate the data model into the module framework.**
3. **Add data acquisition services.**
4. **Add operational types to the node.**
5. **Add refresh parameters.**

   When done, all the parameters go into the following file:

   <*module*>-d.x

▼ Add Alarm Checks

1. **If a property has a threshold type alarm check, define thresholds in the file:** <*module*>-d.def**. Do the following:**
   a. **Specify the alarm criteria.**
   b. **Specify alarm severity.**
   c. **Specify alarm actions.**
2. **If the property has rules:**
   a. **Determine if the rules need to have any editable threshold parameters.**
   b. **Define all the rule initialization parameters in the file:** <*module*>-ruleinit-d.x**.**

      The error messages required for the rules must be defined in the file: <*module*>-ruletext-d.x.
   c. **Create the rules. The** <*module*>-d.rul **file contains the rule logic.**
   d. **Assign the appropriate rule to the property in the** <*module*>-d.x **file using the alarmRule qualifier.**

▼ Install Module Files

- **Make sure that each of the following directories contains the respective module files listed under its directory listing:**
  - `/opt/SUNWsymon/modules/cfg`
    - `<module>-m.x`
    - `<module>-models-d.x`
    - `<module>-d.x`
    - `<module>-d.def`
    - `<module>-d.flt`
    - `<module>-d.prc`
    - `<module>-d.rul`
    - `<module>-ruleinit-d.x`
    - `<module>-ruletext-d.x`
    - `<module>-j.x`
  - `/opt/SUNWsymon/modules/sbin`
    - `<module>.sh`
  - `/opt/SUNWsymon/base/lib/sparc-sun-solaris2.(x)`
    - `lib<module>.so`
    - `pkg<module>.so`

▼ Load a Module

1. **Start all Sun Management Center components.**

   Preferably start the agent interactively. This also enables you to debug the module. For more information on starting the agent interactively, see the Chapter 11.

2. **In the Sun Management Center console, highlight the host you want to monitor with your new module loaded.**

3. **Bring up the Load Module Window. Select the module you want to load.**

   Refer to the *Sun Management Center 2.1 Software User's Guide* for more information on module loading.

▼ Log Data and To Activate Debug Mode

Currently, all debug information is logged in *circular* log files in `/var/opt/SUNWsymon/log` directory. For more troubleshooting information, such as this, refer to the Chapter 18.

- **To see the contents of these files, use the following commands:**
  - `/opt/SUNWsymon/util/bin/sparc-sun-solaris<2.x>/ccat`

    This is similar to the `cat` command in UNIX.
  - `/opt/SUNWsymon/util/bin/sparc-sun-solaris<2.x>/ctail`

    This is similar to the `tail` command in UNIX.
- **To enable a specific level of debug message to be logged:**

  a. Go to the following directory:

```
cd /var/opt/SUNWsymon/cfg
```

  b. Edit the `domain-config.x` file.

  For example, to enable logging for agent add the following lines to the agent section:

```
activeChannels = debug info error status history
defaultOutput   = "clog://localhost/../log ESAgent.log;lines=10000"
```

  This enables you to log debug, info, error, status, and history debug messages into the following file:

  `/var/opt/SUNWsymon/log/ESAgent.log`

  The log file wraps around after every 10,000 lines of entry.
- **To enable specific debugging when an agent is started interactively:**

  a. Start the agent interactively:

```
/opt/SUNWsymon/sbin/es-start -ai
```

  b. Close any existing debug level currently set:

  For example, to turn off the channel open for information level messages, use the following command:

```
ddl close info
```

c. Open a new debug channel, for example:

```
ddl open info desc:stderr
```

This command activates the info level debugging, and sends all the info level messages to `stderr`. You can also send these messages to `stdout` or to a file.

d. To enable the corresponding debug level:

```
ddl enable info
```

▼ Write a Module from an existing SNMP MIB

If you want to write a module for an SNMP MIB, do the following:

1. **Use MIB2x to generate the module configuration files.**
2. **Update the module configuration files to implement data acquisition.**
3. **Write the data acquisition code, with one or more of the following:**
   - Tcl procedures
   - Shell scripts
   - Shared object libraries
4. **Write the rules on the data properties, if required.**

Note – This procedure is optional and, for example, is used if you want to define alarm limits on these properties.

---

5. **Install the module configuration files and other libraries/scripts/procedure files.**
6. **Load the module into the agent.**

▼ Publish an SNMP Interface

If you have some data to be modeled and monitored using Sun Management Center and want to publish an SNMP interface for this data:

1. **Prepare the data model with the following information:**
   - Data items
   - Types of each of these data items

- Groupings and the hierarchy of these data files

2. **Write a models file for the data model.**
3. **Write the data acquisition code.**
4. **Write the rules on the data properties if required.**

Note – This procedure is optional and, for example, is used if you want to want to define alarm limits on these properties.

---

5. **Install the module configuration files and other** `libraries/scripts/procedure` **files.**
6. **Start the agent in interactive mode.**
7. **Load the module into the agent.**
8. **Use** `mibExport` **to export the SNMP MIB for the module.**

Building Consoles

▼ Build Your Own Console

To build your own console to use in place of or in addition to the Sun Management Center console, do the following:

1. **Design the graphical user interface (GUI) using the Java programming language.**

   Refer to the Chapter 16 for information on how to design your GUI to be consistent with the Sun Management Center.

2. **Obtain information from Sun MC programmatically through the Client API.**

   Refer to Chapter 14 and the online Javadoc files for information on the client API.

3. **Invoke the Host Details bean to incorporate all the functionality provided in the console Host Details window.**

   Refer to the description of the Host Details bean in the section, “To Invoke the HostDetailsBean” in the Chapter 13.

Note – For detailed information on building consoles, refer to the Chapter 13.

Using Client API

▼ Use the Client API

1. **Log in to the session.**
2. **Get the SMRawDataRequest handle from the SMClientRMIImpl Class.**
3. **Use it in the constructor of other API class categories.**
4. **Start using the classes documented in the Client API section.**

   The section incudes categories of classes and each category has examples that you use for reference purposes only. You may work with the examples that are part of the code directory.

Note – For more information on building consoles, refer to Chapter 14.

Conforming to Internationalization and GUI Guidelines

▼ Work With a Java Application

1. **Create a `.properties` file for all text to be internationalized.**

   If you need more information on this, refer to the Java documentation.

2. **Import the UcInternationalizer class into your objects:**

```
import com.sun.symon.base.utility.UcInternationalizer;
```

3. **Wherever you display text that needs to be internationalized, enter UcInternationalizer.translateKey("<path to your resource is** `bundle>:<key>`**").**

   For example, to display a label that uses a string defined by the key:

```
"myKey", do
      String s;
      s =
UcInternationalizer.translateKey("myPath.myResourceBundle:myKey");
      new JLabel(s);
```

▼ Internationalize a Module

1. **Internationalize the module loader window:**

   a. **In the module parameter file (**`*-m.x`**), add two lines for each item to be internationalized. The two lines are:**

```
?param:i18n<parameter>?i18n = yes
 param:i18n<parameter> = base.modules.<module>:<key>
```

   For example, to internationalize the Fscan module name, add the following lines to `fscan-m.x`:

```
?param:i18nModuleName?i18n = yes
 param:i18nModuleName  = base.modules.fscan:moduleName
```

   b. **Add the internationalized parameters to the list of parameters to be displayed in the module load window. For each internationalized string, add** `i18n<`*parameter*`>` **to the** `ConsoleHint:moduleParms(param)` **list.**

   For example, the Fscan module parameter list would be:

```
consoleHint:moduleParams(param) = module i18nModuleName version
location enterprise i18nModuleType instance instanceName
filename scanmode
```

   c. **Create a property file. The name of the file is** `<`*module*`>.properties`**.**

d. Add an entry in the properties file for each internationalized string. The entry is of the form: `<key>=<string>`.

For example, for the Fscan module name, add the following entry:

```
moduleName=File Scanning
```

2. Internationalize the text within the module.

a. In the module models file, add the following line for each node:

```
consoleHint:mediumDesc = base.modules.<module>:<key>
```

For example, to internationalize the fileid node in the Fscan module use:

```
consoleHint:mediumDesc = base.modules.fscan:fscanstats.fileid
```

b. In the properties file created in Step c above, add the key/value for each internationalized string. This entry is the same as in Step 1 d:

```
<key>=<string>
```

c. For example, key/value pair for the `fileid` node for the Fscan module is:

```
fscanstats.fileid=File Id
```

Note – For more information on building consoles, refer to the Chapter 15 and the Chapter 16.

Integrating Applications

User applications can be integrated into the console. There are primarily two places in the console where user applications can be added; one is in the Tools menu of the console main window and the other is in the Applications tab in the host details window. Refer to the Chapter 13 for more information.

CHAPTER 4

Introduction to Modules

This chapter covers the following topics:

As discussed before the Sun Management Center agents use SNMP to communicate with the server program. The agent provides a managed set of data for the user to view in the console. The data in the agent is maintained in terms of data sets called modules. The following sections define and explain how you can load and unload modules in Sun Management Center.

Modules Definition

A module is an encapsulated set of monitoring functions that focus on a particular aspect of system or application health and performance. Typical examples include database modules as Oracle® or Sybase, operating system modules as Solaris or SunOS environments, or device modules as Hewlett-Packard printer.

Sun Management Center agents can dynamically load and unload modules. The monitoring functions of a loaded module can also be disabled when not required. These functions are performed by an end user through the Sun Management Center console.

Implementation of modules are discussed in the chapters that follow this one.

How to Load Modules

When the Sun Management Center agent is installed on a system, it is configured to load some default modules. Using the Sun Management Center console application, the user can load additional modules into the agent or unload existing modules from the agent. The agent is shipped with various modules that manage data that is diverse in nature. For the list of the modules shipped with agent, see the *Sun Management Center 2.1 User's Guide*.

Once the agent is installed, it is configured to load a default set of modules, for example, MIB-II, standard host system monitoring, and so forth. Additional modules can be loaded from the Sun Management Center console once the agent is running. By convention, modules loaded from the console are made persistent so that the modules are automatically reloaded should the agent be restarted. Modules can also be unloaded and disabled from the console.

Basic Module Building Concepts

Since Sun Management Center agents are based on TOE technology, many of the module definition files are in *module configuration file* format. Module configuration files can be thought of as ASCII configuration files. TOE and module configuration file concepts are discussed in depth in the "Agent Development" on page 365 and "TOE Objects" on page 367 sections in the Appendix A.

Types of Modules

Modules are classified into the following types:

- Hardware modules manage hardware for the host on which the agent is running, for example, boards, SIMMs.
- Operating system modules manage operating system entities for the host on which the agent is running, for example, swap, CPU usage.
- Local application modules manage entities associated with the host on which the agent is running, but which do not fall into the Hardware or Operating System module categories, for example, file scanning, process monitoring.
- Remote modules are capable of managing entities on remote hosts, for example, Sybase, Oracle, Topology, remote devices.

Module Naming

Module naming is the process of selecting a unique name for the module. This name distinguishes the module from other modules and is used in naming the module definition files.

Module Names and Subspecs

Each module must be assigned a module name and can have an optional subspec. The module name and subspec, if specified, must uniquely identify the module.

The subspec qualifier is optional. When specified, its purpose is to group together related modules.

For example, consider the Solaris operating environment management module. Solaris monitoring can be implemented as a single module. In that case, the module can be named simply *solaris* (with no subspec). Alternatively, Solaris monitoring might be implemented as a group of separate modules (one for network monitoring, one for resource loading, one for filesy stems, and so forth). Such modules can be named with subspecs. For example:

```
solaris-network
solaris-filesystems
solaris-loading
```

The simplified version of the Solaris module, which is used as an example in this document, is assigned the name *solaris-example*. The subspec *example* differentiates this module from the standard Solaris module.

There are no performance or processing considerations when deciding whether to use a subspec as part of the module name. The subspec is largely a convenience tool to assist in keeping module files organized.

Agents that contain the modules communicate to the server using SNMP.

SNMP, MIB, OID are discussed in “Parameters Specification” on page 28 in Chapter 5.

SNMP & Modules

SNMP (Simple Network Management Protocol) is the defacto standard for network based management. SNMP is simple, low bandwidth and elegant way of managing across networked entities. SNMP uses UDP (User Datagram Protocol) for communication. SNMP uses the MIB (management information base) for data modeling. The MIB defines the data organization. The data items are addressed by OID (Object IDentifier) within the MIB.

Since the Sun Management Center agent uses SNMP to communicate with the external entities like Sun Management Center server, the data items in the modules have corresponding OIDs. However, you can ignore this relation of data items and OIDS, if SNMP is not of primary interest to you. If you are a module developer and are interested in SNMP modeling of your module data, refer to the section, “Using the `mib2x` Tool” on page 198, in Chapter 10, and section, “To Generate SNMP MIB From a Module” on page 219, in Chapter 11.

CHAPTER 5

Building a Simple Module

This chapter describes how to build a simple module. It covers the following topics:

Required Components

- Module Parameters Specification—Specifying the parameters required by the module. Every module must specify a standard set of parameters and may specify additional parameters, depending on the requirements of the module..
- Data Model Creation—Identifying the components and properties required to model the managed entity. These components and properties are represented using managed object classes and managed properties, respectively, and are organized in tree hierarchy to reflect the hierarchical nature of managed entities.
- Data Model Realization—To realize the data model produced in the previous step, data acquisition mechanisms are integrated with the data models and the models are incorporated into the module framework so that it may be loaded into a Sun Management Center agent.

File Naming Conventions

Module definition files adhere to the following naming conventions:

<module><-subspec>-<descriptor>.<extension>

where

<*module*> is the module name.

<*subspec*> is an optional qualifier for the module name.

<*descriptor*> is one of a set of standard descriptors indicating the purpose of the file.

<*extension*> is one of a set of standard file extensions indicating the file type.

By convention, the <*module*> and <*subspec*> portions of the filename are common for all files associated with a specific module. This allows related module files to be easily grouped together while eliminating the chances of filename contention with the definition files of other modules.

TABLE 5-1 Standard Descriptors for Module Definition Files

`-d`	Daemon file (Model Realization File)
`-m`	Parameter file
`-models-d`	Model file

Standard Extensions

TABLE 5-2 Standard Extensions for Module Definition Files

`.x`	File in module configuration file format

Parameters Specification

Module parameters used by Sun Management Center agents are specified in a *parameter file*. The parameter file specifies the parameters that are required by the module when it is loaded. The contents of this file are also used to provide a form to prompt the user for any required parameters.

The format is:

Format: *<module>*<-subspec>-m.x

Example: solaris-example-m.x

▼ Creating a Parameter File

To create a parameter file, do the following:

1. **Specify the mandatory parameters in the parameter file.**

   Mandatory parameters are listed in the section, “Mandatory Parameters” on page 29.

2. **Identify any additional parameters required by the module and add the appropriate entries to the parameter file.**

   Additional required parameters are discussed in the section, ““Additional Parameters” on page 151,” in Chapter 9.

   This section describes the format and the possible contents of parameter files. A Solaris example parameter file is also provided in the chapter.

Mandatory Parameters

The parameter file must always include the following lines:

```
[load default-m.x]
consoleHint:moduleParams(param) = module i18nModuleName
i18nModuleDesc version enterprise i18nModuleType
param:module = <agent filename>
param:moduleName = <name of module>
param:version = <version number>
param:console = <console filename>
param:moduleType = <module type>
param:enterprise = <module enterprise>
param:location = <symbolic oid>
param:oid = <numeric oid>
param:desc = <module description>
```

where:

The line `[ load default-m.x ]` loads the descriptions and edit access specifications for all mandatory module parameters. Any parameter definitions that are identical for all modules are placed in the `default-m.x` file, so that they

do not have to be specified redundantly in each module parameter file individually. This makes it easy to add new common parameter definitions to all modules in the future, if required.

`consoleHint:moduleParams(param)` lists the parameters that are displayed to Sun Management Center console users when the module is to be loaded.

Additional parameters can be added to this list as required.

<*agent filename*> must be the filename of the associated module Agent file (<*module*><*-subspec*>-d.x) without the "`-d.x`" suffix. If the proper naming conventions are being followed, this is <*module*><*-subspec*> in all cases.

<*name of module*> is a short string naming the module. This parameter can be used internally by the agent.

<*version number*> is the version number of the module and is used internally by the agent. This must be the same as the version used as part of the module name (<*module*><*-subspec*>).

<*console filename*> must be the same as <*agent filename*>.

This field exists for historical reasons, and is not actively used in the Sun Management Center implementation.

<*moduleType*> identifies the module category. This value determines where the module is placed in the Sun Management Center console when it is loaded.

The <*moduleType*> field must be set to one of the following values:

```
hardware

operatingSystem

localApplication

remoteSystem

serverSupport
```

Note – *serverSupport* modules are not visible in the standard hierarchy view of the agent in the Sun Management Center console. More importantly, modules of this type do not contribute to the overall status of the agent. This module type must be used only for modules that are used internally by the agent. As a result, user must not be a able to load these modules (see the section entitled “Making a Module Not Loadable” for more information).

<*module enterprise*> specifies the SNMP enterprise under which this module is loaded. This value must correspond to the enterprise specified in the <*location*> module parameter.

<*location*> specifies the full symbolic OID (from .iso) where the module is to be loaded. The location string must not contain any “-” characters as indicated by RFC 1903.

<*oid*> specifies the numeric OID described by the <*location*> parameter.

<*module description*> is a verbose description of the module functionality. This parameter is used when exporting the module MIB during the creation of the module MIB text file.

Note – The il8n parameters relate to internationalization and are explained in the section, “Internationalizing Modules” on page 33.

Example Parameter File

The Solaris example module does not require any additional parameters. The Parameter file for the Solaris Example module, `solaris-example-m.x`, is shown below.

```
#
# Parameter file for Solaris Example module
#
[ load default-m.x ]

#
# Mandatory Parameters
#
consoleHint:moduleParams(param) = module i18nModuleName\
i18nModuleDesc version enterprise i18nModuleType

param:module     = solaris-example
param:moduleName = Solaris Example
param:version    = 1.0
param:console    = solaris-example
param:moduleType = operatingSystem
param:enterprise = halcyon
param:location   =
.iso.org.dod.internet.private.enterprises.halcyon.
primealert.modules.solaris.example
param:oid        = 1.3.6.1.4.1.1242.1.2.90.1
param:desc      = This is an example module monitoring cpu, load,
and filesystem statistics.

param:i18nModuleName  = base.modules.solaris-example:moduleName
param:i18nModuleType  = base.modules.solaris-example:moduleType
param:i18nModuleDesc  = base.modules.solaris-example:moduleDesc
?param:i18nModuleName?format = i18n
?param:i18nModuleType?format = i18n
?param:i18nModuleDesc?format = i18n
```

Internationalizing Modules

Internationalization, or l18n (an abbreviation for the term “internationalization” that also denotes the existence of 18 characters between “l” and “n”), is the process of enabling your code so that the programs can display localized text, such as text in French, Chinese, and other languages instead of displaying the English text for labels, errors, headings, titles and so forth.

If no localized text is available, then the default English text is used. I18n of a program consists of defining the I18n keys and I18n text, and using the I18n key instead of a plain text while displaying the text. In Sun Management Center modules, there are some mandatory module parameters that should be internationalized. These are discussed in this section. For more details on this, refer to the Chapter 15.

Mandatory Parameters for Internationalization in the Parameters File

You can specify the I18n (international) keys for the module parameters in the parameters file. When displaying the module parameters in the console, the localized strings corresponding to the I18n key are used.

The parameter file must also include the following lines:

```
param:i18nModuleName  = base.modules.<module><-subspec>:moduleName
param:i18nModuleType  = base.modules.<module><-subspec>:moduleType
param:i18nModuleDesc  = base.modules.<module><-subspec>:moduleDesc

?param:i18nModuleName?i18n = yes
?param:i18nModuleType?i18n = yes
?param:i18nModuleDesc?i18n = yes
```

The mandatory lines are required to internationalize the default values of the `moduleName`, `moduleType`, and `desc` parameters. For user-defined parameters, refer to Chapter 6. These are the values that are displayed in the Sun Management Center console when a module is loaded. The following corresponding entries are required in the module properties file:

```
moduleName=<internationalized text>
moduleType=<internationalized text>
moduleDesc=<internationalized text>
```

where <*internationalized text*> is the internationalized values for the module parameters. The properties file is discussed in the next section.

Properties File

Each module that requires internationalization must have a properties file. The name of this file must be:

```
<module><-subspec>[_<lang>].properties
```

where

<*module*> is the name of the module.

<*subspec*> is an optional subspec for the module.

<*lang*> is the locale representing the language of the internationalized text contained in this file. Not specifying the <*lang*> parameter in the file name indicates that the contents of the this file are to be used as the default when the current locale is not supported.

The properties file contains key/value pairs for each part of the module to be internationalized. The format of the properties file is:

```
<key>=<value>
```

where

<*key*> is of the form <*spec*>[.<*spec*>....]. All <*key*>s in this file must be unique.

<*value*> is the internationalized text.

The corresponding properties file for the Solaris Example, for the English locale, would contain following internationalized text:

```
moduleName=Solaris Example
moduleType=Operating System
moduleDesc=This is an example module monitoring cpu, load, and
filesystem statistics.
```

Example Properties File

For example, a fragment of the English Properties file for the Solaris Example module (`solaris-example.properties`) is:

```
#
# filesystem
#
filesystem=Filesystem Usage

#
# fileTable
#
filesystem.fileTable=Filesystem Usage Table

#
# fileEntry
#
filesystem.fileTable.fileEntry=Filesystem

filesystem.fileTable.fileEntry.mount=Mount Point
filesystem.fileTable.fileEntry.size=Total Size (KB)
filesystem.fileTable.fileEntry.avail=Available Space(KB)
filesystem.fileTable.fileEntry.pctUsed=Space Used (%)
filesystem.fileTable.fileEntry.pctRate=Rate (%/sec)
```

The specification of a long <*key*> separated by dots (.) is for organizational purposes.

Referencing Internationalized Text

To reference internationalized text in the module, the following specification is used in the module definition files:

```
<type>:<qualifier> = base.module.<module><-subspec>:<key>
```

where:

- <*type*> is the type of qualifier. This is typically `consoleHint`.
- <*qualifier*> is the part of the module that requires internationalization.
- <*module*> is the module name.
- <*subspec*> is the subspec of the module, which is optional.
- <*key*> is the same <*key*> that is used in the Properties File.

For example, the internationalized text for the description of the mount point node in the Solaris Example module can be accessed using:

```
consoleHint:mediumDesc =
base.modules.solaris-example:filesytemtable.fileTable.
fileEntry.mount
```

Data Model Specifications

Data model creation defines the hierarchical structure of the managed object classes and managed properties required to model the entity to be managed.

▼ Creating a Data Model

The steps involved in creating a data model are:

1. **Identify the components that comprise the entity to be managed.**

   Refer to the section, ""Identifying Components and Properties of Managed Entity" on page 36," that identifies the properties that describe each of the components.

2. **Define the data model.**

   Refer to the section, ""Defining the Data Model Structure" on page 38," that describes how to assign the structural primitive classes to the identified components and properties and organize them in a tree hierarchy to support status summarization.

3. **Add node descriptions.**

   Refer to the section, ""Adding Node Descriptions" on page 48," that describes how to add descriptions to the managed object classes and managed properties.

Identifying Components and Properties of Managed Entity

In the data model, the physical and logical components are represented by managed object classes. The properties of the components are represented by managed properties.

This action involves identifying each of the physical and logical components and properties of the managed entity that are to be included in the data model.

Note – The data model need not include every component and property of the managed entity. At the very least, it should contain the information that is pertinent to the determination of the status of the entity. Additional information about the entity can be included at the discretion of the module designer.

Solaris Example—Components and Properties

For the Solaris Example module, the following logical components comprise the Solaris operating environment:

- CPU
- System
- File System

Note – The Solaris operating system contains many other logical components like swap, networking, processes, and so forth. However, to simplify the example, the components included in the model are limited to those listed above.

The CPU component encompasses aspects related to CPU usage. The system component covers aspects related to the operating system in general. The file system component includes aspects related to mounted file systems.

The next action is to determine what properties are required to describe each of the components. These properties are used to derive the health of each component, and collectively, summarize the health of the managed entity.

CPU

The CPU component can be characterized by the following properties:

- Percentage of time the CPU is idle
- Percentage of time the CPU is busy
- Percentage of CPU time spent on user processes
- Percentage of CPU time spent on system processes
- Average percentage of time the CPU is busy

System

The system component can be characterized by information about users logged into the system and the system load.

User properties can include such things as:

- Current console user
- Number of users
- Number of sessions
- Primary user

The system load can be described by the 1, 5, and 15 minute load averages.

File System

The file system component is composed of one or more mounted file systems. Each mounted file system is characterized by:

- Mount point
- Total size
- Available disk space
- Percentage of disk space used

Defining the Data Model Structure

The following sections describe how to define the data module structure. All examples are included at the end of this section.

Node Definition and Trees

Because of the hierarchical nature of components of systems, the managed object classes and managed properties are organized in a tree hierarchy referred to as the data model structure. Managed object classes and managed properties are represented by branches and leaves, respectively. Branches and leaves in the tree structure are referred to as *nodes.*

Note that the data model structure is an intermediate representation of the data model as it only contains the skeletal framework of the data model. The structural primitives and the basic object tree configurations are described in the following sections.

Structural Primitives

Structural primitives specify characteristics required by nodes to define their place in the object tree hierarchy.

Nodes can inherit from one of the following structural primitives:

- MANAGED-OBJECT
- MANAGED-PROPERTY
- MANAGED-PROPERTY-CLASS
- MANAGED-OBJECT-TABLE
- MANAGED-OBJECT-TABLE-ENTRY

In module configuration file notation, nodes inherit from a primitive using the following syntax:

```
<node> = { [ use <PRIMITIVE1> <PRIMITIVE2> ] <body> }
```

where

<*node*> is the node name. This name must be unique amongst its peer nodes.

<*PRIMITIVE*> is a defined primitive that the node is inheriting from.

<*body*> are entries for the node in module configuration file format

MANAGED-OBJECT

```
<node> = { [ use MANAGED-OBJECT ] <body> }
```

This primitive is used to identify managed object nodes that are branch nodes in the object tree. Branch nodes do not store data, instead they contain other branch or leaf nodes.

MANAGED-PROPERTY

```
<node> = { [ use MANAGED-PROPERTY ] <body> }
```

This primitive identifies managed property nodes that are leaf nodes in the object tree. These nodes store data associated with the property.

MANAGED-PROPERTY-CLASS

```
<node> = { [ use MANAGED-PROPERTY-CLASS ] <managed properties> }
```

This primitive is used to group related managed properties of a managed object together. Nodes that inherit from this primitive are branch nodes.

Managed property classes are functionally identical to managed obhects. The managed property class primitive has been created for convenience, to indicate more clearly that it is the properties that are being grouped.

MANAGED-OBJECT-TABLE

```
<node> = { [ use MANAGED-OBJECT-CLASS ] <body> }
```

This branch primitive is used in conjunction with the `MANAGED-OBJECT-TABLE-ENTRY` primitive when constructing a table of managed properties. Each managed property in the table can store a vector of data instead of simple scalars. The object that uses this primitive must have a child who uses the `MANAGED-OBJECT-TABLE-ENTRY` primitive. The `MANAGED-OBJECT-TABLE` and `MANAGED-OBJECT-TABLE-ENTRY` nodes are included to support the SNMP MIB representation of tabular data.

Note – The Sun Management Center console retrieves data stored in a table using a single SNMP get request (maximum SNMP packet size is 64k is — this is due to the fact that SNMP uses UDP, which has such a limitation). If the table contains a large number of managed properties or a large number of rows, all the data cannot be retrieved by the Sun Management Center console in a single get request and the console reports an error. In such instances, the table must be made smaller or split into multiple tables for display in the Sun Management Center console.

MANAGED-OBJECT-TABLE-ENTRY

```
<node> = { [ use MANAGED-OBJECT-TABLE-ENTRY ]
    index = <managed property name>
        descColumn = <managed property name>
<managed properties>
}
```

This branch primitive is used in conjunction with the MANAGED-OBJECT-TABLE primitive when constructing a managed object with a table of managed properties. This primitive must be used by an object that is the child of a MANAGED-OBJECT-TABLE object. This object must contain one or more managed property objects that forms the columns of the table.

To allow specific rows of the table to be referenced through SNMP, this object must specify an `index` qualifier that corresponds to one or more of its child managed properties that uniquely identify the row. This is the value that is used as part of the status messages. For example, if `index` is set to the table node containing mount point information and the /var file system is greater than 90% full, the status message is:

```
/var > 90%
```

If more than one child node is specified as part of the `index`, the string used in the status message is a comma separated list of the indexes. For example, if `index` is set to the mount point and disk name nodes, the status string reads:

```
/var,/dev/dsk/c0t0d0s5 > 90%
```

Optionally, this object can also specify `descColumn` to specify a child property value as a descriptive row name to be displayed on the Sun Management Center console for row status messages. If `descColumn` is specified to be a node containing the disk name, the status message is as follows regardless of the `index` setting:

```
/dev/dsk/c0t0d0s5 > 90%
```

Example Data Model File

This section contains the following examples:

- Solaris Example—Model File
- Solaris Example—CPU Data Model Structure
- Solaris Example—Performance Data Model Structure
- Solaris Example—`filesystems` Data Model Structure

Solaris Example—Model File

The following code example lists the Solaris Example Model file, named `solaris-examples-models-d.x`. This example has two independent managed objects: CPU, and system .

CODE EXAMPLE 5-1 Solaris Example—Model File

```
#
# Solaris Managed Object and Property Models
#
type = reference

#
# Cpu Managed Object
#
cpu = { [ use MANAGED-OBJECT ]

    mediumDesc             = CPU Properties
    consoleHint:mediumDesc = base.modules.solaris-example:cpu

    idle = { [ use PERCENT MANAGED-PROPERTY ]
        shortDesc       = Idle
        mediumDesc      = CPU Idle Time
        fullDesc        = Percentage of time the CPU is in the idle state
        units           = %
```

CODE EXAMPLE 5-1 Solaris Example—Model File *(Continued)*

```
    }

    busy = { [ use PERCENT MANAGED-PROPERTY ]
        shortDesc       = Busy
        mediumDesc      = CPU Busy Time
        fullDesc        = Percentage of time the CPU is in the busy state
        units           = %

    }

}

#
# System Managed Object
#
system = { [ use MANAGED-OBJECT ]

    mediumDesc             = System Information
    consoleHint:mediumDesc = base.modules.solaris-example:system

    userstats = { [ use MANAGED-PROPERTY-CLASS ]
        mediumDesc             = User Statistics
        consoleHint:mediumDesc = base.modules.solaris-example:system.userstats

        consoleUser = { [ use STRING MANAGED-PROPERTY ]
            shortDesc       = User
            mediumDesc      = Console User
            fullDesc        = User currently logged in on the console

            consoleHint:mediumDesc = base.modules.\
solaris-example:system.userstats.consoleUser
        }

        numUsers = { [ use INT MANAGED-PROPERTY ]
            shortDesc       = #Users
            mediumDesc      = Number of Users
            fullDesc        = Number of unique users currently logged in

            consoleHint:mediumDesc = base.modules.\
solaris-example:system.userstats.numUsers
        }

    }
```

CODE EXAMPLE 5-1 Solaris Example—Model File *(Continued)*

```
    load = { [ use MANAGED-PROPERTY-CLASS ]
        mediumDesc             = Load Average
        consoleHint:mediumDesc = base.modules.solaris-example:system.load

        one = { [ use FLOAT MANAGED-PROPERTY ]
            shortDesc       = 1min
            mediumDesc      = 1 Min Load Avg
            fullDesc        = The one minute load average

          consoleHint:mediumDesc = base.modules.solaris-example:system.load.one
        }

        five = { [ use FLOAT MANAGED-PROPERTY ]
            shortDesc       = 5min
            mediumDesc      = 5 Min Load Avg
            fullDesc        = The five minute load average

            consoleHint:mediumDesc = base.modules.\
solaris-example:system.load.five
        }

    }
}

# The solaris-example.properties file is shown below. This file also contains
# the module parameter internationalization key and strings.

#
# Module Parameters
#
moduleName=Solaris Example
moduleType=operatingSystem
moduleDesc=This is an example module monitoring cpu, load, and filesystem
statistics.

#
# Node Descriptions
#
cpu=CPU Properties

system=System Information

system.userstats=User Statistics
```

CODE EXAMPLE 5-1 Solaris Example—Model File *(Continued)*

```
system.userstats.consoleUser=Console User
system.userstats.numUsers=Number of Users

system.load=Load Average
system.load.one=1 Min Load Avg
system.load.five=5 Min Load Avg
```

Solaris Example—CPU Data Model Structure

The managed properties of the managed object, CPU, are idle time, busy time, system time, user time, and busy average.

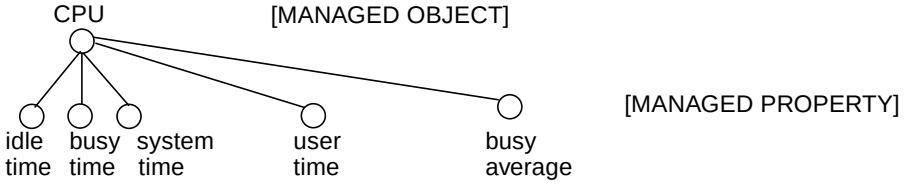

FIGURE 5-1 Simple Managed Object

The data model structure of the CPU is shown in module configuration file format:

CODE EXAMPLE 5-2 Module Configuration File Format

```
cpu = { [ use MANAGED-OBJECT ]
    idle = { [ use MANAGED-PROPERTY ] }
    busy = { [ use MANAGED-PROPERTY ] }
    system = { [ use MANAGED-PROPERTY ] }
    user = { [ use MANAGED-PROPERTY ] }
    average = { [ use MANAGED-PROPERTY ] }
}
```

Solaris Example—Performance Data Model Structure

The performance managed object contains a managed property, console user, plus the nested managed objects: CPU and load average, each of which contain their own managed properties.

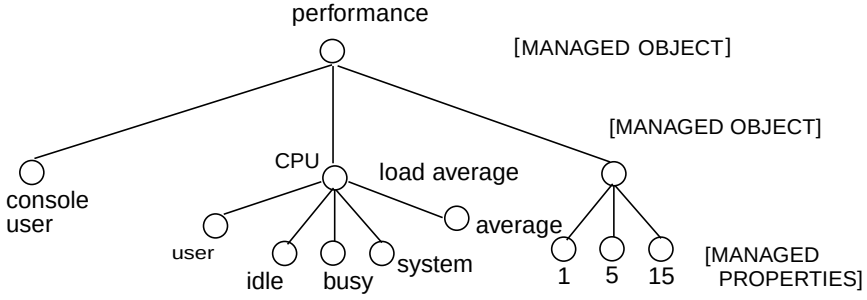

FIGURE 5-2 Performance Managed Object

The following is a code example for the performance data model structure:

CODE EXAMPLE 5-3 Performance Data Model Structure

```
performance = {[use MANAGED-OBJECT]
    consoleUser = {[use MANAGED-PROPERTY]}
    cpu = {[use MANAGED-OBJECT]
        idle = {[use MANAGED-PROPERTY]}
        busy = {[use MANAGED-PROPERTY]}
        system = {[use MANAGED-PROPERTY]}
        user = {[use MANAGED-PROPERTY]}
        average = {[use MANAGED-PROPERTY]}
    }
    loadavg = {[use MANAGED-OBJECT]
        one = {[use MANAGED-PROPERTY]}
        five = {[use MANAGED-PROPERTY]}
        fifteen = {[use MANAGED-PROPERTY]}
   }
}
```

Note – CPU and loadavg can also be MANAGED-PROPERTY CLASS.

Solaris Example—`filesystems` *Data Model Structure*

In this example, the managed object is the file system component. The managed properties are the mount point, total size, KB available, and percent space used.

The `mount` property is designated to be the `index`. This allows each file system to be referenced by their mount point names.

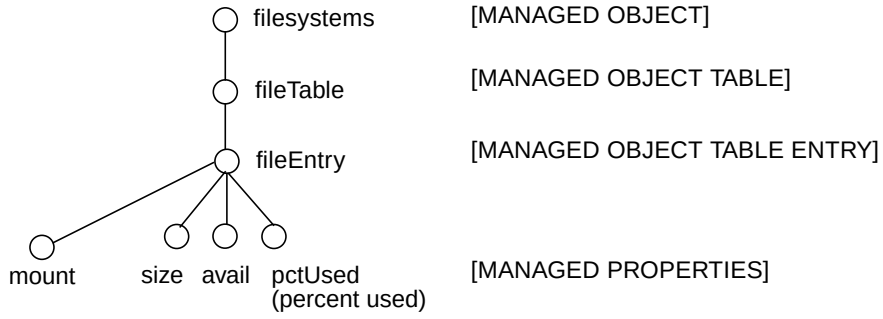

FIGURE 5-3 Managed Object with Tabular Properties

The data model structure of the file system component is shown in module configuration file notation:

CODE EXAMPLE 5-4 File System Data Model Structure Code

```
filesystems = {[use MANAGED-OBJECT]
  fileTable = {[use MANAGED-OBJECT-TABLE]
     fileEntry = {[use MANAGED-OBJECT-TABLE-ENTRY]
       index = mount
       mount = {[use MANAGED-PROPERTY]}
       size = {[use MANAGED-PROPERTY]}
       avail = {[use MANAGED-PROPERTY]}
       pctUsed = {[use MANAGED-PROPERTY]}
    }
  }
}
```

Adding Data Types

After creating the data model structure, data types must be assigned to the managed property nodes. Data primitives classes can be used to characterize the data and alarm types of the manage property nodes. These primitives are assigned to the managed property nodes using the same syntax as before.

```
<node> = { [ use <primitive> MANAGED-PROPERTY ] <body> }
```

All managed properties must use a data type primitive. These primitives define the type of data stored in the property. Data type primitives can be optionally combined with an alarm type that characterizes the alarm checks performed on the property's data value.

These data and alarm primitives have the following form:

```
<data type>[<alarm type>]
```

where

<*data type*> represents the type of data stored in the primitive

<*alarm type*> optionally specifies the type of alarm checks to perform

---

Note – Alarm types are optional and are discussed in the chapter on Chapter 7.

Available Data Types

Data type primitives can be one of the following:

- `STRING`—general string type
- `INT`—`asn.1` integer type
- `FLOAT` —floating point value
- `PERCENT`—percentage
- `COUNTER`—`asn.1` counter type
- `GAUGE`—`asn.1` gauge type
- `OID`—`asn.1` OID type
- `TIMETICKS`—`asn.1` time tick type
- `OCTETSTRING`—`asn.1` octet string type

- IPADDRESS——asn.1 IP address type
- UINT——asn.1 unsigned integer type
- PHYSADDRESS——asn.1 physical address type
- TESTANDINCR——asn.1 test and increment type

The TESTANDINCR is a primitive that has a special behavior. SNMP sets nodes using this primitive causing the agent to check the value that is being set against the current node value. If the values are the same, the current value is incremented by one. If the values do not match, the set will fail.

Adding Node Descriptions

One action in the creation of the data model is to add descriptions to the nodes.

Three levels of description qualifiers can be defined for each node:

- shortDesc (optional)——typically a one word description of the node
- mediumDesc (mandatory)——a descriptive string about the node that must be unique within the scope of the module, less than 20 characters.
- fullDesc (optional)——a complete description of the node

The mediumDesc value is used as part of the status message displayed on the console.

A managed property can also have a unit qualifier. This qualifier specifies the units (if any) of the data value stored in that property and is used for display purposes only. Together, the mediumDesc and unit qualifiers are used as part of a template when generating status messages of the managed property that is displayed on the Sun Management Center console. The unit qualifier is also used to automatically specify the units for the vertical axis when graphing data values, for example, units = sec.

Node Type Based on Operational Behavior

Each node must be assigned a type that defines its operational behavior:

```
type = <node type>
```

One of the node types is reference.

Reference nodes are objects that are loaded for use as a template in the model file. A node is specified as a *reference* node as follows:

```
type = reference
```

The entry, `type = reference`, indicates that the managed object classes defined in this file are to be used only as a reference. These reference objects are typically loaded into a template area from which the active object tree, which will perform the management functions can inherit the model structure. The load into the template area is done through the agent file (<*module*><*-subspec*>-d.x) when the model is realized.

The other node types are discussed in section, “Node Type Based on Operational Behavior.”

Simple Data Model Realization

Creating a data model defines the layout of the data in your module. After doing this, you have to specify the methods by which the agent will acquire the values for this data layout. This process of specifying the methods is referred to as data acquisition (DAQ) or data model realization in the following sections and subsequent chapters.

Data model realization is defined in the agent file. Additional files that can be created to support the agent file include filter file, procedure file, shell scripts, C libraries, and Tcl extension packages.

The `agent` file <*module*><*-subspec*>`-d.x` is mandatory for a module:

```
<module><-subspec>-d.x
```

Example:

```
solaris-example-d.x
```

Note – The data model realization file (`*-d.x`) is sometimes interchangeably called the agent file.

Steps Involved in Data Model Realization

The following sections describe the model realization process:

- Implementing Data Acquisition Mechanisms describes how to devise the means by which you can acquire the data values of the managed properties in the data model.
- Integrating Data Acquisition describes how to create the agent file that loads and inherits from the data models.
- Loading the DAQ Services describes how to add the DAQ mechanisms to the agent file so that the acquired data can be disseminated to the appropriate nodes.

Mandatory Contents of Every Data Model Realization File

The agent file is in module configuration file format, and must contain the following standard entries:

```
[ use MANAGED-MODULE ]
[ load <module><-subspec>-m.x ]
[ requires template <module><-subspec>-models-d ]

<managed object classes>
```

The line `[ use MANAGED-MODULE ]` mandates the inheritance of the `MANAGED-MODULE` primitive at the root of the module. This primitive provides managed properties at the root of the module that reflect the status and availability of the entire module.

The line `[ load <`*module*`><-`*subspec*`>-m.x ]` loads the parameter file entries into the root of the module. The entries can be referenced by other objects in the module.

The line `[ requires template <`*module*`><-`*subspec*`>-models-d ]` loads the model file into the template area for use by the agent. Multiple entries can be specified if more than one model file is required.

The line <`managed object classes`> represents the main body of the agent file. It consists of the managed object classes that use the data models loaded in the template area. The exact contents of this section will vary with each module.

Implementing Data Acquisition Mechanisms

The underlying logic to perform the data acquisition can be implemented in a number of different ways, including:

- UNIX programs and shell scripts
- Tcl/TOE code discussed in the Modules Appendix.
- C-code libraries and Tcl/TOE command extensions discussed in the Modules Appendix

For more information about the Tcl programming language, refer to the *Tcl and TK Toolkit* manual.

UNIX Programs and Shell Scripts

Much system data can be acquired using UNIX system commands like `vmstat`, `swap`, `iostat`, `netstat`, `ps`, and so forth. Existing custom programs and scripts can also be employed to gather data.

Shell scripts can be employed to parse the raw results of the UNIX programs using such tools as `sed`, `grep`, `awk`, and so forth. When returning data to the agent from UNIX programs or scripts, each data element must be separated by newlines.

The standard extension for the shell scripts is `.sh`, for example, `filename.sh`.

Integrating Data Acquisition

This action involves integrating DAQ capabilities with the intermediate agent file created previously. The objective of this step is to facilitate the *refresh* operation. The refresh operation consists of performing DAQ and disseminating the acquired data into the appropriate managed property nodes.

The dissemination of a buffer of data into a tree of managed objects and managed properties is known as the *data cascade*. There are strict rules that govern the manner in which data can be cascaded in the tree. Data can be acquired one piece at a time and placed into managed properties, or larger amounts of information can be acquired in a single data acquisition operation and cascaded into several managed properties.

The DAQ capabilities are integrated into the agent file by doing the following:

- Loading the DAQ services
- Specifying node types
- Specifying refresh parameters

Loading the DAQ Services

Typically, DAQ functionality and/or support services must be loaded or created by the agent to enable the DAQ mechanisms in the Sun Management Center agent. The possible types of DAQ services that can be loaded or created are listed below out of which only Bourne shell script is discussed here. The actual DAQ services that must be loaded or created depends on the DAQ mechanisms being employed.

- Bourne shell service
- Tcl shell service
- Tcl filters
- Tcl procedures
- Tcl command extension package

For more information on Tcl shell service, filters, and procedures, refer to Chapter 6.

Bourne Shell Services

If the DAQ was implemented using UNIX commands or shell scripts, the agent must create a Bourne shell service object to execute these commands. The Bourne shell service is essentially an object maintaining a pipe to one or more shell processes to which commands can be directed and the results returned asynchronously.

The module configuration file specification for a Bourne shell service object is:

```
_services = { [ use SERVICE ]
        sh = {
            command = "pipe://localhost//bin/sh;transport=shell"
            max = <max shells>
        }
    }
```

where:

<*max shells*> specifies the maximum number of shell subprocesses to spawn. This is typically set to 2. This value indicates the number of refresh commands that use the Bourne shell service that can be executed co-currently.

For example, if <*max shells*> is set to 1, refresh commands can be queued waiting for previous commands to finish. Setting this value to a larger number increases the number of captive shells and resources used by the agent. `object _services.sh` can then be used by other objects for Bourne shell DAQ services.

Node Type Based on Operational Behavior

The following section discusses node types based on the operational behavior to be used in Model Realization File.

Each node must be assigned a type that defines its operational behavior:

```
type = <node type>
```

The possible node types are active, passive, derived, or reference.

Passive and drived nodes are discussed in Chapter 6.

Reference nodes are already discussed in this chapter. They are only used in the models file.

Active Nodes

A node is specified to be *active* using the following specification:

```
type = active
```

All data acquisition operations are initiated by an active node. An active node is a managed object or managed property that has refresh information associated with it.

In general, active nodes specify a refresh service, refresh command, and a refresh interval. To acquire data, the refresh command is executed in the context of the refresh service at every refresh interval.

Additional refresh parameters are available for filtering the data, specifying initialization behavior, and setting up internal refresh triggers.

All active nodes must specify the following qualifiers:

```
refreshService = <service object>
refreshCommand = <command to run in the context of refreshService>
refreshInterval = <timex specification defining when to run the command>
```

Other qualifiers that can be specified by active nodes are discussed in the next chapter.

Mandatory RefreshQualifiers for Active Nodes

refreshService

A refresh service is an object within the agent that can be used for data acquisition. A refresh service must be specified for active and derived nodes.

Invoking a *refresh command* in the context of a *refresh service* activates typical refresh operations A refresh command is a service-dependent command that defines the specific operation to perform. A refresh command is sent to the refresh service each time a refresh is triggered.

Refresh services can be any object that supports the service interface. Typical, refresh services are objects that maintain pipes to shells (like Bourne shell or perl process). Other refresh services that can be specified by active nodes are discussed in the next chapter.

The `refreshService` qualifier specifies the context in which the `refreshCommand` runs.

Bourne Shell Service

```
refreshService = _services.sh
```

Use this service when the refresh command is a UNIX command or shell script. To use this service, a Bourne shell service object (`like_services.sh`) must have been created at the root of this module as described in the Loading the DAQ Services section.

refreshCommand

The refresh command must be specified for active and derived nodes.

```
refreshCommand = <command>
```

The refresh command is a service-dependent command that defines the specific operation to perform. Conceptually, the refresh command is sent to the refresh service each time a refresh is triggered.

The refresh command must be appropriate for the specified refresh service. Depending on the refresh service specified, the refresh command can be such things as the following will be discussed in a later chapter:

- UNIX commands and scripts
- Tcl commands and procedures

All module parameters and parameters defined in the `value` slice can be referenced by the `refreshCommand`. This information can be referenced using `%<parameter>` as part of the `refreshCommand`. For example `refreshCommand = myCommand %moduleName` will pass in the `moduleName` module parameter to the command `myCommand`.

The exit status of UNIX commands and scripts are not used by the agent. Instead the agent interprets any data return on `stderr` as a data acquisition error. If this happens, the active node automatically goes into an indeterminate alarm state, indicating that the node failed to update. In addition, no data cascades into the agent regardless if any data was returned on `stdout`.

The number of elements and the type of data returned by the refresh command is dictated by the number and type of nodes into which the data is cascaded. Generally, data cascade is the dissemination of data collected by an active node into passive nodes. The exception is when a leaf node is active and collects data for itself.

If the refresh command returns less than the required number of elements for a complete data cascade, an error message is generated by the agent. However, the data cascade still occurs. All available data is used by the nodes at the beginning of the cascade. Nodes towards the end of the data cascade will not receive any data. If the refresh command returns more than the required number of elements, the additional data elements are ignored.

A DAQ error can also be encountered if there is a mismatch between the type of data being cascaded into a node and the node's data type. Again, the data cascade continues until the node where the data type mismatch occurs. Note that empty strings are valid data values for nodes whose data type is `STRING`.

refreshInterval

The refresh interval specifies the time specification that the refresh command is executed.

```
refreshInterval = <timex specification>
```

The refresh interval must be specified for active and optionally for derived nodes. If no refreshInterval is specified for an active node, it is treated by the agent as an on-demand refresh node, that is, the data values are refreshed whenever the data is requested as opposed to being computed periodically or on initialization only.

Note – Alarm rules or checks are not supported for on-demand nodes. If an on-demand node is detected to have an alarm rule or check during module load, the agent aborts immediately.

If the refresh interval is not specified, the refresh command is executed whenever the data is requested through SNMP.

If the refresh interval is set to 0, the refresh command is executed only on initialization. The refresh command is *not* executed when the data is requested through SNMP. Refer to the Appendix, "Time Expression Specifications," for more information. For more information on refreshQualifiers, refer to Chapter 6.

Example of a Simple Module

Refer to the examples, `helloworld01` through `helloworld03` in the Sun Management Center Developer Environment installed on your system. The online versions of these examples are in the following location once you install the developer environment:

```
/opt/SUNWsymon/sdk/examples/modules
```

This section contains the following examples:

- Example Data Model Realization File
- The `solaris-example-console-user-d.sh` File

Example Data Model Realization File

The following code example lists the Solaris Example Model Realization File.

CODE EXAMPLE 5-5 Solaris Example Model Realization File

```
[ use MANAGED-MODULE ]
[ requires template solaris-example-models-d ]

#
# Load Module Parameters
```

CODE EXAMPLE 5-5 Solaris Example Model Realization File *(Continued)*

```
#
[ load solaris-example-m.x ]

#
# Define services required by this module
#
_services = { [ use SERVICE ]
    #
    # Standard Bourne Shell
    #
    sh = {
        command = "pipe://localhost//bin/sh;transport=shell"
        max     = 2
    }
}

#
# Cpu Information
#
cpu = { [ use templates.solaris-example-models-d.cpu ]

    idle = {
        type            = active
        refreshService  = _services
        refreshCommand  = echo 10
        refreshInterval     = 60
    }

    busy = {
        type            = active
        refreshService  = _services
        refreshCommand  = echo 20
        refreshInterval     = 60
    }
}

#
# System User and Load Information
#
system = { [ use templates.solaris-example-models-d.system ]
    userstats = {
        consoleUser = {
            type            = active
```

CODE EXAMPLE 5-5 Solaris Example Model Realization File *(Continued)*

```
            refreshService  = _services.sh
            refreshCommand  = solaris-example-console-user-d.sh
            refreshInterval     = 60
        }
        numUsers = {
            type            = active
            refreshService  = _services.sh
            refreshCommand  = echo 10
            refreshInterval     = 60
        }
    }

    load = {
one = {
            type = active
            refreshService  = _services.sh
            refreshCommand  = echo 10.2
            refreshInterval     = 60
}
five = {
            type = active
            refreshService  = _services.sh
            refreshCommand  = echo 10.2
            refreshInterval     = 60
        }
    }
}
```

The `solaris-example-console-user-d.sh` *File*

The `solaris-example-console-user-d.sh` file is shown below:

CODE EXAMPLE 5-6 The `solaris-example-console-user-d.sh` File

```
#!/bin/sh
echo "I am a console user (from Sh)"
```

CHAPTER 6

Advanced Data Model Realization Techniques

This chapter includes the following sections:

What are Filters

The filter file defines data filters implemented using Tcl/TOE procedures. These filters are used to extract the pertinent information from the raw results of data acquisition commands. This enables the agent to use raw extensions or system commands (such as `df` for Solaris software) to acquire data, with the processing/parsing of the output being performed within the agent, not through external utilities such as `awk` or `sed`.

Note – For more information on Tcl/TOE, refer to the Appendix A.

Standard Extensions for File Name

```
<module><-subspec>-d.flt
```

This file is optional for a module, and only exists if the module is using filter functions. Only Tcl/TOE procedures can be defined in this file.

Examples of Filters

CPU Data Filter

The UNIX command `vmstat 10 2` returns four lines of data in the format:

```
procs     memory            page            disk          faults      cpu
r b w   swap  free  re  mf pi po fr de sr f0 s0 s1 s2   in   sy   cs us sy id
0 0 0  24120  7040   0  63  6  2  4  0  1  0  4  2  1  215  347  167  5 11 83
0 0 0 280020  1172   0   1  0  8 18  0  2  0  1  0  0   39   52   46  1  1 98
```

This data is passed as the datalist argument in the `cpuFilter` procedure. The procedure parses the percent idle, system and user fields from the fourth line of data, computes the percent busy, and returns the results. The code is written in Tcl.

```
proc cpuFilter { datalist } {
    set line [ lindex $datalist 3 ]
    set user [ string range $line 69 71 ]
    set system [ string range $line 72 74 ]
    set idle [ string range $line 75 77 ]
    set busy [ expr 100 - $idle ]
    return “$idle $busy $system $user”
}
```

User Data Filter

The UNIX command `who` returns data in the following format:

```
tom        console      Oct  2 09:54
tom        pts/1        Oct  4 23:43    (superior)
tom        pts/0        Oct  2 09:55
tom        pts/2        Oct  2 09:55
tom        pts/5        Oct  2 09:55
tom        pts/4        Oct  3 09:29
tom        pts/3        Oct  2 09:55
```

This data is passed to the `userFilter` procedure as the datalist argument. The procedure loops through each line to determine the console user and count the number of unique users and sessions.

```
proc userFilter { datalist } {
    set console none
    set ucount 0
    set scount 0
    foreach line $datalist {
        set name [ lindex $line 0 ]
        set source [ lrange $line 5 end ]
       if { [ lindex $line 1 ] == “console” } { set console $name }
        if { [ catch { set users($name) } ] } {
            set users($name) “”
            incr ucount
        }
        if { [ catch { set sessions($name:$source) } ] } {
            set session($name:$source) “”
            incr scount
        }
    }
    return [ list $console $ucount $scount ]
}
```

Load Data Filter

The UNIX command `uptime` returns data in the following format:

```
 11:45pm  up 4 day(s), 16:29,  2 users,  load average: 0.00, 0.00, 0.01
```

This data is passed to the loadFilter procedure where the 1-, 5-, and 15-minute load averages are picked out using a regular expression pattern.

```
proc loadFilter { datalist } {
    regexp {^.*up *(.*), +[0-9]+ +users*,.+: *([^,]+), *([^,]+), *([^,]+)}
    [ lindex $datalist 0 ] dummy a b c d
    return [ list $b $c $d $a ]
}
```

File System Data Filter

The UNIX command `df -kF ufs` returns data in the following format:

```
Filesystem            kbytes    used   avail capacity  Mounted on
/dev/dsk/c0t0d0s0      22847   11912    8655    58%    /
/dev/dsk/c0t0d0s6     246167  185193   36364    84%    /usr
/dev/dsk/c0t0d0s3     105943    3183   92170     4%    /var
/dev/dsk/c0t0d0s7     793382       9  714043     1%    /export1
/dev/dsk/c0t0d0s5     288855   43131  216844    17%    /opt
```

This data is passed to the `fileFilter` procedure where the desired data fields are picked out from each line.

```
proc fileFilter { datalist } {
    #
    # pick out appropriate fields for each line
    #
    set result ""
    set continuation ""
    foreach line [ lrange $datalist 1 end ] {
        set line $continuation$line
        if { [ llength $line ] < 6 } {
            set continuation "$line "
            continue
        } else {
            set continuation ""
        }
        regsub {%} $line "" line
     lextract $line 1 kbytes 3 avail 4 capacity 5 mount
        lappend result $mount $kbytes $avail $capacity
    }
    return $result
}
```

Adding Filters to Data Model Realization

Example Data Model File

The following code example lists the Solaris Example Model file, `solaris-example-models-d.x`. It has three independent manged objects; CPU, system, and file system.

CODE EXAMPLE 6-1 The `solaris-example-models-d.x` File

```
#
# Solaris Managed Object and Property Models
#
type = reference

#
# Cpu Managed Object
#
cpu = { [ use MANAGED-OBJECT ]

    mediumDesc             = CPU Properties
    consoleHint:mediumDesc = base.modules.solaris-example:cpu

    idle = { [ use PERCENT MANAGED-PROPERTY ]
        shortDesc       = Idle
        mediumDesc      = CPU Idle Time
        fullDesc        = Percentage of time the CPU is in the
                          idle state
        units           = %

    }

    busy = { [ use PERCENT MANAGED-PROPERTY ]
        shortDesc       = Busy
        mediumDesc      = CPU Busy Time
        fullDesc        = Percentage of time the CPU is in the
                          busy state
        units           = %
```

CODE EXAMPLE 6-1 The `solaris-example-models-d.x` File *(Continued)*

```
    }

    system = { [ use PERCENT MANAGED-PROPERTY ]
        shortDesc       = System
        mediumDesc      = CPU System Time
        fullDesc        = Percentage of time the CPU is running
                          in system mode
        units           = %

    }

    user = { [ use PERCENT MANAGED-PROPERTY ]
        shortDesc       = User
        mediumDesc      = CPU User Time
        fullDesc        = Percentage of time the CPU is running
                          in user mode
        units           = %

    }

}

#
# System Managed Object
#
system = { [ use MANAGED-OBJECT ]

    mediumDesc             = System Information
    consoleHint:mediumDesc = base.modules.solaris-example:system

    userstats = { [ use MANAGED-PROPERTY-CLASS ]
        mediumDesc             = User Statistics
        consoleHint:mediumDesc = \
         base.modules.solaris-example:system.userstats

        consoleUser = { [ use STRING MANAGED-PROPERTY ]
            shortDesc     = User
            mediumDesc    = Console User
           fullDesc     = User currently logged in on the console

            consoleHint:mediumDesc = \
          base.modules.solaris-example:system.userstats.consoleUser
        }
```

CODE EXAMPLE 6-1 The `solaris-example-models-d.x` File *(Continued)*

```
        numUsers = { [ use INT MANAGED-PROPERTY ]
            shortDesc       = #Users
            mediumDesc      = Number of Users
            fullDesc        = Number of unique users currently
                              logged in

            consoleHint:mediumDesc = \
           base.modules.solaris-example:system.userstats.numUsers
        }

        numSessions = { [ use INT MANAGED-PROPERTY ]
            shortDesc       = Sessions
            mediumDesc      = Number of User Sessions
            fullDesc        = Number of currently active user
                              sessions

        }

        primaryUser = { [ use STRING MANAGED-PROPERTY ]
            shortDesc       = User
            mediumDesc      = Primary System User
            fullDesc        = The login name of the primary user

        }


    }

    load = { [ use MANAGED-PROPERTY-CLASS ]
        mediumDesc             = Load Average
        consoleHint:mediumDesc = \
         base.modules.solaris-example:system.load

        one = { [ use FLOAT MANAGED-PROPERTY ]
            shortDesc       = 1min
            mediumDesc      = 1 Min Load Avg
            fullDesc        = The one minute load average

            consoleHint:mediumDesc = \
             base.modules.solaris-example:system.load.one

        }
```

CODE EXAMPLE 6-1 The `solaris-example-models-d.x` File *(Continued)*

```
        five = { [ use FLOAT MANAGED-PROPERTY ]
            shortDesc       = 5min
            mediumDesc      = 5 Min Load Avg
            fullDesc        = The five minute load average

            consoleHint:mediumDesc = \
             base.modules.solaris-example:system.load.five
        }

    }
}


#
# Filesystem Table
#
filesystems = { [ use MANAGED-OBJECT ]
    mediumDesc           = Filesystems

    fileTable = { [ use MANAGED-OBJECT-TABLE ]
        mediumDesc       = Filesystem Property Table

        fileEntry = { [ use MANAGED-OBJECT-TABLE-ENTRY ]
            mediumDesc   = Filesystem
            index        = mount

            mount = { [ use STRING MANAGED-PROPERTY ]
                shortDesc     = Mount Pt
                mediumDesc    = Filesys Mount Point
               fullDesc      = The mount point for the filesystem

            }

            size = { [ use INT MANAGED-PROPERTY ]
                shortDesc      = Filesys Sz
                mediumDesc     = Filesystem Size
                fullDesc       = Total filesystem size in KBytes
                units          = KB

            }

            avail = { [ use INT MANAGED-PROPERTY ]
                shortDesc      = FilesysAvl
```

CODE EXAMPLE 6-1 The `solaris-example-models-d.x` File *(Continued)*

```
                mediumDesc     = Filesystem Space
               fullDesc       = Available filesys diskspace in KB
                units          = KB

            }

            pctUsed = { [ use PERCENT MANAGED-PROPERTY ]
                shortDesc      = Disk Used
                mediumDesc     = Filesystem Capacity
                fullDesc       = Percentage of Disk Space Used
                units          = %

            }
        }
    }
}
```

The `solaris-example.properties` file is shown below. This file also contains the module parameter internationalization key and strings:

CODE EXAMPLE 6-2 The `solaris-example.properties` File

```
#
# Module Parameters
#
moduleName=Solaris Example
moduleType=operatingSystem
moduleDesc=This is an example module monitoring cpu, load,
and filesystem statistics.

#
# Node Descriptions
#
cpu=CPU Properties

system=System Information

system.userstats=User Statistics
system.userstats.consoleUser=Console User
system.userstats.numUsers=Number of Users

system.load=Load Average
system.load.one=1 Min Load Avg
```

CODE EXAMPLE 6-2 The `solaris-example.properties` File *(Continued)*

```
system.load.five=5 Min Load Avg
```

Example Data Model Realization File Using Tcl Filters

The following code example lists the Solaris example model realization file using Tcl filters developed in the section, “Examples of Filters” on page 62.”

CODE EXAMPLE 6-3 Solaris Example Model Realization File

```
[ use MANAGED-MODULE ]
[ requires template solaris-example-models-d ]

#
# Load Module Parameters
#
[ load solaris-example-m.x ]

#
# Define services required by this module
#
_services = { [ use SERVICE ]
    #
    # Standard Bourne Shell
    #
    sh = {
        command = "pipe://localhost//bin/sh;transport=shell"
        max     = 2
    }
}

#
# Load filters required by this module
#
_filters = { [ use PROC ]
    [ source solaris-example-d.flt ]
}

#
```

CODE EXAMPLE 6-3 Solaris Example Model Realization File *(Continued)*

```
# Cpu Information uses the cpuFilter which is already discussed
# in the previous section
#
cpu = { [ use templates.solaris-example-models-d.cpu _filters ]
    type                = active
    refreshService      = _services.sh
    refreshCommand      = vmstat 10 2
    refreshFilter       = cpuFilter
    refreshInterval     = 60
}

#
# System User and Load Information uses the userFilter
# which is already discussed
# in the previous section
#
system = { [ use templates.solaris-example-models-d.system ]
    userstats = { [ use _filters ]
        type                = active
        refreshService      = _services.sh
        refreshCommand      = who
        refreshFilter       = userFilter
        refreshInterval     = 120

        primaryUser = {
            type            = active
            refreshService  = _services.sh
            refreshCommand  = solaris-example-primary-user-d.sh
            refreshInterval = 86400
        }
    }

    load = {
        one = {
            type            = active
            refreshService  = _services.sh
            refreshCommand  = echo 10.2
            refreshInterval     = 60
        }
        five = {
            type            = active
            refreshService  = _services.sh
            refreshCommand  = echo 10.2
```

CODE EXAMPLE 6-3 Solaris Example Model Realization File *(Continued)*

```
            refreshInterval     = 60
        }
    }

}

#
# Filesystem Information uses the fileFilter
# which is already discussed
# in the previous section
#
filesystems = { [use templates.solaris-example-models-
d.filesystems
                  _filters]
    type                = active
    refreshService      = _services.sh
    refreshCommand      = df -kF ufs
    refreshFilter       = fileFilter
    refreshInterval     = 120
}
```

The solaris-example-primary-user-d.sh file is shown below:

CODE EXAMPLE 6-4 The `solaris-example-primary-user-d.sh` File

```
#!/bin/sh

echo "I am a primary user (from Sh)"
```

Note – All the filters used, for example, CPUfilter, and so on, in the example above, are defined in the file `solaris-example-d.flt`.

Loading the DAQ Services

Tcl Filters

If the DAQ was implemented using Tcl filters, the Filters file must be loaded into a container object. Nodes that want to call a procedure defined in the Filters file must inherit this object.

The `_services.<shell>` object can then be used by other objects for Tcl shell DAQ services.

RefreshQualifier for Filters

refreshFilter

Refresh filters can be specified in active and derived nodes:

```
refreshFilter = <Tcl command or procedure>
```

The `refreshFilter` qualifier specifies a Tcl command or procedure that is used to process the data acquired by the refresh command. The Tcl procedure must take a single argument that is the result returned by the refresh command. The result of the refresh filter is cascaded into the managed properties. Recall that if the refresh command is implemented as a UNIX command or shell script, any data returned on `stderr` constitutes a data acquisition error. As a result, no data is passed to the refresh filter, regardless if data was returned on `stdout` too.

Solaris Example—Loading the Filter File

In the following example, the Solaris Example Filter file is loaded into the `filters` object. This object can be inherited by other objects that want to use the procedures defined in the Filter file. One such object is the CPU managed object.

The following is a code example of loading the filter file:

CODE EXAMPLE 6-5 Loading the Filter File

```
_filters = { [ use PROC ]
    [ source solaris-example-d.flt ]
}
cpu = { [ use templates.solaris-example-models-d.cpu _filters ]
... }
```

Advanced Data Acquisition Mechanisms

These are other techniques and are discussed in detail later on in this chapter.

Tcl/TOE Code

Standard Tcl/TOE commands, such as `file` to get file statistics, can be used to acquire data. In addition, Tcl procedures can be written to filter raw results returned by UNIX programs. Tcl provides many useful commands for parsing strings (`regexp, regsub,` and so forth.).

In general, Tcl procedures are preferred over using shell scripts when filtering data as they are typically easier to implement and more efficient. When returning data to the agent from Tcl procedures or commands, the data elements must be elements in a Tcl list.

C Code Libraries and Tcl/TOE Command Extensions

C-code libraries and Tcl/TOE command extensions can also be written to perform DAQ. This is accomplished by packaging the DAQ functions as shared object libraries that can be dynamically loaded by the Sun Management Center agent.

When returning more than one item of data from Tcl commands, the data elements must be elements in a Tcl list.

The next few sections introduce concepts necessary to understand advanced data model realization techniques.

Other Node Types based on their Operational Behavior

Passive Nodes

Nodes that do not actively collect data but instead have data cascaded into them are known as *passive* nodes. By default, all nodes are passive, unless otherwise specified using the type qualifier.

A node can be explicitly specified to be passive using the following specification:
`type = passive`

Passive managed property nodes can specify an update filter to process that data being cascade into it.
`updateFilter = <Tcl procedure or command>`

If no update filter or other qualifier is required, it is unnecessary to explicitly declare a passive node in the agent file at all. For such nodes, it is sufficient to model them in the model file only. For more details on `updateFilter`, refer to the section, “updateFilter” on page 81.

Derived Nodes

A node is specified to be a *derived* node using the following specification:

```
type = derived
```

Derived nodes establish dependency relationships with the nodes on which they rely through the use of the *refresh triggers* specification. Nodes can be triggered by the change in value or status of another node, and refresh automatically when such an event occurs. Derived nodes can also refresh at an interval, although this is usually unnecessary if the triggers are specified properly.

A *derived* node can use other MIB nodes as the service(s) for its refresh. In other words, its value is often a function of the values or qualifiers of one or more other managed properties. Through the use of derived variables, it is possible to create nodes whose value represents averages, rates of change, specific digital filters (for example: high pass, low pass, or band pass or other useful calculated information).

All derived nodes must specify the following refresh parameters:

```
refreshService = <service object>
refreshCommand = <command to run in the context of refreshService>
refreshTrigger = <node name>[:<event>] [<node name>[:<event>] ...]
```

The `refreshCommand` could also be Tcl commands and procedures.

Note – The refresh interval is optional for a derived node. Derived nodes may be forced to update at periodic intervals, although this is usually unnecessary if the refresh triggers are speficied properly. For more information on refreshTrigger, refer to the next section.

---

refreshQualifiers & Other Qualifiers

The following refreshQualifiers can also be specified in active and derived nodes.

timeoutInterval

The timeout interval can be specified for active and derived nodes.

`timeoutInterval = <timex specification>`

If the refresh command does not complete within the specified `timeoutInterval`, the command is aborted. In that case, the alarm state of the node is marked indeterminate (unknown value). The default is no timeout. This can be used in conjunction with `refreshMode = sync` (described later) to ensure that the agent does not hang while collecting data.

refreshTrigger

Refresh triggers must be specified in derived nodes.

`refreshTrigger = <node name>[:<event>] [<node name>[:<event>] ...]`

Derived nodes establish dependency relationships with one or more nodes on which they rely through the use of the *refresh triggers* specification. Nodes can be triggered off the change in value or status of another node, and refreshes automatically when such an event occurs.

The `refreshTrigger` specifies the name of the node that the derived node depends on. The possible events are:

`status`—event generated upon status change of the node

`refresh`—event generated when an active node is to refresh its value

`update`—event generated in managed properties when the data values are updated

`set`—event generated in the node when a SNMP set is made

If no event is specified, the occurrence of any of the above events in the specified node will trigger the execution of the refresh operation. If an event is specified, only the occurrence of the specified event on the specified node triggers the execution of the refresh operation.

Specifying Node Name

<*node name*> can be specified two ways. If the triggering node is a direct child of the current node, or is the direct child of a superior of the current node, the object name can be used directly. For example, in the Solaris Example Module, the `CPU` managed object has several child managed properties, including: `idle`, `user`, `system`, `busy`, and `average` (derived). Any of these refresh triggers are valid for the derived `average` node:

```
refreshTrigger = busy:update
```

```
refreshTrigger = idle:refresh
```

```
refreshTrigger = system:status
```

If the triggering node does not meet the above criteria, the full name of the triggering node must be specified. Wildcards are allowed. It is important to take care that the name is uniquely specified. Otherwise, the first node matching the name becomes the trigger.

The `average` node could trigger off the 15 minutes load average as follows:

```
refreshTrigger = *solaris*load.fifteen
```

Wildcards can be used as placeholders only for *full* node names.You cannot use wildcards to partially specify a node name. For example, the following is *not* valid, because the node name `solaris` is not fully specified:

```
refreshTrigger = *sol*load.fifteen
```

Specifying RefreshTriggers from a Node in Another Module

The triggering node must not reside in a different module from the current node. Otherwise, if the other module is unloaded, the triggering relationship is lost, and both modules must be reloaded to restore the relationship.

Note – This cannot be used for active nodes.

---

refreshParams

Refresh params can be specified in active and derived nodes:

```
refreshParams = <params>
```

The `refreshParams` qualifier is used to specify arguments to be passed to the refresh command. If the refresh mode is set to `multi`, `multilist`, or `multiecm`, and the refresh params specifies a space separated list of arguments, the refresh command is executed once for each argument. The next section describes `refreshMode`.

refreshMode Qualifier

The *refresh mode* can be specified in active and derived nodes:

```
refreshMode = async | sync
```

The `refreshMode` qualifier specifies the execution mode of the refresh command.

async

By default, active nodes have a refresh mode of `async`. This specification implies that the refresh operation is asynchronous. That is, the agent is allowed to process other events during the execution of the refresh command.

sync

If the refresh command must return immediately with the result, setting the `refreshMode` to `sync` reduces the overhead associated with an asynchronous command.

initInterval

The initialization interval can be specified in active and derived nodes.
`initInterval = <`*`timex specification`*`>`

The `initInterval` specifies, the time window within which the node must run the `refreshCommand` for the first time after the module initializes.

Note – `initInterval` does not specify an exact time at which the node will first issue its refresh command. Rather, this interval specifies a time range. Sometime within that time range, the first refresh command is issued. The exact time is randomized by the agent to spread out load.

If `initInterval` is specified as –1, the first execution of the `refreshCommand` is executed as specified by the `refreshInterval`.

The initialization interval enables module designers to control the rate at which the module nodes begin their data acquisition operations. This might be done to prioritize the order in which nodes initialize, or to avoid large spikes in data acquisition activity during module start up.

initHoldoff

The initialization *holdoff* can be specified in active and derived nodes:

```
initHoldoff = <timex specification>
```

The `initHoldoff` qualifier specifies the time, in Timex specification, to wait before running the refresh command the first time.

The initialization *holdoff* is typically used to delay the execution of a specific node that depends on the value of another node that must execute first. Effectively, the `initHoldoff` time is added to the `initInterval` time, thereby delaying initialization. `initHoldoff` must never be set to less than 2 seconds.

Here are some examples of how initHoldoff and initInterval interact:

```
initInterval = 10
initHoldoff = 2
```

The node will initialize sometime between 2 and 12 seconds.

```
initInterval = 10
initHoldoff = 30
```

The node will initialize sometime between 30 and 40 seconds.

Check Qualifiers

checkCommand, checkService and checkInterval

The checkCommand, checkService, and checkInterval can be specified for active and derived nodes to perform *check operations*.

`checkService = <service specification>`

`checkCommand = <command to execute in service context>`

`checkInterval = <timex specification>`

The check operation provides a mechanism for triggering refresh operations based on some criteria tested by the check operation. Typically, the check operation is a lighter weight operation and is performed at a higher frequency relative to the refresh operation. This mechanism enables managed objects to be monitored in a more timely fashion without the performance penalty of executing the refresh operation at a higher frequency.

The check operation triggers the execution of the refresh operation only when the value returned by the check command differs from the value from the previous check.

For example, when monitoring the contents of a file, the refresh operation can involve reading the contents of the file at every refresh interval. To monitor the file more effectively, the last modification date of the file can be checked at a higher frequency to determine if the file has changed. If the check detects that the file has changed, the refresh operation is triggered. Hence, the check mechanism provides an efficient way to monitor the file since checking the last modification date of a file (using *stat* system call) is much more lightweight than having to open, read, and close the file.

`checkService` specifies the service used to run the `checkCommand`. If this qualifier is not specified, the `checkCommand` shall use the `refreshService` service.

`checkInterval` specifies the interval, in `timex` specification, at which to run the `checkCommand`. This `checkInterval` must be specified if the `checkCommand` is specified.

`checkInterval` and `refreshInterval` operate completely independently of each other. `refreshInterval` specifies the interval at which refreshes will definitely occur. `checkInterval` specifies the interval at which refreshes *can* occur, depending on the result of the check condition.

For example:

```
checkInterval = 10
refreshInterval = 300
```

This means that every 10 seconds `checkCommand` is executed. If the check passes, the refresh command is invoked. But regardless of those checks, every 5 minutes the refresh command is invoked.

updateFilter

Passive managed property nodes can specify an update filter to process the data being cascaded into it.

`updateFilter = <Tcl command or procedure>`

The update filter specifies a Tcl command or procedure to process the data being cascaded into the passive node. Like the `refreshFilter`, the Tcl procedure specified by the `updateFilter` takes a single argument that is the data that is cascaded into the passive node.

To execute a user-defined Tcl procedure, the procedure must be available in the current node's context. To execute a user-defined Tcl command extension, the appropriate Tcl package must be loaded as described in the earlier section *Loading the DAQ Services*.

refreshService

Besides the `refreshService` command discussed in the previous chapter, the other `refresh` services are:

- SNMP stack (for doing data acquisition from other agents)

- Internal service that enables access to built-in or dynamically-loaded extensions to the agent process
- Another node in the MIB tree (from which you can acquire data)

SNMP Service

```
refreshService = .services.snmp
```

Specify the SNMP service when the refresh command is an SNMP get request for acquiring data from another SNMP agent. The SNMP service object is created by all Sun Management Center agents for general SNMP communications.

Internal Service

```
refreshService = _internal
```

Specify the internal service when the refresh command is a Tcl/TOE command or procedure to be executed in the current node's context.

To execute user-defined Tcl procedures, the procedures must be available in the current node's context.

To execute user-defined Tcl command extensions, the appropriate Tcl package must be loaded as described in the earlier section"Loading the DAQ Services" on page 73.

Superior Service

```
refreshService = _superior
```

Specify the superior service when the refresh command is a Tcl/TOE command or procedure to be executed in the context of the current node's superior in the tree hierarchy.

For example, in the Solaris Example module, the CPU managed object node contains a managed property node called "busy." In that example, the CPU node is the superior of the busy node.

To execute user-defined Tcl procedures, the procedures must be available in the superior's context.

To execute user-defined Tcl command extensions, the appropriate Tcl package must be loaded as described in the earlier section Loading the DAQ Services.

MIB Node Service

```
refreshService = <node name>
```

Use this type of service when the refresh command is to be executed in the context of another MIB node. The refresh command specified must be available in the context of the specified MIB node.

This specification is often used for derived nodes that specify the MIB node whose value the derived node depends on.

For example, in the Solaris Example Module, the `CPU` managed object has several child managed properties, including: `idle`, `user`, `system`, `busy`, and `average` (derived). So, the refresh service for the derived `average` node is set to one of these other children, such as:

```
refreshService = busy
```

For information on what are valid *<node name>* specifications, refer to the description of `refreshTrigger` in the section on “refreshTrigger” on page 76.

Note – The specified node could reside in an entirely different module. However, this type of interdependency between modules is not suggested, since the modules can be loaded or unloaded independently.

Data Model Realization Specifications with Tcl procedures as DAQ

Example Data Model File

The following code example lists the additional specifications needed in the Solaris Example Model file:

CODE EXAMPLE 6-6 Solaris Example Model File

```
# Additional managed property average needs to be managed and
# is to be added to the managed object cpu
# in the models file of solaris-example from the previous section o
# on the Example Data Model File.
cpu = { [ use MANAGED-OBJECT ]
       .....
       ....
    average = { [ use PERCENTHI MANAGED-PROPERTY ]
        shortDesc       = AvgCPU
        mediumDesc      = Average CPU Usage
        fullDesc        = Average percentage of time CPU is busy
        units           = %

    }

}
```

The following is an example of the Data Model Realization file using Tcl procedures as DAQ:

CODE EXAMPLE 6-7 Solaris Example Model Realization File

```
[ use MANAGED-MODULE ]
[ requires template solaris-example-models-d ]

#
# Load Module Parameters
#
[ load solaris-example-m.x ]
```

CODE EXAMPLE 6-7 Solaris Example Model Realization File *(Continued)*

```
#
# Define services required by this module
#
_services = { [ use SERVICE ]
    #
    # Standard Bourne Shell
    #
    sh = {
        command = "pipe://localhost//bin/sh;transport=shell"
        max     = 2
    }
}

#
# Load filters required by this module
#
_filters = { [ use PROC ]
    [ source solaris-example-d.flt ]
}

_procedures = {
        [ use PROC ]
        [ source solaris-example-system.prc ]
        [ source solars-example-average-d.prc]

}

#
# Cpu Information
#
cpu = { [ use templates.solaris-example-models-d.cpu _filters ]
    type               = active
    refreshService     = _services.sh
    refreshCommand     = vmstat 10 2
    refreshFilter      = cpuFilter
    refreshInterval    = 60

    average = { [ use _procedures ]
        type           = derived
        refreshService  = _internal
        refreshTrigger  = busy:update
        refreshCommand  = getAverage
```

CODE EXAMPLE 6-7 Solaris Example Model Realization File *(Continued)*

```
    }
}

#
# System User and Load Information
#
system = { [ use templates.solaris-example-models-d.system }
    userstats = { [ use _filters ]
        type                = active
        refreshService      = _services.sh
        refreshCommand      = who
        refreshFilter       = userFilter
        refreshInterval     = 120

        numUsers{ [ use _procedures ]
        refreshService = _internal
        updateFilter = numUsersFilter


        primaryUser = {  [ use _procedures]
            type            = active
            refreshService  = _internal
            refreshCommand  = getPrimaryUser
            initInterval    = 10
            refreshInterval = 86400
        }
    }

    load = {
        one = {
            type            = active
            refreshService  = _services.sh
            refreshMode = sync
            refreshCommand  = echo 10.2
            initInterval = 20
            refreshInterval     = 60

        }
        five = {  [ use _procedures ]
            type            = active
            refreshService  = _services.sh
            refreshCommand  = echo 10.2
```

CODE EXAMPLE 6-7 Solaris Example Model Realization File *(Continued)*

```
            initInterval = 60
            refreshInterval     = 60
            checkService = _internal
            checkCommand = testFive
            checkInterval = 10
        }
    }


}

#
# Filesystem Information
#
filesystems = { [use templates.solaris-example-models\
d.filesystems _filters]
    type                = active
    refreshService      = _services.sh
    refreshCommand      = df -kF ufs
    refreshFilter       = fileFilter
    refreshInterval     = 120
}
```

The following is an example of the corresponding `solaris-example-system.prc` File:

CODE EXAMPLE 6-8 The `solaris-example-system.prc` File

```
# This is called by the primaryUser node
proc getPrimaryUser{}{
 set primaryUser "Hello"
 return $primaryUser
}

# This is the updateFilter for numUsers
proc numUsersFilter{ index value } {
    if { $value > 0 } {
        return $value
    } else {
        return 1
    }

}
```

CODE EXAMPLE 6-8 The `solaris-example-system.prc` File *(Continued)*

```
# This is the checkCommand
proc testFive{}{
   set result 10.2
   return $result
}
```

The following is an example of the corresponding `solaris-example-average-d.flt` File:

CODE EXAMPLE 6-9 The `solaris-example-average-d.flt` File

```
# The corresponding solaris-example-average-d.flt file
# finds the value of the other nodes and finds their average.

proc getAverage{} {
   set busy [ toe_send [ locate busy ] getValue ]
   set idle [ toe_send [ locate idle ] getValue ]
   set user [ toe_send [ locate user] getValue ]
   set system [ toe_send [ locate system] getValue ]
   set average [ expr  ( $busy + $idle + $user + $system )/4 ]
}
```

Standard Extension for File Name

```
<module><-subspec>-d.prc
```

Objects in the MIB tree can perform special data acquisition functions or alarm status actions. These specialized functions can be specified as Tcl procedures and placed in a module-specific procedures file. This provides a simple mechanism to override or extend the functionality of the core MIB object primitives.

This file is optional for a module, and only exists if the module is using procedures. Only Tcl/TOE procedures can be defined in this file.

Loading the DAQ Services

The following section describes the Tcl procedures and node types based on the operational behavior to be used and refresh qualifiers.

Tcl Procedures

If the DAQ was implemented using Tcl procedures, the Procedures file must be loaded into a container object. Nodes that need to call a procedure defined in the Procedure file must then inherit this object.

```
_procedures = { [ use PROC ]
     [ source <modules><-subspec>-d.prc ]
}
<node> = { [ use <PRIMITIVE> _procedures ] ... }
```

Node Type Based on Operational Behavior

Node types can be active, derived or passive.

Refresh Qualifiers

All the refresh qualifiers discussed earlier are applicable. Also, the refreshService will be `_internal`.

Data Model Realization Specifications with C libraries and Tcl/TOE Command Extensions as DAQ

Solaris Example Data Model Realization File

This is an example of the migration of DAQ functionality to Tcl command extensions. The discussion of this example follows in the next few sections.

CODE EXAMPLE 6-10 Agent File Modifications

```
[ requires package ssi ]
[ requires template solaris-example-models-d ]
[ use MANAGED-MODULE ]
```

CODE EXAMPLE 6-10 Agent File Modifications *(Continued)*

```
[ load solaris-example-m.x ]

_services = { [ use SERVICE ]
    sh = {
        command = "pipe://localhost//bin/sh;transport=shell"
        max             = 2
    }
}
#
# default the refresh service for the entire module
#
refreshService      = _internal

cpu = { [ use templates.solaris-example-models-d.cpu ]
    type                = active
    refreshMode         = sync
    refreshCommand      = ssinfo cpu
    refreshInterval     = 60

    average = {
        type            = derived
        refreshTrigger  = busy:update
        refreshCommand  = digitalFilter [ valueOf busy.0 ]
        refreshParams   = 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1
    }
}
system = { [ use templates.solaris-example-models-d.system ]
    userstats = { [ use _filters ]
        type                        = active
        refreshCommand              = ssinfo user
        refreshInterval             = 120

        primaryUser = {
            type            = active
            refreshService  = _services.sh
            refreshCommand  = solaris-example-primary-user-d.sh
            initInterval    = 10
            refreshInterval = 86400
        }
    }
 load = { [ use _filters ]
        type                        = active
        refreshCommand              = ssinfo load
        refreshInterval             = 120
    }
}
```

CODE EXAMPLE 6-10 Agent File Modifications *(Continued)*

```
filesystems = { [use templates.solaris-example-models-
d.filesystem ]
    type                = active
    refreshMode         = sync
    refreshCommand      = ssinfo file
    refreshInterval     = 120
}

[ load solaris-example-d.def ]
```

Shown below are code fragments from the `ssi` package files.

CODE EXAMPLE 6-11 Code Fragments From `ssi` Package File

```
packages.h
.
.
.
#define PKG_SSI  "ssi"
extern int Ssi_Init();
.
.
.
pkgssi.c
.
.
.
int
Ssi_Init(interp)
    Tcl_Interp *interp;
{
    int code;

    code = Tcl_PkgProvide(interp, PKG_SSI, "1.0");
    if (code != Tcl_OK) {
        return code;
    }
    /* --- create “ssinfo” command --- */
    Tcl_CreateCommand(interp, “ssinfo”, cmdSsinfo, (ClientData)
“ssinfo”,
                      (Tcl_CmdDeleteProc *) NULL);
    return(Tcl_OK);
}

int
cmdSsinfo(dummy, interp, argc, argv)
    ClientData dummy;                   /* Not used. */
```

CODE EXAMPLE 6-11 Code Fragments From `ssi` Package File *(Continued)*

```
    Tcl_Interp *interp;                /* Current interpreter. */
    int argc;                          /* Number of arguments. */
    char **argv;                        /* Argument strings. */
{
.
.
.
switch(*argv[1]) {
        case "l":
        case "L":
            if(!strcasecmp(argv[1], "load")) {
                float one, five, fifteen;
             code = ssiGetLoadAverage(&one,&five,&fifteen);
               if (code == 0) {
 sprintf(buf, "%3.2f %3.2f %3.2f", one, five, fifteen);
                    Tcl_AppendResult(interp, buf, (char *)NULL);
                }
                else {
                    ssi_error = 1;
                }
                   found_option = 1;
            }
            break;
.
.
.
    return(status);
}
```

Shown below are code fragments of the DAQ C library code:

CODE EXAMPLE 6-12 DAQ C code

```
siSolaris.c
A code fragment from the ssi code is shown below.
.
.
.
int ssiGetLoadAverage(float *pfOneMin, float *pfFiveMin, float
*pfFifteenMin)
{
    long    laAveNRun[ 3 ];
    int             iSize;

    if ( !bSSIInit )
    {
        return ( ssiAPI_not_init );
```

CODE EXAMPLE 6-12 DAQ C code *(Continued)*

```
    }

    if ( !( NlistArray[ X_AVENRUN ].n_value ) )
    {
         int iEntry = X_AVENRUN;
         if ( initKvmEntries ( &iEntry , 1 ) != 0 )
         {
             return ( ssiAPI_kvm_nlist_failed );
         }
    }

    iSize = sizeof ( laAveNRun );
    if ( kvm_read(pKD, NlistArray[ X_AVENRUN ].n_value,
                   (char *)(laAveNRun), iSize) != iSize )
    {
            return ( ssiAPI_loadavg_failed );
    }

    *pfOneMin  = (float)loaddouble ( laAveNRun[ 0 ] );
    *pfFiveMin = (float)loaddouble ( laAveNRun[ 1 ] );
    *pfFifteenMin  = (float)loaddouble ( laAveNRun[ 2 ] );

    return ( 0 );
} /* end ssiGetLoadAverage () */
.
.
.
Solaris Example - Tcl command extension
```

Steps Involved

The migration of functionality from scripts to C involves the following steps:

- Writing a C library to perform the functions performed by the scripts
- Writing a Tcl extension to integrate the library functions
- Modifying the agent definition file to make use of the new Tcl extensions

Writing a C Library

Although the C-code data acquisition functions can be put directly in the Tcl extension, placing the functions in a generic library enables other C programs to use the same functions, which improves code coverage and increases the reliability of the code. For example, the following is a C subroutine used to retrieve the system load average:

CODE EXAMPLE 6-13 Code Fragment Used to Retrieve System Load Average

```
int ssiGetLoadAverage(float *pfOneMin, float *pfFiveMin, float *pfFifteenMin)
{
    long    laAveNRun[ 3];
    int             iSize;

    if ( !bSSIInit )
    {
        return ( ssiAPI_not_init );
    }

    if ( !( NlistArray[ X_AVENRUN ].n_value ) )
    {
         int iEntry = X_AVENRUN;
         if ( initKvmEntries ( &iEntry , 1 ) != 0 )
         {
             return ( ssiAPI_kvm_nlist_failed );
         }
    }
iSize = sizeof ( laAveNRun );
    if ( kvm_read(pKD, NlistArray[ X_AVENRUN ].n_value,
                   (char *)(laAveNRun), iSize) != iSize )
    {
            return ( ssiAPI_loadavg_failed );
    }

    *pfOneMin  = (float)loaddouble ( laAveNRun[ 0 ] );
    *pfFiveMin = (float)loaddouble ( laAveNRun[ 1 ] );
    *pfFifteenMin  = (float)loaddouble ( laAveNRun[ 2 ] );
    return ( 0 );
} /* end ssiGetLoadAverage () */
```

This function can then be combined with other functions that determine system information to create a library of functions that access the kernel and determine system specific information.

Writing a Tcl Extension

To use C library functions in Tcl, Tcl extension files, referred to as `packages`, must be created. These packages define an initialization procedure that enables Tcl commands to run C code. When creating packages, consider the following issues:

- Package naming
- Writing the initialization procedure
- Returning data to Tcl

These topics are described in the following sections. Refer to the Tcl documentation for information about Tcl functions (`Tcl_*`).

Package Naming

To enable a single Tcl application to incorporate many different packages without experiencing naming conflicts, a naming convention must be followed. This convention specifies a short, unique prefix for each package. For example, a package that uses the system- specific C library functions may use a prefix of 'ssi' (system specific interface). This prefix is then used to name the initialization function (described in the next section) and the package shared object file (`pkgssi.so`).

Init Function

Each package must include an initialization function. This function is called when the package is loaded. The name of the initialization procedure must contain the package prefix with the first letter capitalized followed by `_Init`. For example the initialization function for the `ssi` package would be named `Ssi_Init`.

Package Registration

The initialization function is used for package and command registration.

Package registration ensures that no other versions of the same package is being used currently or will be loaded later. Package registration is done using the `Tcl_PkgProvide` command. For example, the registration of the `ssi` package version 1.0 is:

```
code = Tcl_PkgProvide(interp, "ssi", "1.0");
if (code != Tcl_OK) {
   return code;
}
```

Command Registration

Commands provided by the package are registered in the Tcl interpreter using the `Tcl_CreateCommand` function. There must be one call to `Tcl_CreateCommand` for each Tcl command created. For example:

```
Tcl_CreateCommand(interp, "ssinfo", cmdSsinfo, (ClientData)
"ssinfo",
                      (Tcl_CmdDeleteProc *) NULL);
```

In this example, a single Tcl command (`ssinfo`) is created to access all the ssi library functions. The `Tcl_CreateCommand` function creates a link between the Tcl `ssinfo` command and the C `cmdSsinfo` function. From the `cmdSsinfo` function, the appropriate `ssi` function is called based on the `ssinfo` command arguments. In this example, a design decision was made to create a single Tcl command to access all the `ssi` library functions. The other possibility is to create a Tcl command for each possible `ssi` library function. It is up to the developer to decide which is better.

Returning Data into Tcl

After the C library function has been called to acquire data, the data must be returned to Tcl interpreter. This is done by using the following Tcl commands:

- `Tcl_AppendElement`
- `Tcl_AppendResult`

For example to return the data from the `ssiGetLoadAverage` function call, the function `Tcl_AppendResult` is used:

```
code = ssiGetLoadAverage(&one,&five,&fifteen);
if (code == 0) {
    sprintf(buf, "%3.2f %3.2f %3.2f", one, five, fifteen);
    Tcl_AppendResult(interp, buf, (char *)NULL);
}
```

Note – Data must be returned to the Tcl interpreter as a string.

Loading the DAQ Services

Tcl Command Extension Packages

If DAQ is implement using a Tcl command extension package, the package shared object must be loaded using the following directive:

```
[ requires packages <package name> ]
```

This loads the package so that the Tcl commands provided by the package is available in the agent context. For example, to load a package named `pkgssi.so`, the command is:

```
[ requires packages ssi ]
```

Node Type Based on Operational Behavior

These can be active, derived or passive.

Refresh Qualifiers

To use the Tcl commands from the package, the `refreshCommand` must be modified to call the appropriate package function. In addition, the `refreshService` must be set to `_internal` to denote that the `refreshCommand` is to be run in the agent context. Finally, the `refreshMode` can be set to `sync` to optimize the function call. Other refreshQualifiers will remain the same.

Another DAQ Service

Tcl Shell Service

To enable the agent to execute Tcl command extensions in a separate subshell process:

1. **Create binary extensions in the form of a Tcl package.**

   This is described in the “C Code Libraries and Tcl/TOE Command Extensions” on page 74.

2. **Create a simple Tcl wrapper script to load the packages required by the subshell.**

   By convention, Tcl wrapper scripts are named with a `.Tcl` extension and use the `pkgload` command to load the required Tcl packages.

3. **Add a Tcl shell service object in the agent file.**

   The Tcl shell service object is an object maintaining a pipe to one or more Tcl subshell processes where commands can be directed and the results returned asynchronously.

   The module configuration file specification for a Tcl shell service object is:

```
 _services = { [ use SERVICE ]
        <shell> = {
command = "pipe://localhost//<Tcl shell name>;transport=shell"
            max = <max shells>
        }
    }
```

   where:

   <*shell*> is the name of the Tcl shell.

   <*Tcl shell name*> is the name of the Tcl script containing the `pkgload` commands.

   <*max shells*> specifies the maximum number of shell subprocess to spawn. This is typically set to 1 or 2.

The `_services.solarisShell` object can be used by other objects for Tcl shell DAQ services.

Solaris Example—Tcl Shell

The `_services.solarisShell` can be used to collect data asynchronously in a separate sub-shell.

Performance Considerations

Some data acquisition mechanisms are more efficient than others. In general, C-code libraries and Tcl command extensions are much more efficient than UNIX commands and shell scripts.

For example, CPU usage statistics can be computed using a shell script that executes the UNIX command (`vmstat`) and parses the result using (`awk`). Similarly, these CPU statistics can also be computed by running `vmsta`*t* directly and parsing the results using a Tcl procedure. Using a Tcl procedure to parse data is slightly more efficient than using UNIX filter commands like `sed` and `awk`.

Alternatively, this information can be much more efficiently computed using a Tcl command extension and C system calls that do not include the overhead of creating processes and parsing data in a shell script or in Tcl.

Note – Because the Sun Management Center agent is single threaded when running Tcl commands, it is assumed that all Tcl commands return their results with little delay. If it is expected that the Tcl command will take a significant amount of time to return its result, a Tcl shell that loads the appropriate Tcl package can be spawned as a subprocess of the agent. The Tcl command can then be directed to the subprocess and the result can be returned asynchronously to ensure that the agent is not blocked by the execution of the command.

CHAPTER 7

Alarm Management

This chapter covers the following topics:

What are Alarms

The Sun Management Center software monitors your hardware and software. When abnormal conditions occur, the Sun Management Center notifies you, through alarms. These alarms are triggered by conditions falling outside of predetermined ranges, or by Sun Management Center rules. Default alarm conditions and rules are included in the modules. In addition, you may also set up your own alarm thresholds.

Modules and Alarms

In Sun Management Center, an alarm rule performs one or more *alarm checks*. Each alarm check evaluates *alarm criteria* to determine if the managed property is in a corresponding *alarm state*. Actions can also be triggered by the alarm states; these actions are known as *status actions*.

Thus, each alarm rule is associated with a number of alarm criteria, alarm states, and optional status actions. If none of the alarm criteria are satisfied, the node is considered to be in the ok state, and hence nodes without an alarm rule are always considered okay.

Typically, the alarm rule is evaluated after completing the refresh operation (the refresh request and the subsequent data cascade). For some alarm rules, the rule may be triggered whenever a particular error message appears in a log file. These are referred to as *log rules*.

Built-In rCompare Rule

Each managed property can be assigned a single associated alarm rule. This assignment is done in the model file. A generic rule, called `rCompare`, is provided. This rule performs numeric comparisons, regular expression checks, or string comparisons. The exact checks that are to be performed are controlled by *alarm check* and *alarm limit* parameters (as described later in this chapter).

Each managed property is assigned a single associated alarm rule. This assignment is done in the model file. For standard alarm types (`HI`, `LO`, etc.), the rule `rCompare` is used by default. If a managed property is not assigned an alarm rule, then no alarm checking is performed on that managed property.

Writing Custom Rules

Custom rules can employ a wide variety of alarm criteria. They can examine the value of the node to which the rule is attached, or the value or status of a different node. A special category of rules, referred to as *log rules*, can be triggered to fire whenever a message matching a specified regular expression appears in a log file.

For more information, refer to the Chapter 8.

Alarm Management using rCompare Rule

Example Alarm File (solaris-example-d.def)

The alarm file for the Solaris example is shown below. Note that the tree structure specified in the agent file is used in this file when specifying alarm information for the managed properties.

Alarm severities are specified to reflect the relative significance of the various managed properties. Alarm limits are also specified when appropriate.

Alarm limits are specified for a number of file systems typically found on many systems.

CODE EXAMPLE 7-1 Alarm File

```
cpu = {
    idle = {
        alarmSeverity         = 3
        alarmlimit:error-lt   = 10
        alarmlimit:warning-lt = 15
        alarmlimit:info-lt    =
    }

    busy = {
        alarmSeverity         = 3
        alarmlimit:error-gt   = 95
        alarmlimit:warning-gt = 90
        alarmlimit:info-gt    =
    }

    system = {
        alarmSeverity   = 3
        alarmlimit:error-gt   = 95
        alarmlimit:warning-gt = 90
        alarmlimit:info-gt    =
    }

    user = {
        alarmSeverity         = 3
        alarmlimit:error-gt   = 95
        alarmlimit:warning-gt = 90
        alarmlimit:info-gt    =
    }

    average = {
        alarmSeverity         = 7
        alarmlimit:error-gt   = 95
        alarmlimit:warning-gt = 90
        alarmlimit:info-gt    =
    }
}

user = {
    numUsers = {
        alarmSeverity         = 5
        alarmlimit:error-gt   = 10
```

CODE EXAMPLE 7-1 Alarm File *(Continued)*

```
        alarmlimit:warning-gt = 5
        alarmlimit:info-gt    =
}
    numSessions = {
        alarmSeverity         = 5
        alarmlimit:error-gt   = 30
        alarmlimit:warning-gt = 25
        alarmlimit:info-gt    =
    }
}

filesystems = {
    fileTable = {
        fileEntry = {
            avail = {
                alarmlimit:error-lt()       = 5000
                alarmlimit:warning-lt()     = 10000
                alarmlimit:info-lt()        =
                alarmlimit:error-lt(/)      = 5000
                alarmlimit:warning-lt(/)    = 10000
                alarmlimit:info-lt(/)       =
                alarmlimit:error-lt(/usr)   = 5000
                alarmlimit:warning-lt(/usr) = 10000
                alarmlimit:info-lt(/usr)    =
                alarmlimit:error-lt(/var)   = 10000
                alarmlimit:warning-lt(/var) = 20000
                alarmlimit:info-lt(/var)    =
                alarmlimit:error-lt(/tmp)   = 10000
                alarmlimit:warning-lt(/tmp) = 20000
                alarmlimit:info-lt(/tmp)    =
                alarmlimit:error-lt(/opt)   = 5000
                alarmlimit:warning-lt(/opt) = 10000
                alarmlimit:info-lt(/opt)    =
                alarmlimit:error-lt(/usr/openwin)   = 1000
                alarmlimit:warning-lt(/usr/openwin) = 2000
                alarmlimit:info-lt(/usr/openwin)    =
            }
pctUsed = {
                alarmlimit:error-gt()       = 98
                alarmlimit:warning-gt()     = 90
                alarmlimit:info-gt()        =
                alarmlimit:error-gt(/)      = 95
                alarmlimit:warning-gt(/)    = 90
                alarmlimit:info-gt(/)       =
                alarmlimit:error-gt(/usr)   = 98
                alarmlimit:warning-gt(/usr) = 95
                alarmlimit:info-gt(/usr)    =
```

CODE EXAMPLE 7-1 Alarm File *(Continued)*

```
                alarmlimit:error-gt(/var)   = 90
                alarmlimit:warning-gt(/var) = 80
                alarmlimit:info-gt(/var)    =
                alarmlimit:error-gt(/tmp)   = 90
                alarmlimit:warning-gt(/tmp) = 80
                alarmlimit:info-gt(/tmp)    =
                alarmlimit:error-gt(/opt)   = 98
                alarmlimit:warning-gt(/opt) = 95
                alarmlimit:info-gt(/opt)    =
                alarmlimit:error-gt(/usr/openwin)   =
                alarmlimit:warning-gt(/usr/openwin) = 100
                alarmlimit:info-gt(/usr/openwin)    =
            }
        }
    }
}
```

▼ Managing Alarms using rCompare

1. **Create the rule in the models file.**

   These are explained in the section on “Using the rCompare Rule in the Models File.”

2. **Create the alarm definition file.**

   These are explained in the section on “Creating the Alarm File.”

3. **Speficy alarm limits and other alarm criteria.**

   These are explained in the section on “Specifying the Alarm Criteria.”

4. **Specify actions to be performed based on the alarm state.**

   These are explained in the section on “Specifying Status Actions.”

Using the rCompare Rule in the Models File

Example—Intermediate Data Model

In this action, data and alarm type primitives are added to the managed properties defined in the data model structure created previously. For instance, the CPU managed properties like idle, busy, system, user, and average can be represented by the `PERCENT` data type.

For the properties that represent CPU usage levels (busy, system, user, and average), it is also prudent to perform a high alarm check to detect instances when these properties exceed specified limits. Thus, these properties must use the `PERCENTHI` primitive. Conversely, the idle property reflects the percent of time the CPU is not in use; as a result, a `PERCENTLO` primitive can be used to detect times of low CPU usage.

Similar reasoning is exercised when assigning data and alarm type primitives to the other managed properties. Also, to illustrate the use of rules, the `rUsrChk` rule (defined in the `solaris-example-d.rul` file) is attached to the `consoleUser` object.

The resulting data model structure with data and alarm type primitives is:

CODE EXAMPLE 7-2 Solaris Example—Intermediate Data Model

```
cpu = { [ use MANAGED-OBJECT ]
    idle = { [ use PERCENTLO MANAGED-PROPERTY ] }
    busy = { [ use PERCENTHI MANAGED-PROPERTY ] }
    system = { [ use PERCENTHI MANAGED-PROPERTY ] }
    user = { [ use PERCENTHI MANAGED-PROPERTY ] }
    average = { [ use PERCENTHI MANAGED-PROPERTY ] }
}
system = { [ use MANAGED-OBJECT ]
    userstats = { [ use MANAGED-PROPERTY-CLASS ]
        numUsers = { [ use INTHI MANAGED-PROPERTY ] }
        numSessions = { [ use INTHI MANAGED-PROPERTY ] }
        primaryUser = { [ use STRING MANAGED-PROPERTY ] }
    }
    load = { [ use MANAGED-PROPERTY-CLASS ]
        one = { [ use FLOATHI MANAGED-PROPERTY ] }

        five = { [ use FLOATHI MANAGED-PROPERTY ] }

        fifteen = { [ use FLOATHI MANAGED-PROPERTY ] }

    }
}
filesystems = { [ use MANAGED-OBJECT ]
    fileTable = { [ use MANAGED-OBJECT-TABLE ]
        fileEntry = { [ use MANAGED-OBJECT-TABLE-ENTRY ]
            index = mount
            mount = { [ use STRING MANAGED-PROPERTY ] }
            size = { [ use INT MANAGED-PROPERTY ] }
            avail = { [ use INTLO MANAGED-PROPERTY ] }
            pctUsed = { [ use PERCENTHI MANAGED-PROPERTY ] }
```

CODE EXAMPLE 7-2 Solaris Example—Intermediate Data Model *(Continued)*

```
        }
    }
}
```

How to specify Alarms in the Data Model File

Data type primitives can be optionally combined with an alarm type that characterizes the alarm checks performed on the property's data value.

These data and alarm primitives have the following form:

```
<data type>[<alarm type>]
```

where

<*data type*> represents the type of data stored in the primitive

<*alarm type*> optionally specifies the type of alarm checks to perform

Alarm Types

The alarm type specification is optional and must be combined with a valid data type. The alarm type defines the alarm check to perform on the data value. The alarm check criteria is specified in the Chapter 8. If the alarm type is not specified, no alarm checks are performed on the affected managed property.

The possible values for the alarm type are:

- `HI`—checks if data value is greater than the specified alarm limits. This alarm type can only be combined with the `INT`, `FLOAT`, `PERCENT`, and `COUNTER` data types. It cannot be used in combination with the `STRING` data type.
- `LO`—checks if data value is less than the specified alarm limits. This alarm type can only be combined with the `INT`, `FLOAT`, `PERCENT`, and `COUNTER` data types. It cannot be used in combination with the `STRING` data type.
- `HILO`—checks if data value is less than or greater than the specified alarm limits. This alarm type can only be combined with the `INT`, `FLOAT`, `PERCENT`, and `COUNTER` data types. It cannot be used in combination with the `STRING` data type.
- `EQ`—checks if data value is equal to the specified alarm criteria. This alarm type can only be combined with the `INT`, `FLOAT`, `PERCENT`, `COUNTER` and `STRING` data types.

- NE—checks if data value is not equal to the specified alarm criteria. This alarm type can only be combined with the INT, FLOAT, PERCENT, COUNTER and STRING data types.
- REGEXP—checks if data value matches the regular expression alarm criteria. This alarm type can only be used in combination with the STRING data type.
- RULE - executes a rule check procedure. This alarm type can be used in conjunction with any data type, or, optionally, the data type can be left blank if the node exists only to support a rule and has no associated data, discussed in the Rules chapter.

Data and Alarm Type Primitive Examples

Examples of data and alarm type primitives are:

- INT—general integer type with no alarm checking
- FLOATHI—a floating point value that will check to see if the data value is greater than the specified alarm limits
- STRINGREGEXP—specifies a string type with alarm checks using regular expression patterns
- INTRULE—an integer value to which a rule check is applied
- RULE—contains no value (empty string), but a rule is to be executed for the node

Required Content in the Model Realization File

The required content in the model realization file to load the alarma file is:

```
[load <modules><-subspec>-d.def]
```

Creating the Alarm File

The alarm file defines information used by alarm checks performed on managed properties. This file is loaded by the agent file with the following line:

```
[ load <module><-subspec>-d.def ]
```

The contents of the alarm file are overlaid on top of the MIB object tree defined in the agent file.

In general, alarm management information is likely to be modified by site administrators, whereas the object hierarchies and DAQ mechanisms specified in the agent file are not typically modified. Therefore, alarm management information is defined in a separate file from the agent file to facilitate the specification of site-specific alarm information defaults.

File Name

The file name format is:

```
<module><-subspec>-d.def
```

For example:

```
solaris-example-d.def
```

Note – The alarm file (`*.def`) is sometimes interchageably called the default file.

Contents

The alarm file, which is in the module configuration file format mimics the same tree structure specified in its corresponding agent file and contains entries only for those nodes with alarm specifications.

If alarm file is left empty, the refresh operations are still executed and managed property data is aquired, but the managed properties never goes into alarm.

The alarm file can specify the following alarm management related qualifiers for any of the managed property nodes in the agent file tree structure:

```
alarmChecks = <alarm check1> < alarm check2)
alarmlimit:<alarm check> = <alarm limit>
alarmSeverity = <integer [0-9]>
alarmWindow = <Alarm window timex specification >

statusActions(<alarm event>)   = <action1> <action2>...
statusService(<action1>)       = <service>
statusCommand(<action1>)       = <command>
statusService(<action2>)       = <service>
statusCommand(<action2>)       = <command>
```

where:

<alarm check> is a specification of an alarm check that has the following format *<alarm state>-<alarm test>*. The sections, "Alarm Checks" and "Specifying Alarm Limits," describe this specification.

<alarm limit> specifies the threshold criterion for this check. The sections "Alarm Checks" and "Specifying Alarm Limits" describe this specification.

<actionN> specifies a logical name of an action to be executed.

<service> specifies an execution context for the command to be run.

<command> specifies a command to execute.

Specifying the Alarm Criteria

Specifying Alarm Checks

The actual alarm checks that are performed when using the `rCompare` (using the standard alarm types) rule are specified by the `alarmChecks` qualifier. The `alarmChecks` qualifier does not typically have to be specified in the alarm file since every alarm type already defines an appropriate set of default alarm checks.

In general, the alarm checks specified by the alarm type primitives, which are used by managed properties, are adequate. However, if a different set of alarm checks must be specified for a managed property, the `alarmChecks` qualifier can be used to override the default alarm checks specification.

```
alarmChecks = <alarm check> [<alarm check2> ... ]
```

where

<alarm check> is an alarm check specification of the form: *<alarm state>-<alarm test>*

Possible alarm states are `info`, `warning`, or `error`.

The following sections describe the possible alarm tests and alarm checks.

Alarm Checks

The standard alarm types and their default alarm checks are:

HI—alarmChecks = error-gt warning-gt info-gt

LO—alarmChecks = error-lt warning-lt info-lt

HILO—alarmChecks = error-gt error-lt warning-gt warning-lt info-gt info-lt

EQ—alarmChecks = error-eq warning-eq info-eq

NE—alarmChecks = error-ne warning-ne info-ne

REGEXP—alarmChecks = error-rx warning-rx info-rx

Alarm checks are performed in the order they are listed. Thus, alarm checks should be listed from highest to lowest alarm severity.

For example, the HI alarm type primitive defines the following alarm checks:

```
alarmChecks = error-gt warning-gt info-gt
```

This indicates that three alarm checks may be performed. The first check, `error-gt`, tests whether the data value is greater than the corresponding alarm limit. If the test is positive, the managed property is given the error alarm state and no more alarm checks are performed.

If the first check is negative, the second check, `warning-gt`, tests whether the data value is greater than the corresponding alarm limit. If the test is positive, the managed property is given the warning alarm state and no further alarm checks are performed.

If the second check is negative, the last check, `info-gt`, tests whether the data value is greater than the corresponding alarm limit. If the test is positive, the managed property is given the `info` alarm state. Otherwise, the managed property is given the `ok` state.

Specifying Alarm Limits

An alarm limit can be specified for each alarm check defined for the managed property. Alarm limits can be specified for managed properties whose data values are either scalars or vectors.

If no alarm limits are specified for a managed property, then the alarm checks are not performed for that managed property.

Scalars

Alarm limits are specified for scalars as follows:

```
alarmlimit:<alarm check> = <alarm limit or criteria>
```

where:

<alarm check> is an alarm check specification of the form: *<alarm state>-<alarm test>*.

Solaris Example—Scalar Alarm Limit

The `cpu.busy` managed property is assigned the `FLOATHI` data and alarm type. Since it does not override the alarm checks, it uses the default alarm checks specified in the `FLOATHI` primitive.

```
cpu = {
    busy = {
        alarmlimit:error-gt   = 95
        alarmlimit:warning-gt = 90
        alarmlimit:info-gt    =
    }
}
```

Vectors

Alarm limits can also be specified for a table of managed properties whose data values are vectors. An alarm limit can be specified for each vector element by qualifying the alarm limit with the *rowname*. The rowname is used as an index to identify rows in the table. The managed property designated to be the rowname is specified in the Model file using the `index` qualifier.

In addition, default alarm limits can be specified for vector elements that do not have explicitly defined alarm limits.

Alarm limits are specified for vectors as follows:

```
alarmlimit:<alarm check>() = <default alarm limit>
alarmlimit:<alarm check>(<rowname>) = <alarm limit for row element>
```

where:

<alarm check> is an alarm check specification of the form: *<alarm state>*-*<alarm test>*.

<rowname> is the data value used to identify the row in the table. The column is specified by the `index` qualifier.

Solaris Example—Vector Alarm Limit

The following example demonstrates the specification of alarm limits for the `avail` managed property in the `filesystems` table of the Solaris Example module. The `avail` managed property is the amount of available disk space.

Default `error`, `warning`, and `info` alarm limits for the amount of available disk space in a file system are specified by the alarm limit entries qualified by `()`. These default alarm limits are applied to filesystems that do not have explicitly set alarm limits.

The `mount` managed property was designated as `index`; hence, its values are used as the rowname. Thus, the `mount` name of file systems is used to reference specific rows.

Alarm limits for the amount of available disk space for the `/usr` filesystem are specified by the alarm limits entries qualified by `(/usr)`.

```
filesystems = {
    fileTable = {
        fileEntry = {
            avail = {
                alarmlimit:error-lt()       = 7000
                alarmlimit:warning-lt()     = 12000
                alarmlimit:info-lt()        =
                alarmlimit:error-lt(/usr)   = 5000
                alarmlimit:warning-lt(/usr) = 10000
                alarmlimit:info-lt(/usr)    =
            }
        }
    }
}
```

Alarm Severities

The alarm severity provides additional granularity for the ranking of alarms within each alarm state. You can prioritize alarms associated with specific managed properties relative to the alarms of other managed properties within the same alarm state by setting the `alarmSeverity` qualifier.

```
alarmSeverity = <integer>
```

The `alarmSeverity` can be set to an integer value ranging from 0 to 9. The greater the number, the higher the alarm rank. TABLE 7-1 lists the default alarm severities.

TABLE 7-1 Alarm Severities

Alarm State	State Value	Default Severity
OK	0	0
OFF	0	1
DIS	0	1
INF	0	5
WRN	1	5
ERR	2	5
IRR	2	7
DWN	2	9

Solaris Example—CPU Alarm Severity

The `cpu average` managed property is assigned a higher alarm severity than the `cpu busy` managed property since the average CPU is of greater significance than instantaneous CPU measurements.

Thus, if both the `busy` and `average` managed properties go into the error state, the average alarm would be ranked higher.

If the `busy` goes into `error` while the `average` property goes into `warning`, the `busy` alarm would be ranked higher.

The `cpu busy` and `average` alarm information with the specification of their relative severities are:

```
cpu = {
    busy = {
        alarmSeverity         = 3
        alarmlimit:error-lt   = 95
        alarmlimit:warning-lt = 90
        alarmlimit:info-lt    =
    }

    average = {
        alarmSeverity         = 7
        alarmlimit:error-gt   = 95
        alarmlimit:warning-gt = 90
        alarmlimit:info-gt    =
    }
}
```

Alarm Window

Alarm checking can be set to be active only at particular times for a managed property by specifying the `alarmWindow` qualifier.

Scalars:
`alarmWindow = < time specification >`

Vectors:
`alaraWindow() = < time specification>`
`alarmWindow(<`*rowname*`>) = < time specification>`

where

<*rowname*> is the data value used to identify the row in the table. The column is specified by the index qualifier.

The `alarmWindow` can be set to any valid time specification window, and specifies the time window during which alarm checking is performed. At times outside the window, the alarm checks are not executed at all. If no alarm window is specified, alarms checks are by default always done.

Solaris Example—CPU Alarm Window

In the following example, the `cpu busy` time alarm window is set so that alarms can only be generated between 1:00 in the afternoon and midnight. At other times, alarm checks are not performed.

```
cpu = {
    busy = {
        alarmWindow        = time>13:00
        alarmlimit:error-lt   = 95
        alarmlimit:warning-lt = 90
        alarmlimit:info-lt    =
    }
}
```

Specifying Status Actions

When the alarm state of a managed property changes, an *alarm event* is generated. These alarm events can be used to trigger actions to perform pro-active or remedial actions based on the detected alarm condition. These actions are referred to as *status actions* and can be specified in the alarm file:

```
statusActions(<a21larm event>)   = <action1> <action2> ...
statusService(<action1>)       = <service>
statusCommand(<action1>)       = <command>
statusService(<action2>)       = <service>
statusCommand(<action2>)       = <command>
```

where:

<*actionN*> specifies a logical name of an action to be executed.

<*service*> specifies an execution context for the command to be run.

<*command*> specifies a command to execute.

An event is generated for a managed property whenever the alarm state of the managed property changes. Possible <*alarm event*> values include:

`init`—when tree is initialized

`change`—on any alarm state change

`ok`—when alarm state goes to the `ok` state

`irr`—when alarm state goes to irrational state, this is typically caused by a data acquisition error

`down-eq`—when an entire module changes to the down state (such as a database being unavailable)

`off-eq`—when an entire module turns off as scheduled (through the module active time window qualifier)

`disabled-eq`—when an entire module is disabled manually by a an end-user at the console

`info-`<*alarm test*>—when the info alarm check is satisfied

`warning-`<*alarm test*>—when warning alarm check is satisfied

`error-`<*alarm test*>—when error alarm check is satisfied

The <*alarm test*> can be `lt`, `gt`, `eq`, `ne`, or `rx`, depending on the alarm type primitive used by the managed property.

Solaris Example—CPU Status Action

In the following example, the 'help' message is sent to the console when the `cpu busy` time goes into the error state.

```
cpu = {
    busy = {
        statusActions(error-lt) = sayhello
        statusService(sayhello) = _services.sh
        statusCommand(sayhello) = echo "hello"> /dev/console
        alarmlimit:error-lt   = 95
        alarmlimit:warning-lt =
        alarmlimit:info-lt    =
    }
}
```

CHAPTER 8

Rules

This chapter covers the following topics:

Detection of alarm conditions and subsequent triggering of actions is the basic function of the Sun Management Center 2.1 framework program. Alarm conditions are determined by:

- Simple alarm checking (for example, threshold checks and regular expression matching)
- Rule evaluation

Both mechanisms achieve the same purpose. To provide comprehensive alarm capabilities, Sun Management Center 2.1 supports both mechanisms. However, this chapter focuses only on the Rule evaluation.

Rules can be considered an arbitrarily complex type of alarm check, which normally depends on other objects (that is, a rule is usually associated with a derived object.) In the version 2.1 agent rules are implemented through an extension of alarm checking of derived objects. Thus, rules can be treated as another alarm check mechanism.

Currently no industry standard exists for rule syntax; thus, rules must be introduced into the agent based on how they are written. However, there is a consistent convention for specifying which rule is to be fired in the agent configuration file (module configuration file format). Storage of any state or persistent data required by a rule is provided in the object that invokes the rule.

Rules Agent Infrastructure

This section covers the rules agent infrastructure.

Rules and Derived Objects

Rules and derived objects are inherently related. For the Sun Management Center framework, a rule is implemented in a derived object; consequently, there is a one-to-one relationship between a rule and a derived object.

Rule Naming

Each rule must be named as:

```
r<n>
```

For example: `r231` represents rule 231.

Module designers can create custom rules that use a wide variety of alarm criteria. These custom rules can examine the value of the node to which the rule is attached, or the value or status of a different node. A special category of rules, referred to as *log rules*, can be triggered to fire whenever a message matching a specified regular expression appears in a log file.

Rules usually use parameters that are stored in separate files where they can be customized on a per-machine basis by site administrators. Certain rule parameters can be declared editable by end users through the Sun Management Center console.

Rule Assignment

Because a rule is considered a complex alarm check, it is natural to extend the existing agent alarm checking mechanism to encompass rules. A qualifier, *<alarmRules>*, specifies a particular rule for a given node. This variable is normally assigned to a node in the module `Model` file.

```
alarmRules = r231
```

Whenrever a rule is used, no other alarm check is allowed. If the agent detects both an `alarmChecks` and a valid rule in the <*alarmRules*> variable for a given node, only <*alarmRules*> is used in status determination; the `alarmChecks` are ignored.

When an agent encounters *alarmRules b*alarmRules, it invokes the rule directly using a `ruleFire` procedure, which is described later in this section.

Note – A log rule, described later in this chapter, can also be invoked by the file scan service through `ruleFire`, after the rule has been subscribed to that service (see the “Rules Attributes” on page 124” for a description of `ruleFire`).

If the node is a vector (that is, it can have more than one row), there is no change in the preceding discussion. However, internally, the rule specified in <*alarmRules*> is available to each row in the vector node and any data stored in slices is distinct for each `rowName`.

Rule Files

The program provides module specific rules, general rules or base rules, and rules created by clients.

Module-Specific Rules

Tcl rules associated with a particular module are placed within the file:

```
<module><-subspec>-d.rul
```

Parameter definitions and message text definitions for such rules go in the associated files:

```
<module><-subspec>-ruleinit-d.x
<module><-subspec>-ruletext-d.x
```

Any custom module rule must be made available within the context of the node that requires it. Here is an example of what you enter into the model file to achieve this:

```
_rules = { [ use PROC ]
    [ source solaris-example-d.rul ]
}

system = { [ use MANAGED-OBJECT ]
      consoleUser = { [ use STRINGRULE MANAGED-PROPERTY _rules ]
        alarmRules = rUsrChk
    }
}
```

In this example, the `rUsrChk` rule is associated with the `consoleUser` node. This rule determines the state of the `consoleUser` node.

General Rules or Base Rules

Rules that are more general in nature, and that can be used in many different modules, should be placed in the `base-rules.rul` file. To reduce overhead, only those rules that must be globally accessible should be made base rules.

Parameter definitions and message text definitions for such rules go in the associated `base-ruleinit-d.x and base-ruletext-d.x files.` Rules placed in these files are automatically available within the context of all nodes and do not have to be sourced explicitly.

An example of how to attach a rule that is defined `base-rules.rul` is:

```
system = { [ use MANAGED-OBJECT ]
      consoleUser = { [ use STRINGRULE MANAGED-PROPERTY ]
        alarmRules = rGenRule
    }
}
```

Note – Base text messages are *not* loaded into a `_rules node`; these messages are added directly to the appropriate file.

Rules Created By Clients

In the future, clients may wish to create their own custom rules. Such rules should be placed in the file:

```
user-rules.rul
```

Parameter definitions and message text definitions for such rules go in the associated files:

```
user-ruleinit-d.x
user-ruletext-d.x
```

Placing client rules in these separate files, allows the client rules to be saved readily across new code releases.

Note – The purpose of these files is to segregate possible client customizations from the base distributed code. The wider issues of code management, configuration, and distribution are outside the scope of this document.

As in the case of `base-rules.rul`, rules placed in `user-rules.rul` are automatically available within the context of all nodes and do not have to be explicitly sourced. They are globally accessible.

Rule Placement in Hierarchy

Because a rule is connected to a node through the <*alarmRules*> variable, it is obvious that a rule must be associated with a node. However, some special circumstances require attention:

- A node can require more than one rule.
- A rule can have no natural node to which to be attached.
- A node can have a rule but no data.

Each of these cases is described in the following sections.

A Node Can Require More Than One Rule

An example of this multiple rule requirement can be taken from the rules. Rules `rcr4u209`, `rcr4u212`, and `rcr4u300` all apply to memory SIMMs. In this case, if a module hierarchy has a node for a particular SIMM, for example, J3201, which has a leaf node for its status, these three rules cannot be associated with the leaf nodes *<alarmRules>* variable because only one rule is allowed.

The solution is to create three more leaf nodes as inferiors of J3201, with one rule per node. By doing so, hierarchical summarization of status up to the SIMM node is handled by the existing agent status propagation mechanism.

Alternatively, if possible, the three rules can be redesigned and collapsed into a single rule.

Rule Can Have No Natural Node to be Attached to

Several existing rules (for example, rules `rknrd105`,) do not drive any alarms (that is, they do not affect the alarm status of any node). Instead, they simply generate events. Normally, a rule is attached to the node that it alarms. In this case, a node must be created specifically for hosting the rule.

It would be wise to collect all such orphan nodes together into a single rules-only module.

Node Can Have a Rule but No Data

In the preceding sections, nodes have been introduced to host a rule. Such nodes have no real data associated with them. To identify such nodes, a new primitive, `RULE`, must be introduced.

Rules Attributes

Rule Data Storage

This section describes the Tcl and C/C++ compiled rules.

Note – Wherever method names are mentioned, the method should be implemented in two forms: as a Tcl/TOE method and as a C/C++ function).

Rules have four types of variables (also referred to as *attributes* or `parameters` in this description):

- **Temporary**: These variables are local (known only to the rule procedure proper); an example would be a loop counter like `loopCnt`. The rule procedure has read/write access to such variables. The values of the variables are strictly temporary and are not retained across invocations of the rule.
- **Dynamic**: These variables are local (known only to the rule procedure proper). The rule procedure has read/write access to such variables, and the variable values are retained across subsequent invocations of the rule. The rule can use such parameters to save information that is needed between one execution of the rule and the next.
- **Static**: These variables are read only. They are initialized in the `ruleinit` file. An example would be the rule group that a rule designer assigns to indicate that a rule is part of a larger group, like hardware or capacity planning. The rule procedure can read, but not modify, these variables.
- **Editable**: These variables are read-only to the rule procedure. An example of an editable parameter would be `swap_thresh`, which represents a threshold that a user can choose to tune. Global default values for such variables are defined in the `ruleinit` file, and those global defaults *cannot* be modified directly by end users. However, end users *can* specify local override values on an individual node/row basis. An end user makes such changes through the Sun Management Center console, and the modified values are stored in a <*module*><*-subspec*>`.dat` override file for the module, so that the overrides are saved across restarts of the agent. The rule procedure can read, but not modify, the editable parameters.

Except for the first case of temporary variables, all other variables must be stored in a TOE slice, and accessed through the `getRuleParm/setRuleParm` (see *"Rules Functions"* for the Tcl/TOE implementation). Each node must have its own copy of whatever slices it requires whenever node-specific data is created:

TABLE 8-1 Rule Variables

Variable Type	Scope	Initialization	Persist Across	Slice	Read Access	Write Access
temporary	local	`rule proc`	-	-	Tcl ${name}	Tcl set
dynamic	local	`rule proc`	rule invocations	rule-dyn	getRuleParm	setRuleParm
static	global	`ruleint` file	Agent restarts	rule	getRuleParm	-
editable	global	`ruleint` file	Agent restarts	alarmlimit	getRuleParm	local overrides via shadowmap

Another slice, `rulemsg`, must be created once for each loaded module, under the `node _rules`. This slice must contain a key-value pair for every text message.

Note – The `rulemsg` key name does NOT require a <*ruleId*> prefix, and normally should NOT have a prefix, since messages can apply to multiple rules.

TABLE 8-2 Rule Message Key

`rulemsg` key	Usage
<*msgId*>	Contains a string defining the specified message (for example, `ir209msg`)

Sun Management Center 2.1 has no restriction on what data or how much data is saved by a rule between invocations. The name and usage of such data is strictly rule-specific (that is, up to the rule designer), and is accessed through the `getRuleParm`/`setRuleParm` methods (see “Rule Functions” on page 129 for the Tcl/TOE implementation).

Note – The internal data is *always* available for every rule. It is maintained transparently by the underlying rule implementation). *This data must never be modified by a rule designer.* The rule designer has read-only access to certain internal data as shown in the next table.

TABLE 8-3 Rule Designer Access to Internal Data

Slice	Key	Description	Equivalent	Rule Read Access
rule-state	$rule[($row)]	Last rule state (for example, 0=init, 1=open, 2=continue and 3=close)	ACTIVE	ruleActive
rule-cond	$rule[($row)]	ast value of rule event status (blank for ok, info, warning or error	n/a	n/a
`rule-count`	$rule[($row)]	Number of consecutive iterations of the rule, counting from the last time it transitioned to open; (for example, a value of 3 means the rule has detected a true `CONDITION` for the last 3 iterations).	COUNT	getRuleCount
`rule-statetime`	-	Time of the last transition in rule-state.	n/a	n/a
`rule-starttime`	-	Time the rule `CONDITION` last transitioned from false to true.	START_TIME	getRuleStartTime

TABLE 8-3 Rule Designer Access to Internal Data *(Continued)*

`rule-fs`	-	Present for log rules only. Contains a list of File Scan service subscriptions made by the rule.	n/a	n/a
`rule-match`	-	List of pattern matches from File Scan service, saved for subsequent log rule invocation.	n/a	Passed to rule as parameter to “open” or “continue” action
`rule`	$rule-editparm	Present only if editable parameters have been identified by a rule; contains a list of the editable parameters.	n/a	n/a

Rule State Transitions

To generate alarms, a rule defines a CONDITION (in Tcl rules this is the Tcl script in the condition case) that is evaluated every time the rule is executed. If the condition is true, this is an active event.

An event is an alarm generated by a rule. An inactive event is equivalent to saying that there is no alarm generated for a particular rule. Events can transition through various states; the underlying rule determines these transitions. TABLE 8-4 lists the allowed event states and transitions.

TABLE 8-4 Rule State Transistions and Events

State	Meaning	May Transition To
init	Rule is initialized, ready to detect event transitions; event is inactive.	open
open	Rule condition has just transitioned from false to true; event is now active.	continue, close
continue	Rule condition was true in the last invocation, and is still true in the current invocation; event is still active.	continue, close
close	Rule condition was true in the last invocation, and is now false in the current invocation; event is now inactive.	open

The `ruleFire` procedure determines these states. The preceding section describes the state variables that persist between rule invocations.

Of these are two special operations that can be performed on an active (that is, open or continue state) event; these operations are `ack` and `fix`. Presently, neither operation causes an event state transition; thus, if a rule detects either operation in effect, it executes the corresponding Tcl script specified in the rule, and returns with an empty string ("").

The ack operation signifies a user action to acknowledge an event; the event remains active.

The fix operation signifies a user action to manually repair a hardware-related event; whether this affects the event state isto be determined.

Rule Invocation Procedure (ruleFire)

All rules must be coded as Tcl procedures (or C/C++ functions) that are invoked by the agent through the ruleFire procedure. This procedure must do the following:

1. Invoke the specified rule procedure, specifying an init action if initialization is required, otherwise, perform the steps that follow. Init is read the first time the rule is read. The rule remains in the Init state for as long as the rule condition evaluates to false.
2. Invoke the specified rule procedure to determine whether the event condition is true or false (as determined by the CONDITION script of the rule, described later in this document).
3. Determine the current rule/event state (for example, open, continue, close), based on the evaluated condition.
4. The following state transitions can occur based on evaluating the condition and the current rule state:
   - If condition is false, and if event is inactive, no new event is generated and the rule continues to be in init state.
   - If condition is true, and if event is inactive, the event transitions to the open state (which means that the event is now active). This could result in an info, warning, or error event status.
   - If condition is true, and if event is already active, the event transitions to the continue state (which means that the event is still active and continues to have an info, warning, or error status).
   - If condition is false, and if event is active, the event transitions to a close state (which means that the event is now inactive).
5. Invoke the specified rule procedure, whenever an end-user performs an ack or fix action on a currently active alarm.
6. Perform internal processing to maintain the state of the rule and related events.

Rule Event Status

A rule must return a string value that corresponds to a valid event status. This string must then be used by the agent. TABLE 8-5 lists the allowed return strings:

TABLE 8-5 Rule Event Status

Event Status	Meaning
ok	Inactive event
info[-<*qualifier*>]	Informational event
warning[-<*qualifier*>]	Warning event
error[-<*qualifier*>]	Error event

Note – The mandatory portion of the return string (`ok`, `info`, `warning`, and `error`) must be used by the console to determine the icon.

The optional <*qualifier*> allows additional descriptive text to be appended. The qualifier can be used to differentiate events (for example, `error-temp`, `error-parity`). The qualifier is a maximum of eight characters.

Examples of valid return strings:

```
info
warning-rx
```

Rule Functions

This section lists the methods that Tcl rules can call.

TABLE 8-6 summarizes all methods that rules can call directly.

TABLE 8-6 Rule Functions

Tcl Rule Method	Arguments	Description
closeEvent	ruleId, rowName, estatus	Closes an event (see also `logEvent`).

TABLE 8-6 Rule Functions *(Continued)*

externalTableOK	ruleid, nodeId, [default]	Return a 1 if specified table has an active alarm; 0 if no active alarm; default if alarm status of specified table cannot be acquired.
getExternalRowStatus	ruleId, nodeId, rowName, [default]	Get status of specified vector row of specified node.
getExternalRowValue	ruleId, nodeId, rowName, [default]	Get value of specified vector row of specified node.
getExternalStatus	ruleId, nodeId [default]	Get status of specified node.
getExternalValue	ruleId, nodeId [default]	Get value of specified node.
getGRuleParm	parm, [default_parm]	Get global rule parameter.
getMyStatus	ruleId, rowName, [default_value]	Get current node status (that is, error-gt); default value returned if status cannot be acquired.
getMyValue	ruleId, rowName, [default_value]	Get current node value; default value returned if data cannot be acquired.
getRuleCount	ruleId, rowName	Get number of consecutive iterations of the rule, counting from the last time it transitioned to open (for example, a value of 3 means the rule has detected a true `CONDITION` for the last 3 iterations).
getRuleMsg	msgId, [default]	Get specified message.
getNodeName	level	Get portion of superior node name (level 0 means current node, level 1 means immediate superior, and so forth)
getRuleParm	type, ruleId, rowName, parm, [default_parm]	Get rule parameter. <*type*> is one of: DYN (dynamic parameter) STATIC (static parameter) EDIT (editable parameter)
getRuleStartTime	ruleid, rowname	Get the time (# of seconds since epoch) that rule last transitioned to open.
getTime		Get current time as seconds since the epoch.

TABLE 8-6 Rule Functions *(Continued)*

logEvent	ruleId, rowName, estatus	Logs an event. Event is logged as opening and immediately closing, but the node does not go into the open alarm state.
logSubscribe	ruleId, file, pattern, rowName, callback option	Log rule subscription to File Scan service. The pattern shall be enclosed in double quotes, and the callback shall be identical to the ruleId. Escape ([)_and (]) with backslashes (\).
ruleActive	ruleId, rowName	Return a 1 if rule is current active (OPEN or CONTINUE state)
setGRuleParm	parm, newval	Set global rule parameter.
setRuleParm	type, ruleId, rowName, parm, newval	Set rule parameter. Valid only for <*type*> = DYN (dynamic)
setRuleText	ruleid, rowname, estatus, shortmsg, longmsg	Set text message for indicated state for underlying node/row. longmsg is obsolete.

Third Party Rule Engine Interface Functions

Rule Loading

Rules are usually implemented in the form of Tcl procedures. If necessary, for performance reasons, rules can also be created as C or C++ code.

As an example, an agent configuration file in module configuration file format loads Tcl procedures from file `pfrules-d.prc`:

```
#
# Load pfrules procedures
    #
_procedures = { [use PROC ]
    [ source pfrules-d.prc ]
}
```

Any rules written in C or C++ must be loaded as packages.

As an example, an agent configuration file in module configuration file format loads package `pkgrules.so`:

```
#
# Load rules package
#
[ requires package rules ]
```

The rules that actually are loaded into the agent must return a string indicating the detected rule state.

Rule Assignment

The actual rule to be invoked for a derived object must be assigned through the specification of the following refresh variables:

```
refreshTrigger = <node>[:<event>]
refreshCommand = r<n>
refreshService = _internal
```

The effect of the `refreshCommand` is to invoke the specified rule, which has already been loaded either as a Tcl procedure or a C/C++ package (see “Rule Syntax & Loading”).

Key TOE Functions

Function prototypes,typeddefs and so forth required to interface with TOE must be available in header file, `sdk/include/toeInt.h`. The convention for describing each function is to consider the first argument as *arg1*, the second argument as *arg2*, and so forth. The following TOE functions are listed to provide an indication of how a rule can access agent object data:

TABLE 8-7 Key TOE Functions

TOE Function	Description
`ToePtr toeGetCurrentObject (Tcl_Interp*, int)`	Returns pointer to current TOE object (for example, the object from which the rule has fired), or NULL if error. *arg1* - pointer to interpreter handle *arg2* - literal value of TOE_DICTIONARY, to access the object data (as opposed to the commands).
`ToeVal toeDefine (ToePtr, char*,char*,DataType*,void*)`	Defines value in specified object's slice. Returns pointer to allocated value or NULL if error. *arg1* - pointer to TOE object *arg2* - pointer to slice arg3 - pointer to slice entry arg4 - pointer to data type of value Header file `converters.h` defines `dataGetType` to get a data type: `DataType * dataGetType( char * name );` You can use these data type names in program: `int, uint, long, longlong, ulong, ulonglong, short, double, float, char, boolean, string, pointer`. arg5 - pointer to value
`int toeUndefine (ToePtr,char*,char*)`	Removes entry from specified object's slice. Returns 0 on success, -1 on failure. *arg1* - pointer to TOE object *arg2* - pointer to slice *arg3* - pointer to slice entry
`ToeVal toeLookup (ToePtr,int,char*,char*)`	Returns pointer to value from specified object's slice, entry, or NULL if error. *arg1* - pointer to TOE object *arg2* - tree search scope (`TOE_SCOPE_INSTANCE, TOE_SCOPE_PARENTS,TOE_SCOPE_SUPERIORS,TOE_SCOPE_ALL`) *arg3* - pointer to slice *arg4* - pointer to slice entry
`DataType *toeValGetDataType (ToeVal)`	Returns pointer to data type structure for specified pointer to value, or NULL on error. *arg1* - pointer to object slice,entry

TABLE 8-7 Key TOE Functions *(Continued)*

TOE Function	Description
`void *toeValGetType (ToeVal,DataType*)`	Returns pointer to data, or NULL on error. Programer cannot free the pointer, because the memory is maintained by TOE. *arg1* - pointer to object slice, entry *arg2* - pointer to type to return (NULL if string)
`int toeValSetType (ToeVal,DataType*,void*)`	Sets specified value into specified object's slice, entry. Returns CVT_OK on success, CVT_ERROR on failure. *arg1* - pointer to object slice, entry *arg2* - pointer to data type structure for arg1; normally set to *arg1*'s `type` member (for example, arg1`->type`) *arg3* - pointer to value

How to Write A Tcl Rule

In Sun Management Center 2.1, rules are implemented as Tcl procedures, which provide the following advantages:

- Rules can be introduced to an agent directly, without parsing.
- Rules can be invoked directly through the ruleFire procedure.
- Rule syntax becomes more rigorous, since it must conform to the Tcl language.

This section provides a guide to writing a Tcl rule by describing the following:

- An example of a Tcl rule and rule file
- Guidelines for writing a Tcl rule, including a template of a rule procedure
- Other rule support issues (sourcing a rule file, specifying initialization data and rule messages, specifying editable parameters)

The examples in this section all pertain to the ConfigReader example introduced earlier. A rule designer can customize the example code to create the files needed to add new rules.

The *principles* in this section apply to C/C++ compiled rules as well, since the basic structure of a rule is simply a procedure with a switch statement to allow the appropriate code to be invoked to handle each case. A natural way to proceed is to port the Tcl methods introduced in this section to C/C++; these methods constitute the interface between a C/C++ rule and the agent. The C/C++ rules would be made available to the agent by creating Tcl packages.

Tcl Rule Example

Methods referenced in the example (for example, `getRuleParm`) are described in “Rule Functions” on page 129.

CODE EXAMPLE 8-1 Tcl Rule Example

```
# Rule:  rcr4u209
# Purpose:
# Log a message for SIMM errors with corrected ECC Data Bit
# Arguments:
#    action – one of
#      {condition|init|open|continue|ack|fix|close}
# ruleId – rule identifier
# rowName – name of vector row (only valid for vector nodes);
#       defaults to “”
# rowIndex – index of vector row (only valid for vector
#        nodes); defaults to 0
# matchList – set to matched substrings if this is a
#        log rule and the File Scan service is
#        notifying the rule about matches found;
#        defaults to “”
# This rule is attached to the ConfigReader...memory.SIMM
# hierarchy.
#
   proc rcr4u209 { action ruleId { rowName “” } { rowIndex 0} \
{ matchList “” } } {
   set estatus “error”

   # Fire state transition actions
    switch $action {
     init {
# Get names of board node & boardrefno node (e.g. SIMM(0), J3201)
    set node_logword1 [ getNodeName 2 ]
    set node_logword2 [ getNodeName 1 ]

# Derive values of logword1, logword2 from node names (e.g. 0, J3201)
    regexp {([0-9+])} $node_logword1 logword1
    regexp {([0-9+])} $node_logword2 logword2

    logSubscribe $ruleId /var/adm/messages “ECC \[Mm\]emory \ \[Ee\]rror.*SIMM\
Board ($logword1) ($logword2).*ECC Data Bit (\[0-9\]+) was corrected” $rowName\
$ruleId
    return “”
  }
  open
    set boardno [ lindex $matchList 0 ]
    set boardrefno [ lindex $matchList 1 ]
```

CODE EXAMPLE 8-1 Tcl Rule Example *(Continued)*

```
    set bitno [ lindex $matchList 2 ]
    set nodeName [ getNodeName 0 ]
# Note that reference to $target is here only to mimic the SyMON 1.x
# rule r209; global parameters that have to be available to all rules
# in a module should be set through get/setRuleGParm
       set msg [ format[ getRuleMsg ir209msg ]"$target"
             "$nodeName" "$bitno" ]
        setRuleText $ruleId $rowName $estatus $msg
# Note that logEvent is called here to mimic the SyMON 1.x
# behaviour of log rules (i.e. an event is opened and then
# closed immediately); however, it is possible and advisable to
# not call logEvent at all, and to instead return $estatus instead
# of returning "ok". That will cause the event to remain open until # closed
(e.g. by a Fix action);

      logEvent  $ruleId $rowName $estatus
    return ok
    }
    fix {
# Any special “fix” actions would go here
    return
    }
  }
}
```

Tcl Rules File Format

All Tcl rule files must have an extension of ‘rul’ (for example, `config-reader-d.rul`). A portion of a rule file for the running example in this section would look like this:

CODE EXAMPLE 8-2 Tcl rules File Format

```
# File: config-reader-d.rul
#
# configReader rules
#
    proc rcr4u209 {action ruleId {rowName ""} {rowIndex 0} \
{matchList ""}} {
    body of procedure goes here ...
}
    proc rcr4u212 {action ruleId {rowName ""} {rowIndex 0} \
{matchList “”}} {
    body of procedure goes here ...
}
```

CODE EXAMPLE 8-2 Tcl rules File Format

```
    proc rcr4u300 {action ruleId {rowName ""} {rowIndex 0} \
{matchList “”}} {
    body of procedure goes here ...
}
```

Tcl Rule Template

The template in this section shows the minimum elements that must exist in every Tcl rule. Examples of all required files, based on the running example of `ConfigReader` rule `rcr4u209`), follow.

The major steps required to create a rule are listed in the section below.

Guidelines

1. Determine the name for the rule (for example, `rcr4u209`). The name must be unique across all rules.

2. Determine the initial values for any static (for example, group), or editable parameters required by the rule that will be stored in a slice. Create initial values for any of these parameters by adding them to a rule parameter initialization file named after the module that the rules belong to (for example, `config-reader-ruleinit-d.x`). The load of this file must be done at the bottom of the Model file.

   If any of the preceding parameters must be editable, a special static parameter, `editparm`, must be initialized in the <*module*><*-subspec*>`-ruleinit-d.x` file to contain a list of all parameters that are to be editable. For example, if parameters `alarmThresh` and deadband were to be editable in some rule r*XXX*, `editparm` would be initialized by

   ```
   _rule = {
        rule:rXXX-editparm = alarmThresh deadband
   }
   ```

   where XXX can be any name with or without numbers.

Note – The listed editable parameters *do not* have the rule identifier as a prefix.

For the console, the displayed text describing the editable parameter must be internationalized. To internationalize, a special static parameter, `keypath`, is used to specify the path to the module's Properties file. Normally, the properties file must be in the Sun Management Center software proto tree under classes:

```
/com/sun/symon/base/modules/<module><-subspec>.properties
```

For example, to internationalize the text for the editable parameters, include the following in the <*module*><*-subspec*>`-ruleinit-d.x` file:

```
rule:rXXX-keypath = "base.modules.<module>"
```

The module properties file must contain entries for the internationalized text for the module's editable parameters like these:

```
editAtt.rXXX.alarmThresh=<internationalized text for threshold>
editAtt.rXXX.deadband=<internationalized text for deadband>
```

You can also define datatypes for any of the editable parameters. The data types would then be enforced whenever an end-user modifies the parameters. Set an editable parameter datatype by including a line like the following in the <*module*><*-subspec*>`-ruleinit-d.x` file:

```
ruledatatype:rXXX-alarmThresh = "float"
```

If no datatype is specified, the default is string. TABLE 8-8 lists the allowed datatypes:

TABLE 8-8 Datatypes Allowed

Datatype	Description
int	integer
uint	unsigned integer
float	floating point number
string	character string (this is the default)

Note – The rule datatype definitions are same as the datatype definitions for the underlying OS.

---

3. Define a description of the rule and its parameters; this description is displayed in the Attribute Editor. The descriptions (like the preceding example of editable parameters) must be internationalized. The descriptions are specified in the same module Properties file as for the editable parameters. Typical entries would look like this:

```
editAtt.rXXX.desc=<internationalized text for rule description>
editAtt.rXXX.paramsdesc=<internationalized text for rule parameters description>
```

4. Determine any text messages (for example, `.rcr4u209msg`) that belong with the rule, and add them to a message file for the module (for example, `config-reader-ruletext-d.x`). The load of this file must be done in the module's Model file (for example, `config-reader-models-d.x`).

5. Design the Tcl code for all possible states / actions; the rule is basically a switch statement, with a case for each state / action (for example, in`it, condition, open, continue, ack, fix, close`). The case can be empty, or left out entirely, if there is no code to execute for a particular state.

Note – The code for the condition case is used to determine if the underlying event is active; therefore, the last statement in the condition case shall evaluate to 0 or 1; this is returned to the caller (`ruleFire`).

---

6. For log rules, use `logSubscribe` in the init case; specify the regular expressions in parentheses in the pattern argument to `logSubscribe`; ensure that the regular expressions are specific to the particular instance of the rule (for example, see `logword1, logword2` in the `rcr4u209` example). Note that to specify a left or right square bracket (that is, "[" or "]") in the regular expression, the bracket must be preceded with a backslash (\).

   The option argument is an optional argument. If option is specifiedas `-m`, it indicates that the pattern being subscribed for is to be applied across multiple lines. If option is not specified or is set to any value but `-m`, the corresponding pattern is applied to individual lines.

7. Only use `getRuleParm/setRuleParm` to read / write parameters that are saved between rule invocations (that is, in TOE slices). Do not use direct lookups with TOE methods.

8. Source the rules in the module configuration file through a special node, _rules (see `config-reader-models-d.x`).

9. Attach the rule to the appropriate node(s) in the module models configuration file (for example, `config-reader-models-d.x`).

The following code example includes a template.

CODE EXAMPLE 8-3 Template

```
# Rule:  <ruleId>
# Purpose:
# Arguments:
#   action – one of
#      {condition|init|open|continue|ack|fix|close}
#   ruleId – rule identifier
#    rowName – name of vector row (only valid for vector nodes); default=“”
#   rowIndex – index of vector row (only valid for vector nodes); default=0
#   matchList – set to matched substrings if this is a log rule and the
# File Scan service is notifying the rule about matches found; default=“”
#
# Notes:
#    The Tcl code for the various states may be empty, or the case left out
#    entirely, if there are no actions to be taken.
#
    proc <ruleId> {action ruleId {rowName “”} {rowIndex 0} {matchList “”}} {
# Set state transition actions
# For a log rule, condition should always be true, since
# the log rule executes only when triggered by callback from
# the File Scan service upon matching the subscribed pattern
            switch $action {
               condition {
# Make sure that the last statement in the condition Tcl script
# evaluates to 0 for false, 1 for true; this is used to
# determine if the underlying event is active (i.e true) or not.
             < code to evaluate the condition >
     return {0 | 1}
                    }
     init {
# Call logSubscribe  $ruleId <log file> <rowName> <pattern>
# <callback function>  if this is a log rule
                   < code for init >
          return
 }
          open {
# Call setRuleText <ruleId> <rowName> <pattern> <statusmsg>
#
# <callback> <option> if this is a log rule, there are two choices now:
# (a) Call logEvent to log the event, and return "ok"
# (b) Return a state-qualifier ("error-gt", for example). Do not call
#     logEvent. The event will be logged through the normal open event
#     mechanism. The event will remain open in this case.
```

CODE EXAMPLE 8-3 Template *(Continued)*

```
 < code for open >
           return <event status >
     }
        continue {

# If the event status is to remain unchanged on continuation, simply
# return the previous event status, or return "".
#
# If the event status is to change (for example, perhaps the event is
# being escalating to "error" from "warning"), then:
#
#   (a) call setRuleText to set the message text for the new state
#   (b) return the new state.
#
# If there is a previous open event, and you return a new event state,
# the previous event will close automatically.
#
# Do not call setRuleText unless you are returning a new event state also.
#
# Note that if this is a log rule, and you let a previous file match
# remain in the open state, subsequent file matches will be sent
# as "continue" rather than open. In such cases, call "logEvent"
# to log the new event (if desired). You can call "closeEvent" to
# explicitly close the previous open event (if desired).
    < code for continue >
    return <event status>
        }
ack {
    < code for ack >
    return
      }
fix {
    < code for fix >
    return
      }
close {
    < code for close >
    return
      }
    }
}
```

Attaching a Rule to the Module Configuration Files

When a module requires a rule, the appropriate rule file (for example, `config-reader-d.rul`) is sourced. Rule files must be located in the same directory as their corresponding module configuration files. The source command must be put in a container node (typically called `_rules`) in the module's Model file. The `_rules` node must be inherited by any node requiring access to the rules.

CODE EXAMPLE 8-4 Module Model File

```
# File:  config-reader-models-d.x
#
# Node _rules will contain the TOE slices for the ConfigReader
# rule initialization data and text messages.

_rules = { [ source config-reader-d.rul ] }
availability = {
    mediumDesc = Available
}

simm = { [ use MANAGED-OBJECT ]

    r209 = { [ use RULE _rules MANAGED-PROPERTY ]
              mediumDesc = Rule 209
                alarmRules = rcr4u209
    }

    r212 = { [ use RULE _rules MANAGED-PROPERTY ]
              mediumDesc = Rule 212
                 alarmRules = rcr4u212
    }

    r300 = { [ use RULE _rules MANAGED-PROPERTY ]
              mediumDesc = Rule 300
                  alarmRules = rcr4u300
    }
}

disk = {

}


cpu = {

}
```

CODE EXAMPLE 8-4 Module Model File *(Continued)*

```
    [ load config-reader-ruleinit-d.x ]
    [ load config-reader-ruletext-d.x ]
```

The Module Agent File

The `ConfigReader` module agent file, `config-reader-d.x`, might then look like this:

CODE EXAMPLE 8-5 Module Agent File

```
# File: config-reader-d.x
#
# ConfigReader module configuration file
#

[ use MANAGED-MODULE ]
[ load config-reader-m.x ]
[ requires template config-reader-models-d ]

availability = {
    refreshInterval = ...
    refreshService = ...
    refreshCommand = ...
}

memory = {
SIMM(0) = { [ use templates.config-reader-models-d.simm ]
}

    ...

    SIMM(31) = { [ use templates.config-reader-models-d.simm ]
    }
    }

# loading alarmlimit defaults now.
# config-reader-ruleinit-d.x and config-reader-ruletext-d.x
# loaded in models file

    [ load config-reader-d.def ]
```

Assigning Initial Values to Rule Parameters

The initial values for static and editable parameters are assigned in the <*module*><*-subspec*>-`ruleinit-d.x` file. Note that editable parameters are placed in the alarmlimit slice; all others go in the rule slice.

When this file is loaded into an Agent, the initialized parameters are available to be read by any rule attached to any node. If an end user modifies an editable parameter, a local slice shall be created for the affected node to contain the customized value; other nodes will not be affected.

The datatype for an editable parameter can be specified in the <*module*><*-subspec*>-`ruleinit-d.x` file (see the example below). The default datatype is string.

Dynamic parameters are not initialized in the <*module*><*-subspec*>-`ruleinit-d.x file`. Such parameters are set directly within the rule logic, either as needed, or in the init action section. Dynamic parameters are stored in a local slice for the affected nodes; their values are not available to other nodes (even other nodes running the same rule).

A portion of the initialization file for the running example might be:

```
# File: config-reader-ruleinit-d.x
#

_rules = {
     rule:rcr4u209-group = hardware
     rule:rcr4u209-version = 0.1

    rule:rXXX-group = example
    rule:rXXX-version = 0.1
    rule:rXXX-editparm = ”sample_thresh”
    alarmlimit:rXXX-sample_thresh = 0.10
    ruledatatype:rXXX-sample_thresh = “float”
    rule:rXXX-keypath = “base.modules.configReader”
}
```

Specifying Rule Text Messages

Rules can use various text messages to convey status; these messages can be collected into a file for a particular module in order to centralize the messages. Note that messages can apply to more than one rule, so there need not be any rule identifier in the message name. Messages shall be assigned to the rulemsg slice.

The text messages shall be associated with node `_rules`.

A portion of the initialization file for the running example can be as follows:

```
# File: config-reader-ruletext-d.x
#
_rules = {
    rulemsg:rcr4u209msg = “%s: %s: Error. ECC Data bit %s was
corrected”
        rulemsg:lowmsg = “has less than %s percent free space”
}
```

For every event that occurs, a rule can create two different descriptive messages:

- English Status Message
- Internationalized Status Message

English Status Message

The English status message appears in the Sun Management Center hierarchy and are also in the short message that appears in the Alarm Manager console for the event. The English status message is automatically prepended with host, module name, rowname, and medium description, so the rule designer should not include these in the message.

For example, if the rule specifies a status message such as the following:

```
has less than 2 percent free space
```

the actual message in the Alarm Manager console or the hierarchy is:

```
muskoka Solaris /export1 Filesystem has less than 2 percent free space
```

where:

host = muskoka

module = Solaris

rowname = /export1

mediumDesc = Filesystem

The message is set in within the rule logic in the open action as follows. If it is not set, it defaults to something like:

```
muskoka Solaris /export1 Filesystem rcr4u209 error
```

Internationalized Status Message

It is anticipated that the English status message will be supplemented in the future with an internationalized version of the status message. This internationalized status message will consist of a series of keywords that will be translated appropriately by the displaying console.

The internationalized messages supplement, not replace, the English status messages. That is, both types of messages will have to be created explicitly by each rule. In the much longer term, the English status messages may eventually be phased out entirely, in favor of the internationalized message.

Currently, rule designers should not attempt to format internationalized status strings.

More Examples Of Rules

CODE EXAMPLE 8-6 Simple Rule

```
#Rule:  rknrd402
#Purpose:
#Checks if available swap space drops below 10% for two hours.
#Storage of the last time CPU load was below 6 is maintained
#between rule invocations.  This parameter is initialized to
#some date in the year 2001.
#Note:
#This rule should be attached to KernelReader.mem
#
proc rknrd402 { action ruleId {rowName “”} {rowIndex 0} {matchList “”} } {

         set estatus info
# Fire state transition actions

    switch $action {
       condition {
          set value [ getExternalValue {
             $ruleId KernelReader.mem.swap_avail  “”
        } ]
     set swapb [ getExternalValue {
           $ruleId KernelReader.mem.swap_total “”
        } ]

     set cur [ getTime ]
         if { ($value/$swapb} > [ getRuleParm $ruleId swap_thresh 0.10 ] } {
      setRuleParm DYN $ruleId  $rowName user_timestamp $cur
       }
      set 1st [ getRuleParm DYN $ruleId $rowName user_timestamp 999999999 ]
         return [expr { ($cur-$1st)>14400 }]
          }
open {
         set msg [ getRuleMsg rknrd402msg ]

         setRuleText $ruleId $rowName $estatus $msg
         add_cpa SWAP $cur
         trim_cpa
         return $estatus
       }
    }
```

Config Reader Rule

See coding of (log) rule `rcr4u209` in this document.

Log Rule

In addition to the rule `rcr4u209` example, here is another:

CODE EXAMPLE 8-7 Log Rule

```
 Rule:  rknrd106
# Purpose:
#   Check for no swap space left.
#
#
 proc r106 { action ruleId {rowName “”} {rowIndex 0} {matchList “”} } {

  set estatus warning

  # Fire state transition actions

  switch $action {

    init {
     logSubscribe  $ruleId /var/adm/messages “no swap space.*pid (\[0-9\]+)}”\
$rowName $ruleId

            return
       }

       open {
             set pid [ lindex $matchList 0 ]
             set rmsg [ getRuleMsg rknrd106msg ]
             set msg [ format “$rmsg” “$pid” ]
             setRuleText $ruleId $rowName $estatus $msg
             logEvent $ruleId $rowName $estatus
             return $estatus
       }
    }
```

CHAPTER 9

Additional Specifications for a Module

This chapter covers the following sections:

Additional Parameter Specifications

The following is an example of a Solaris `m.x` file.

Example: Solaris m.x File

```
#
# Parameter file for Solaris Example module
#
[ load default-m.x ]

#
# Create multiple groups
#
consoleHint:moduleParamGroups = param misc
?misc:?description = base.modules.solaris-example:misc
#
# Specify which parameters belong to which group
#
consoleHint:moduleParams(param) = module i18nModuleName
i18nModuleDesc version enterprise i18nModuleType
consoleHint:moduleParams(misc) = instance instanceName line
rootPassword favouriteFood foodGroup
#
# Mandatory Parameters
#
param:module     = solaris-example
param:moduleName = Solaris Example
param:version    = 1.0
param:console    = solaris-example
param:moduleType = operatingSystem
param:enterprise = halcyon
param:location   =
.iso.org.dod.internet.private.enterprises.halcyon.
primealert.modules.solaris.example
param:oid        = 1.3.6.1.4.1.1242.1.2.90.1
param:desc      = This is an example module monitoring cpu, load,
and filesystem statistics.
param:i18nModuleName  = base.modules.solaris-example:moduleName
param:i18nModuleType  = base.modules.solaris-example:moduleType
param:i18nModuleDesc  = base.modules.solaris-example:moduleDesc
?param:i18nModuleName?format = i18n
?param:i18nModuleType?format = i18n
?param:i18nModuleDesc?format = i18n
#
# Additional Parameters
#
param:instance      =
param:instanceName  =
param:rootPassword  =
```

```
param:favouriteFood = base.modules.solaris\
-example:favouriteFoodValue

param:foodGroup = vegetable
?param:favouriteFood?i18n = yes

?param:instance?description = base.modules.default:instance
?param:instance?reqd        = yes
?param:instance?format      = instance

?param:instanceName?description =
base.modules.default:description
?param:instanceName?reqd        = yes
 ?param:line?format              = separator

?param:rootPassword?format      = password
?param:rootPassword?description = base.modules.\
solaris-example:rootPassword

?param:favouriteFood?description = base.modules.\
solaris-example:favouriteFood

?param:foodGroup?description = base.modules.\
solaris-example:foodGroup
?param:foodGroup?format = list:meat,base.modules.\
solaris-example:meat|
vegetable,base.modules.solaris-example:vegetable|\
fruit,base.modules.solaris-example:fruit
```

Additional Parameters

Additional parameters may be specified for the module, and are used to prompt the user for additional information when the module is loaded. For each additional parameter, this file must contain the following lines:

```
param:<parameter> = <default value>
?param:<parameter>?description = \
base.modules.<module><-subspec>:<key>
```

where:

<*parameter*> is an identifying name for the parameter. The name must be a single string with no whitespace.

<*default value*> is the default initial value for the parameter.

<*module*> is the module name.

<*subspec*> is the optional module subspec.

<*key*> is a key used to look up the internationalized string in the properties file. This key must be unique in the module. When a user loads a module, this string is displayed to the user (along with the default initial value for the parameter).

For every additional parameter there must be an additional entry in the properties file for that module. This entry is the internationalized description (`?param:<parameter>?description`) for the parameter and has the following format:

```
<key>=<internationalized text>
```

where

<*key*> is the same key used in the Parameter file for the parameter

<*internationalized text*> is the internationalized string describing the parameter

If you want to internationalize the <*default value*> of the parameter, then you must replace <*default value*> with `base.modules.`<*module*><*-subspec*>:<*key*>.

```
param:<parameter> = base.modules.<module><-subspec>:<key>
```

The corresponding line in the module properties file must also be added:

```
<key>=<internationalized text>
```

If the internationalized default value is to be editable by the user when the module is loaded, then the following is also required:

```
?param:<parameter>?i18n = yes
```

If this value is to be read-only, then `?param:<parameter>?i18n = yes` can be replaced with:

```
?param:<parameter>?format = i18n
```

or

```
?param:<parameter>?format = i18ncomment
```

The `i18ncomment` indicates that the value is a multiline entry. In this case, the *<internationalized text>* would have `\n` to indicate new line separations.

To be able to view additional parameters in the console, each optional parameter that is specified must also be added to the space separated list of the `consoleHints:moduleParams(param)` specification. The order in which the parameters are displayed in the Sun Management Center console is defined by the order in which the parameters are listed.

```
consoleHint:moduleParams(param) = module i18nModuleName
i18nModuleDesc version enterprise i18nModuleType <parameter>
```

Predefined Additional Qualifiers

Any of the following optional qualifiers can be specified for any parameter:

TABLE 9-1 Predefined Additional Qualifiers

Parameter	Values	Description
`?param:<parameter>?access =`	`ro|rw`	Specifies if the parameter is read-only (`ro`) or read-write (`rw`) access. A parameter with `rw` access can be edited by the end user when the module is loaded. This does not need to be specified when using `?i18n = yes`. Default is `rw`.
`?param:<parameter>?editaccess =`	`ro|rw`	Specifies if the parameter can be edited after the module has been loaded (there is an “Edit Module Parameters” function available to the end user from the console). Default is `rw`.
`?param:<parameter>?reqd =`	`yes|no`	`yes` specifies that the parameter is required. This means that the value must not be left blank when the module is loaded. Default is `no`.

TABLE 9-1 Predefined Additional Qualifiers *(Continued)*

Parameter	Values	Description
?param:<*parameter*>?format =	separator	Specifies a dummy parameter that is displayed as a solid line. This is used for console formatting purposes.
?param:<*parameter*>?format =	blank	This dummy parameter is displayed as a blank line. It is to be used for console formatting purposes.
?param:<*parameter*>?format =	header	The description and value of this parameter is displayed as read-only in bold and larger font.
?param:<*parameter*>?format =	comment	The value of this parameter is displayed as read-only in a multiline text format.
?param:<*parameter*>?format =	boolean	This parameter is displayed as a yes/no check box.
?param:<*parameter*>?format =	list:<*a*>|<*b*>|<*c*>	Specifies that the parameter can take a list of values. The possible values are specified as "|" separated entries. Each value in the list may include white space. The available options are displayed in a picklist. To internationalize list elements, the elements must be replaced with <*a*>,base.modules.<*module*><*subspec*>:<*keya*>. The internationalized text for <*keya*> is displayed in the console. However, the value <*a*> is set in the agent. The corresponding <*keya*> must be added to the module properties file. An example is given at the end of this Chapter.
?param:<*parameter*>?format =	password	Any value entered for this parameter is displayed as *. Also, values of such parameters are returned as a string of asterisks in response to an SNMP get request. This qualifier must be set for parameters such as database administration passwords, which must not be accessible through SNMP get.

TABLE 9-1 Predefined Additional Qualifiers *(Continued)*

Parameter	Values	Description
?param:<*parameter*>?format =	timex	The value for this parameter must conform to the agent time specification (refer to the Appendix B for more information). An Advanced button is added to facilitate entering the correct time specification.
?param:<*parameter*>?format =	timewindow	Same as timex but only comparison time specifications are allowed. This is used for module active time windows.
?param:<*parameter*>?format =	multi-line	The value of this parameter is displayed as multiline writable text.
?param:<*parameter*>?format =	width=<*pixels*>	The input field for this parameter remains at the specified fixed pixel size.
?param:<*parameter*>?format =	nodisplay	If no default value for this parameter is set, the parameter will not be displayed.
?param:<*parameter*>?format =	unicode	Allow non-ASCII text to be entered as the value.
?param:<*parameter*>?format =	instance	Specifies that the value must contain at least one alpha character, no whitespace, and no special characters such as \, $, ;, &, and *. This setting must be used for 'instance' parameters.
?param:<*parameter*>?i18n =	yes	The default value for this parameter can be edited as multi byte text.
?param:<*parameter*>?type =	int	Restricts the value entered for this parameter to be an integer.
?param:<*parameter*>?type =	float	Restricts the value entered for this parameter to be a floating point value.
?param:<*parameter*>?type =	nospace	Does not allow any spaces in the value entered for this parameter.

Note – The mandatory parameter param:location cannot be used for a module that can be loaded more than once.

Creating Multiple Instances of a Module

Instance Specification

For some types of modules, multiple instances of the module can be run simultaneously on a single host. For example, consider a module designed to monitor the health of a single printer. For a system with several printers, the printer monitoring module would be loaded multiple times, once for each printer. In that scenario, there would be several separate instances of the printer module running simultaneously.

For such modules, it is necessary to distinguish the different instances that are loaded and running (this step is optional). This is done by specifying two additional parameters in the parameter file: *instance* and *instanceName*.

The instance parameter identifies the module instance uniquely and is used internally by the agent. This is used with the <*location*> module parameter to determine the location where the module is to be loaded. The *instanceName* parameter is a description of the instance and is combined with the display name for the module (`moduleName`) for display purposes on the Sun Management Center console and in status messages.

The end user is prompted to enter values for both parameters when the module is loaded. Here is an example of some values a user might enter to distinguish two printers. The user would load the module twice, specifying different values each time.

```
First Printer:
    instance = p1
    instanceName = 4th floor

Second Printer:
     instance = p2
     instanceName = 3rd floor
```

If the module name was printer monitoring, then the display names used on the console for the two printers would be a combination of the instance name and the module name:

```
printer monitoring [ 4th floor ]
printer monitoring [ 3rd floor ]
```

To include instance and instance name parameters in a module, the following lines must be included in the parameter file:

```
param:instance =
param:instanceName =
?param:instance?description = <display label for instance entry>
?param:instance?reqd = yes
?param:instance?format = instance

?param:instanceName?description = <display label for instanceName
entry>
?param:instanceName?reqd = yes
```

Both the instance and the *instanceName* parameters must be marked as required (*?reqd=yes*).

The `instance` parameter must be qualified with *?format=instance*. This forces the value entered for instance to begin with an alpha character and contain no whitespace or special characters such as /, &, or $. The `instanceName` value entered by the end user may contain spaces.

Note – If the module can be loaded multiple times, the following specification is required in the agent file:
`consoleHint:mediumDesc = base.modules.<modules><-subspec>:moduleDetail`
The corresponding entry in the properties file woul be:
`moduleDetail = <internationalized module name>[{O}]`
`[{O}]` allows the Sun Management Center console to display the internationalized module name with instance and it is required.

Organizing Module Parameters

Each additional parameter added to the `consoleHint:moduleParams(param)` specification is added to the list of parameters shown to a user when loading the module. For organizational purposes (optional step), the parameter list may be split and shown on separate console views. To specify the number of parameter groups to be displayed, add the following to the parameter file:

```
consoleHint:moduleParamGroups = param <group> [<group> <group>
...]
?<group>:?description = base.modules.<module><-subspec>:<key>
```

where

<*group*> is the name given to each grouping of parameters.

<*module*> is the module name.

<*subspec*> is the optional module subspec.

<*key*> is the key used to identify the internationalized text describing the parameter group.

To internationalize the group name, the following line must be added to the module properties file:

```
<key>=<internationalized text>
```

where

<*key*> is the same key used in the Parameter file

<*internationalized text*> is the string that is displayed to identify the group of parameters

The preceding specification indicates that the module parameters is grouped into multiple views. The `param` group, by default, will contain the mandatory module parameters. Parameters can be added to additional groups using:

```
consoleHint:moduleParams(<group>) = <param> [ <param> <param>... ]
```

where

<*group*> is the name of the group to which the parameters are to be added.

<*param*> is the parameter name to be added to this group.

An example is provided in this chapter.

Making a Module Not Loadable

There can be a need to make a particular module not loadable in an agent. (This is optional step). For example, modules that are of serverSupport module type are not loadable. The serverSupport modules are not visible in the standard hierarchy view of the agent in the console. More importantly, modules of this type do not contribute to the overall status of the agent. This module type should only be used for modules that are used internally by the agent.

The specification `agentHint:loadable = false` in the module parameters file can be used to restrict the loading of a module from the Sun Management Center console. The `agentHint:loadable` specification overrides the `consoleHint:loadable`.

In addition, the ability to prevent older agents from loading a module that is designed for newer agents can be specified using:

```
agentHint:loadable = true
agentHint:requiresVersion = <version>
```

The `agentHint:requiresVersion` is only used if `agentHint:loadable` is explicitly set to `true`. If the version of the agent is greater than or equal to <*version*>, then the module is loadable, otherwise, the module is not loadable. For example, if <*version*> is 2.1, and the agent version is 2.0.1, then the module will not be able to be loaded.

Alternate Way of Specifying a Module Location

This section contains information on the enterprise module parameter.

Enterprise Module Parameter

The enterprise module parameter is used to specify the OIDs file in which the location of the module is defined. Modules defined to use the `sun` or `halcyon` enterprise must define the location of where the module is to be loaded in the `base-oids-sun-d.dat` or `base-oid-halcyon-d.dat` file respectively. The contents of the OIDs file is:

```
<symbolic OID> = <numeric OID>
```

where

<*symbolic OID*> is a '/' delimited symbolic OID value for the module. This OID must also contain the appropriate `enterprise` and `moduleType` values.

<*numeric OID*> is the numeric sub id for this branch.

For example, the <*enterprise*> and <*moduleType*> specifications for the Solaris Example module are:

```
param:enterprise = halcyon
param:moduleType = operatingSystem
```

Based on the above specifications, the corresponding lines in the `base-oids-halcyon-d.dat` file could be:

```
/iso/org/dod/internet/private/enterprises/halcyon/primealert/
modules/operatingSystem/solaris = 5
/iso/org/dod/internet/private/enterprises/halcyon/primealert/
modules/operatingSystem/solaris/example = 2
```

Note – Since there exists a Solaris Standard module (more complete than the Solaris Example module), the `example` node is given the numeric sub id of 2.

Using the above specifications, the Solaris Example module is loaded under the following MIB location:

```
.iso.org.dod.internet.private.enterprises.halcyon.primealert.hal
cyon.operatingSystem.solaris.example
```

If an enterprise other than Sun and Halcyon is specified, a new file `base-oids-<enterprise>-d.dat` must be created and released. This file needs to contain the symbolic to number OID mapping of the location where the module for that enterprise is to be loaded.

Referencing Parameters

Module parameters may be referenced from the Agent file using the following syntax:

```
% <parameter name>
```

For example, to reference the `moduleName` parameter use:

```
% moduleName
```

Improving Performance using Server Override Properties File

Server Override Properties File

The Server Overrides properties file contains a list of internationalized module names which the Sun Management Center server consults before reading the module properties files for module names when filling in the Load Module pick list window. Specifying the module name in this file will remove the need in the server to read the entire properties file for the module name. Not having the module name in this file will increase the time require to fill in the Load Module window.

To specify the module name in the Server Overrides properties file use:

```
com.sun.symon.base.modules.<properties file>.moduleName=<value>
```

where:

<*properties file*> is the name of the module properties file without the suffix (.properties). If standard module naming convention is being followed this will be <*module*><*-subspec*>.

<*value*> is the internationalized module name.

Example Server Override Properties File

```
com.sun.symon.base.modules.solaris-example.moduleName=Solaris
Example
```

Additional Data Model Specifications

Specifying Hidden Managed Properties

By default all managed properties and their data are visible in the Sun Management Center console. To make a managed property and its data invisible in the Sun Management Center console, specify the following in the managed property:

```
consoleHint:hidden = true
```

Typically, this specification is used in managed properties that store data that would be meaningless to end users but are required for the module.

Data Logging Support

Automatic Data Logging

Sun Management Center agents support the logging of data to a file or to an internal cache. Logging to a file can be used for long term data storage, while data logged to the internal cache is used by the Sun Management Center console graphing.

Typically, data logging is turned on or off manually using the Sun Management Center console. However, a module can turn on data logging automatically when it is first loaded.

To do this, the following must be specified in the managed property whose data values are to be logged in the data file (note that the following files go into the agent file):

```
historyInterval = <time specification>
historyTargets(<type>) = <rowname> [ <rowname> ... ]
```

where:

<*time specification*> is the interval at which the data is logged. This value is independent of the interval at which new data is collected.

<*type*> is either `cache` or `file`. `cache` specifies that the values is logged to an internal cache, and `file` specifies that the data values is logged to a file.

<*rowname*> specifies the name of the row (for a table managed property) whose value is to be logged. Multiple rows can be logged by specifying a space separated list of row names. If the node is a scalar node, <*rowname*> must be set to `{}`.

Based on the <*type*> of data logging to be done, additional qualifiers are required.

Logging To Internal Cache

To log data to the internal cache, the following additional qualifier is required:

```
historyLength(<rowname>) = <num>
```

where:

<*rowname*> is the name of the row (for a table managed property) whose data values are to be logged. If the node is a scalar, <*rowname*> must be empty.

<*num*> is the number of data points to be stored internally. The maximum value can be set to 1500.

Logging To File

Data can be logged to two different file types:

- Typical flat file

- Circular log file

Note – Use caution when automatically turning on data logging to a flat file. Since there are no restrictions to the maximum size of the file, the file can get very large. The circular log file format maintains a fixed file size. However, the disadvantage of this is that log file eventually wraps and old data is lost.

▼ To Log Data to a Typical Flat File

- **Specify the following qualifiers:**

```
historyFileType(<rowname>) = text
historyChannel(<rowname>) = "file://localhost/../log
<filename>;flags=rw+;mode=644"
```

where:

<*rowname*> is the name of the row (for a table managed property) whose data values are to be logged. If the node is a scalar, <*rowname*> must be empty.

<*filename*> is the name of the file where the data is logged. This file is located in `/var/opt/SUNWsymon/log`.

▼ To Log Data to a Circular Log File

- **Specify the following qualifiers:**

```
historyFileType(<rowname>) = circular
historyChannel(<rownam) = history
```

These qualifers log data to the file `/var/opt/SUNWsymon/log/history.log`. No other circular log file can be specified.

Logging Data of a Scalar Node to an Internal Cache

The following example automatically logs data of a scalar node to an internal cache.

```
myscalarnode = {
    historyInterval = 120
    historyTargets(cache) = "{}"
    historyLength() = 100
}
```

Logging Two Rows of a Table Managed Property

The following example automatically logs two rows of a table managed property. One row is logged to a flat file, the other a circular log file.

```
myvectornode = {
    historyInterval = 120
    historyTargets(file) = row1 row2

    historyFileType(row1) = text
    historyChannel(row1) = "file://localhost/../log\
/myfile.txt;flags=rw+;mode=644"

    historyFileType(row2) = circular
    historyChannel(row2) = history
}
```

Specifying Module Availability

For `MANAGED-MODULES`, an `availability` managed property can be used by the module developer to flag if the entire module becomes unavailable. An example of a module becoming unavailable would be a database like Oracle that is down.

Use of the availability property is entirely optional. It does not make sense for all modules. The most common use is for database modules.

When the `availability` property indicates that the module is unavailable, all periodic data updates for the underlying objects within the module cease. The module enters a `DOWN` state, and the LEDs for the module turn black. The associated status message is <*module*> `Is Not Accessible`.

Specifying the Availability Property in the Agent File

CODE EXAMPLE 9-1 shows how to specify the availability property in the agent file:

CODE EXAMPLE 9-1 Specifying Availability Property

```
[ use MANAGED-MODULE ]
[ load <module><-subspec>-m.x ]
[ requires template <module><-subspec>-models-d ]

_procedures = { [ use PROC ]
    [ source <modules><-subspec>-d.prc ]
}
#
# Module Availability
#
availability = { [ use _procedures ]
    refreshCommand = moduleAvailability
    refreshInterval = 60
    initInterval = 0
    initHoldoff = 2
}
```

The `initInterval` and `initHoldoff` values must be set so that the availability check runs *before* any of the other refresh command within the module so that when the module is first started, the availability check is performed first and has a chance to turn off all other refresh operations if the module is not available. The values shown in the preceding example are appropriate in most cases.

Note – `initHoldoff` *must be* at least 2.

Within the <*module*><-*subspec*>`-d.prc` file is a function named `moduleAvailability` that returns `AVAILABLE` or `UNAVAILABLE`. An example of such a function is:

```
proc moduleAvailability {} {
   if { [ <some type of test> ] == true } {
      set return_code AVAILABLE
   } else {
      set return_code UNAVAILABLE
   }
   return $return_code

}
```

The data acquisition for the `availability` object can be performed in any of the ways discussed in the previous sections. However, the only limitation is that the availability object must return the string `AVAILABLE` if the module is to be available. Returning any other string puts the module in a unavailable state.

Making a Module a Core Module

Core Modules

Certain modules known as core modules must always be loaded for Sun Management Center to work correctly. As a result, these modules cannot be unloaded by users using the Sun Management Center console. To make a module a core module, include the following in the agent file:

```
consoleHint:family = core-modules
```

Persistence

```
persistentSlices = <slice1>[:<file2>] [<slice2>[:<file2>]]
persistWhenUnloaded = true
```

A `persistentSlices` qualifier can be specified in the agent file. This qualifier is a space separated list of *slices* that are to be saved to a file whenever a change in the slice value occurs. Slices represent internal data storage used by the agent, and are used to store, among other things, values that can be set by an end-user from the console (like alarm limits). The persistence functionality is required to ensure that such settings are written to a disk file, so that their values are preserved across restarts of the agent.

By default this qualifier is set to:

```
persistentSlices = value alarmlimit
```

All changes to the value and alarmlimit slices are saved to a file. The name of the file defaults to <*module*><*-subspec*>`-d.dat`, and is read by the agent on restarts.

Note – Currently, persistence is only supported for `value` and `alarmlimit` slices.

Note – By default, unloading the module deletes the persistence file. To retain the file, even if the module is unloaded, set the `persistWhenUnloaded` qualifier to `true` (default is `false`).

Specifying Adhoc Commands

Ad hoc commands provide the ability to execute certain commands in the Sun Management Center agent using the Sun Management Center console. Currently, two types of ad hoc commands are supported. They are probe commands and SNMP table management commands. SNMP table management commands are discussed in the “Using SNMP Table Management Commands” on page 190.

Command Specification

To specify an ad hoc command for a managed object, the following is required in the agent file:

```
consoleHint:commands = <command1> [<command2> ... ]
```

The `consoleHint:commands` qualifier specifies a space separated list of logical names for commands. Each logical command name must have the following additional qualifiers in the agent file:

```
consoleHint:commandLabel(<command>) = base.modules.<module><-subspec>:<key>
consoleHint:commandSpec(<command>) = <adhoc command>
```

The `consoleHint:commandLabel(<command>)` specifies the internationalization key for the text that appears in the Sun Management Center console for the command. Remember a corresponding entry in the module properties file is required. This entry has the following form:

```
<key>=<internationalized text>
```

The `consoleHint:commandSpec(<command>)` specifies the action command that runs in the agent. This command varies depending on the type of command required.

Row-Specific Commands

Adhoc commands can also be specified for vector data. As such, the commands can be differentiated based on the row of the table data on which it is executed. To do so, the `consoleHint:commands` qualifier must be replaced with `consoleHint:tableCommands`.

To specify commands that are only available at the table header use `consoleHint:tableHeaderCommands`. The usage is the same as the `consoleHint:commands` specification.

Probe Commands

Many data acquisition functions need not be performed on a periodic basis. Also, some queries (such as screen grabs) are not well suited for SNMP transport. To deal with these cases, a probe system is used that enables secure ad hoc commands and queries to be performed to remote hosts. Any module-specific functions that are not well suited to implementation within the MIB tree of the agent can be placed in the probe config file for that module.

▼ To Specify a Probe Command

The `consoleHint:commandSpec` for a probe command must be specified as:

```
consoleHint:commandSpec(<command>) = probeview %windowID snmp://
%targetHost:%targetPort/mod/<module><-subspec>/\
<path to node>?runadhoccommand.<command>
```

The `consoleHint:commandSpec(<command>)` launches a new window to display the results of the command that is executed in the agent. The *<path to node>* is a "/" delimited path to the node in the agent that specifies the command to execute. The command that is executed is specified using:

```
adhocCommand(<command>) = probeserver -c <unix command>
```

where *<unix command>* can be any UNIX command with arguments. Tcl and TOE commands can also be used to specify arguments. In this case, *<unix command>* can include `\[ <Tcl/Toe command> \]`. The Tcl or TOE command is evaluated before calling the Unix command. For example, `probeserver -c df -k \[ myFunction \]` would use the return value from `myFunction` as an argument to the UNIX command `df`.

To specify a probe command that runs `top` for the `load` object in the Solaris example module, the following is required in the agent file:

```
load = { [ use _filters ]
    type                        = active
    refreshService              = _services.sh
    refreshCommand              = uptime
    refreshFilter               = loadFilter
    refreshInterval             = 120

    consoleHint:commands        = top

    consoleHint:commandLabel(top) = base.modules.solaris-example:top
    consoleHint:commandSpec(top)  = probeview %windowID snmp://
%targetHost:%targetPort/mod/solaris-example/system\
/load?runadhoccommand.top

    adhocCommand(top)            = probeserver -c /common/local/bin/top
}
```

The corresponding entry in the properties file would be:

```
#
# Probe Queries
#
top=Top
```

Row Dependent Probe Queries

To differentiate the rows on which the query is executed, `%targetFragment` can be added to the `consoleHint:commandSpec` qualifier and `%fragment` to the `adhocCommand` qualifier. When used, both values are set to the value of that row in the column referenced by the `index` qualifier as shown in the following example.

Find Files Example

In this example, a Probe command is added to the `filesystems` node to allow users to search for files that are greater than 2 Mbytes.

CODE EXAMPLE 9-2 Find Files

```
#
# Filesystem Information
#
filesystems = { [use templates.solaris-example-models-d.filesystems _filters ]
    type                = active
    refreshService      = _services.sh
    refreshCommand      = df -kF ufs
    refreshFilter       = fileFilter
    refreshInterval     = 120

  adhocCommand(findallgt2m) = probeserver -c find %fragment -local -mount \
( -type b -o -type c ) -prune -o ( -size +4096 ) -ls

fileTable = {
        fileEntry = {
            index = mount
            consoleHint:tableCommands = findallgt2m
             consoleHint:commandLabel(findallgt2m) = base.modules.\
solaris-example:findallgt2m
           consoleHint:commandSpec(findallgt2m)  = probeview %windowID snmp:\
//%targetHost:%targetPort/mod/solaris-example/
filesystem?runadhoccommand.findallgt2m %targetFragment
        }
    }
}
```

In this example, the first argument to the find command will be the value stored in the mount column for the row on which this Probe command is executed. The corresponding entry in the Solaris Example Properties File would be:

CODE EXAMPLE 9-3 Entry in the Solaris Example Properties File

```
#
# Probe Queries
#
top=Top
findallgt2m=Find All Files > 2M
```

Note – In this example, the `consoleHint` qualifiers are specified under the `fileEntry` node. These are required at this level for all tables. However, the `adhocCommand` is specified at the `filesystems` node level. The `adhocCommand` can be specified at any level as long as the <*path to node*> specification describes the proper path to the command.

Probe Command Security

Execution of probe commands are done through the agent's shadow MIB. As such, security access of probe commands is controlled by shadow ACLs specifications. Specifically, the `runadhoccommand` shadow MIB attribute is used to run Probe commands.

By default, all probe commands are executable by all users who have read access. To limit access to specific Probe commands, the security access for the `runadhoccommand` shadow attribute needs to be overridden in the node where the probe command is defined.

▼ To Limit Top Probe Command

To limit access to the `top` probe command to users who have write access based on the standard MIB ACLs, specify the following in the `load` node:

```
load = { [ use _filters ]
    ...
    shadowMIBAccessMode(runadhoccommand) = useMIBAccess
}
```

This only allows users who have write access to the `load` node to be able to run any Probe commands for the `load` node only (this does not affect the subtree below this node).

CHAPTER 10

Modules and SNMP

This section covers the following topics:

Adding Support for SNMP Table Management

RFC 1903 defines the RowStatus textual convention to manage the creation and deletion of conceptual rows using SNMP. To provide support for management of SNMP tables in a module, the table must:

- Contain a managed property that inherits from the `ROWSTATUS` primitive
- Specify an instance node to distinguish rows in the table
- Indicate which managed properties must have their values specified by the user when creating a new row
- Indicate any data format restrictions for the user set values

ROWSTATUS Primitive

To support SNMP management of a table, a managed property that stores the state of each row is required. This managed property must inherit from the `ROWSTATUS` primitive and is typically hidden from the end-user. Nodes that inherit from `ROWSTATUS` primitive do not need to inherit from any data or alarm types. These values are set automatically by the `ROWSTATUS` primitive.

```
<node> = { [ use ROWSTATUS MANAGED-PROPERTY ]
    ...
    consoleHint:hidden = true
}
```

Instance Node

To be able to distinguish rows in the table, an instance node in the table is required. The `index` qualifier for the table refers to this node. The instance node must also specify a `instance` data format. Typically, the specific instance value for each row is specified by the user when adding a row.

```
<table entry node> = { [ use MANAGED-OBJECT-TABLE-ENTRY ]
   index = <instance node>
   ...
   <instance node> = { [ use STRING MANAGED-PROPERTY ]
      dataFormat = instance
   }
   ...
}
```

Required Values

Managed properties whose values must be specified by a user when a row is added in the table must specify the following qualifier:

```
required = true
```

Data Formats

Managed properties whose values must conform to certain data formats or need to be display in a special manner can use the following specification:

```
dataFormat = { instance | nospace | unicode | boolean |
list:<a>|<b>|... | timex}
```

where:

`instance` indicates that the value entered cannot contain any white space or special characters such as \ & * $. This is required for the instance node.

`nospace` indicates that the value entered cannot contain any spaces.

`unicode` indicates that the value entered does not need to be restricted to ASCII characters.

`boolean` indicates that parameter iw displayed as a yes/no check box when adding a row. The value that is set to the agent is a 1 or 0, respectively.

`list:<a>|<b>...` specifies "|" separated list that is displayed as a picklist when adding a row. To internationalize the values, the list elements can be replaced with *<a>*,`base.modules.`*<module><-subspec>*`:`*<keya>*. *<keya>* is the key in the Properties file corresponding to the internationalized text used for display purposes. *<a>* is the value set in the agent if that option is selected.

`timex` indicates that the value entered must conform to the required time specification. Using this value also places an *Advanced...* button in the row adder window to facilitate the entry of the time specification.

Example—`Filesize`

Shown below are fragments of the model file for the `Filesize` table module from the Appendix C.

CODE EXAMPLE 10-1 Model file For the Filesize Module

```
type = reference
file = { [ use MANAGED-OBJECT ]
    ...
    fileTable = { [ use MANAGED-OBJECT-TABLE ]
        ...
        fileEntry = { [ use MANAGED-OBJECT-TABLE-ENTRY ]
            ...
            index = instance
```

CODE EXAMPLE 10-1 Model file For the Filesize Module *(Continued)*

```
            rowstatus = { [ use ROWSTATUS MANAGED-PROPERTY ]
                ...
                consoleHint:hidden     = true
            }

            instance = { [ use STRING MANAGED-PROPERTY ]
                ...
                dataFormat             = instance
            }

            name = { [ use STRING MANAGED-PROPERTY ]
                ...
                required               = true
            }

            size = { [ use INTHI MANAGED-PROPERTY ]
                ...
            }
        }
    }
}
```

Adding Support for Global Table or Row Actions

Support is provided for actions operating on the entire table or an individual row in the table. These global actions are initiated after the completion of an SNMP set of one or more rows in the table. Global set actions are executed only once for a entire row or table, as opposed to individual actions performed for every column in which a SNMP set is done, for example, a table with N columns whose values must be written to a file.

If a new row is created, the new data could be written to the file in one of two ways:

- A set action can be specified for each column that writes the data to a file after the set. Thus N writes are required for the N columns.
- A global action to write the entire row to the file once all the values have been set. Only one write would be required for the entire table or each row that is set.

To provide support for global actions, each node whose value must be set before the global action is executed must inherit from one of two primitives:

- GLOBTABLENODE to support global table actions
- GLOBROWNODE to supports global row actions

```
<node> = { [ use <globprimitive> <data/alarm type> MANAGED-PROPERTY ] {
    ...
```

where:

<*node*> is the name of the node whose value must be set before the global action is executed.

<*globprimitive*> is either GLOBTABLENODE or GLOBROWNODE.

<*data/alarm type*> is the data and/or alarm type primitive.

If multiple nodes in a table inherit from the global primitives, they must all use the same primitive. In this case, the global actions are only executed after an SNMP set to all nodes inheriting from the global primitive is complete.

The difference between the two primitives is that the GLOBROWNODE executes the global actions for each row that has been set. The GLOBTABLENODE executes the global actions once only, regardless of the number of rows that were set. For example, if a table containing nodes that inherit the GLOBTABLENODE primitive, has 2 rows added using a single SNMP set, the global actions are only executed once. However, if the table nodes inherited from the GLOBTABLEROW primitive, the global actions are executed twice. Once for each row that was set.

Adding Node Icons

A managed object (cannot be used for managed properties) can be assigned an icon to display in the Sun Management Center console. To do so, the following must be specified in the node:

```
consoleHint:smallIcon(DFT) = stdimages/<name>16x16-j.gif
consoleHint:largeIcon(DFT) = topoimages/<name>32x32-j.gif
```

By convention, the larger icon (32x32) is used only in the topology view in the Sun Management Center console. In all other instances, the smaller icon (16x16) is used in the Sun Management Center console.

Adding SNMP Table Management

When adding support for SNMP table management, the table must specify which of its managed properties must have its values set in order to create a new row. The value of the remaining managed properties of the table is set to a predefined default value when a new row is created.

To indicate which managed properties must have their values set to create a new row, the managed property must be made externally SNMP-writable. To specify that a managed property is writable, use the following construct in the data model realization file:

```
access = rw
```

By default, all nodes are set to read only (`ro`) so access only needs to be specified for those managed properties that must have their values set to create a row. If the value of any writable managed property is not specified in an SNMP set, the set fails and the row is not created.

Note – Inherits from the `ROWSTATUS` primitive automatically have `access` set to `rw`.

When using the Sun Management Centerconsole to add a new row, the values for all writable managed properties can be specified by the user. If you wish to have a managed property that is SNMP writable, but not have its value specified using the Sun Management Center console, use:

```
access = rw
consoleHint:editAccess = ro
```

By default, all nodes have `consoleHint:editAccess` set to `rw`. As indicated above, all non-writable managed properties of table *must* specify a default value. The managed property is initialized to this value when a new row is created. If non-writable node does not specify a default value, the set to create a new row fails. To specify a default value, use:

```
defaultvalue = <value>
```

The default value can also be specified for SNMP writable nodes. In this case, the default value is presented to the user in the Sun Management Center console when creating a new row.

User-defined Actions

The Sun Management Center agent supports the ability to execute user-defined actions that are triggered when a managed property is created or when values are set into the managed property. The specification for user actions is:

```
<type>Actions[(<qualifier>)] = <action> [<action> <action> ... ]
```

where:

<*type*> is the type of event used to trigger the execution of the actions. The available <*type*> of actions are described in the following sections.

<*qualifier*> is optional and dependent on the <*type*> of actions to execute.

<*action*> is a space separated list of logical names of the actions to be executed in the order that they are listed.

Each action name must be associated with a corresponding service and command that is used to execute the action.

```
<type>Service(<action>) = <service>

<type>Command(<action>) = <command>
```

where:

<*service*> is the service used to run the command. Conceptually, the <*command*> is sent to the service to be run.

<*command*> is the command that is run.

The specific actions of <*type*>s are described in the following sections.

Activate Actions

User-defined actions can be executed before or immediately after a MIB node has been created.

- **To specify actions to be executed before the node is created, use:**

```
activateActions(pre) = <action> [<action> ...]
```

- **To specify actions to be executed after the node is created, use:**

```
activateActions(post) = <action> [<action> ...]
```

A service and command must be defined for each *<action>* specified:

```
activateService(<action>) = <service>
activateCommand(<action>) = <command>
```

For example, to set a default value (0) for a scalar node use:

```
activateActions(post) = setdefaultvalue
activateService(setdefaultvalue) = _internal
activateCommand(setdefaultvalue) = setValue 0 0
```

SNMP Set Actions

The Sun Management Center agent MIB supports the specification of actions to be executed when the value of MIB objects are set. Managed property values can be set by the data cascade from a refresh command or from external SNMP sets. For a node to allow sets by external SNMP sets, the node must be made to be SNMP-writable. This is done using:

```
access = rw
```

At several points during the set value process, user-defined actions can be triggered. The set value process consists of:

1. **Prevalidate the set.**
2. **Set the value (no actions can be defined at this stage).**
3. **Post-set check for `ROWSTATUS` nodes only.**
4. **Post-validate the set.**

5. **Row set actions for `ROWSTATUS` nodes only.**

6. **Set actions.**

   If any of these actions (after the prevalidate step) fails, the value automatically rolls back to its preset value. User-defined rollback actions can also be specified. Each of these actions is described in the next section.

Prevalidate Actions

Prevalidate actions can be specified to execute before a value is set into a managed property. The purpose of prevalidate actions is to ensure that the value can be set into the node. Nodes that inherit from the `ROWSTATUS` and `TESTANDINCR` primitives have predefined prevalidate actions. User-defined pre-validate actions can be defined using:

```
validateActions(pre) = <action> [<action> ... ]
```

The service and commands for each <*action*> must be defined using:

```
validateService(<action>) = <service>
validateCommand(<action>) = <command>
```

The parameter `%value` is available to the <*command*> for reference. The <*command*> must return a 0 if the validation was not successful. Returning a zero value generates an `inconsitentValue` SNMP error and the value is not set.

postrow Actions

Nodes that inherit from the `ROWSTATUS` primitive (see *"Adding Support for SNMP Table Management"* in the previous chapter), have predefined postrow actions. Users can also specify postrow actions. These actions are triggered to execute after the set but before the postvalidate actions. These actions can be specified using:

```
postrowActions(<rowstatus state>) = <action> [ <action> ... ]
```

where:

<*rowstatus state*> is the state of the ROWSTATUS node corresponding to the value that is set into it. TABLE 10-1 lists the allowable states.

TABLE 10-1 Allowable rowstatus States

SNMP Set Value	State
0	doesNotExist
1	active
2	notInService
3	notReady
4	createAndGo
5	createAndWait
6	destroy

For each <*action*> specified, a service and command must be defined.:

```
postrowService(<action>) = <service>
postrowCommand(<action>) = <command>
```

The value and row index parameters are available to the <*command*> using %value and %index respectively. <*command*> must return a 0 if the action was not successful. Returninga zero value generates an inconsitentValue SNMP error and the object returns to its pre-set value. A zero return code also triggers any user-defined rollback actions.

Postvalidate Actions

Post-validate actions can be specified to validate the set value. Nodes that inherit from the ROWSTATUS, GLOBROWNODE, and GLOBTABLENODE primitives have predefined postvalidate actions. User-defined postvalidate actions can be defined using:

```
validateActions(post) = <action> [<action> ... ]
```

The service and commands for each <*action*> must be defined using:

```
validateService(<action>) = <service>
validateCommand(<action>) = <command>
```

The parameter %value is available to the <*command*> for reference. The <*command*> must return a 0 if the validation was not successful. Returning a zero value generates an inconsitentValue SNMP error and returns the object to its preset value. A zero return code also triggers any user-defined rollback actions.

If validateActions(post) actions are specified for a node that inherits from either the GLOBROWNODE or GLOBTABLENODE primitives the actions list must include incrglob. For example:

```
validateActions(post) = myaction incrglob
```

The incrglob service and command are predefined and need not be stated again.

setrow Actions

Nodes that inherit from the ROWSTATUS primitive have predefined setrow actions. Users can also specify setrow actions. These actions are triggered to execute after the post-validate check but before the set actions. These actions can be specified using:

```
setrowActions(<rowstatus state>) = <action> [ <action> ... ]
```

where:

<*rowstatus state*> is the state of the ROWSTATUS node corresponding to the value that is set into it. The allowable states are described in TABLE 10-1.

For each <*action*> specified a service and command must be defined:

```
setrowService(<action>) = <service>
setrowCommand(<action>) = <command>
```

The value and row index parameters are available to <*command*> using %value and %index respectively. <*command*> must return a 0 if the action was not successful. Returning a zero value generates an inconsitentValue SNMP error and returns the object to its preset value. A zero return code also triggers any user-defined rollback actions.

Set Actions

Set actions are triggered after all validation checks have passed and the value has been set. All nodes that have an <*alarm type*> specified as well as nodes that inherit from the `ROWSTATUS, GLOBROWNODE and GLOBTABLENODE` primitives have predefined set actions. User-defined set actions can be specified using:

```
setActions = <action> [<action> ... ]
```

If no `setActions` are specified, the value of the object is set to the value specified in the set.

An asterisk (*) can be prepended to a single action to indicate that the value returned by the corresponding `setCommand` must be the value set into the object at the end of the execution of the `setActions`. If more than one action has a (*) prepended to it, the return value from the last action with the (*) is used. If no action has an (*) prepended, the value remains as it was set.

Each set action must specify a service and a command:

```
setService(<action>) = <service>
setCommand(<action>) = <command>
```

The value that was set as well as the row name and corresponding index for tables can be referenced by the <*command*> using `%value, %rowname`, and `%index` respectively.

If the `setService` is a shell service and if the `setCommand` script returns any data on `stderr`, the set action fails. Any data returned, from `stdout` for a script or Tcl return value can be used as the new value for the object.

If `setActions` actions are specified for a node that inherits from either the `GLOBROWNODE or GLOBTABLENODE` primitives the actions list must include `decrglob`, for example:

```
setActions =+ myaction decrglob
```

The `decrglob` service and command are predefined and need not be stated again.

CODE EXAMPLE 10-2 Set Actions

```
setObjectNode = {
    access                   = wo
    setActions               = *run_script run_tcl_proc decrglob
    setService(run_script)   = _services.sh
    setCommand(run_script)   = script.sh %value
    setService(run_tcl_proc) = otherObject
    setCommand(run_tcl_proc) = tcl_proc %value
}
```

Rollback Actions

Rollback actions are triggered if the `set, postrow, postvalidate` or `setrow` actions fail. Nodes that inherit from `GLOBROWNODE` and `GLOBROWTABLE` primitives have predefined rollback actions. The purpose of rollback actions is to restore the state of the object after the failed set. Rollback actions can be specified using:

```
rollbackActions = <action> [<action> ... ]
```

Each *<action>* specified must have a service and command defined.

```
rollbackService(<action>) = <service>
rollbackCommand(<action>) = <command>
```

If `rollbackActions` actions are specified for a node that inherits from either the `GLOBROWNODE or GLOBTABLENODE` primitives the actions list must include `clearglob`, for example:

```
rollbackActions = myaction clearglob
```

The `clearglob` service and command are predefined and need not be stated again.

Global Actions

Tables that support global table or row actions can specify global actions using:

```
globActions = <action> [ <action> .. ]
```

Each <*action*> specified must also define a service and command to execute.

```
globService(<action>) = <service>
globCommand(<action>) = <command>
```

The value that was set plus the row name and corresponding index for tables can be referenced by the <*command*> using `%value`, `%rowname`, and `%index` respectively.

These global action qualifiers *must* be specified in the table node that inherits from the `MANAGED-OBECT-TABLE-ENTRY` primitive.

Adding SNMP Security

The Sun Management Center agent MIB supports the specification of multiple levels of SNMP read or write access controls. These access control (ACL) specifications define the minimum security level required of users and/or groups to perform SNMP read or write operations on objects in the MIB.

ACLs for the module can be specified in any node. If the ACL is specified in a branch, the ACL applies to the entire subtree (unless it is overridden by another ACL specification in an inferior node). ACLs specified in a leaf apply to the leaf node only.

ACLs are specified using the following format:

```
userAccess(<userName>,<accessType>) = <securityLevel>
groupAccess(<groupName>,<accessType>) = <securityLevel>
```

where:

<*userName*> can be a UNIX user name or a logical user or community name. Logical user and community names are defined below.

<*groupName*> can be a UNIX group or a logical group name. Logical group names are defined below.

<accessType> can be read or write and is the SNMP operation that is to be controlled.

<securityLevel> can be none, priv, auth or noauth and defines the minimum level of security required for this type of access.

In general, if the default ACLs are insufficient, module designers must specify their own ACLs at the root node of the module. Where applicable, additional ACLs can be specified for specific subtrees and nodes within the module.

Note – As a design rule, UNIX users and groups must not be hard-coded for *<userName>* and *<groupName>* respectively. Doing so assumes that such users and groups always exist. Instead, the use of logical users and groups is encouraged.

Logical Users, Groups, and Community Names

Logical users, groups, and community names are used to enable module designers to grant access to the MIB based on a space separated list of UNIX users, UNIX groups, and community names that is modifiable at run-time. The list of UNIX users and groups must be defined in the UNIX domain of the Sun Management Center server layer and are logically OR'ed together to determine membership.

Three levels of logical users, groups, and community names are defined:

`admin`—users, groups, and communities belonging to this category are able to perform all operations

`operator`—users, groups, and communities in this category are allowed to perform selected operations.

`general`—users, groups, and communities in this category have read access only

To specify logical users in the ACL specifications use `%adminUsers`, `%operatorUsers`, or `%generalUsers` for the three different levels of logical users.

To specify logical groups in the ACL specifications use `%adminGroups`, `%operatorGroups`, or `%generalGroups` for the three different levels of logical users.

To specify logical communities in the ACL specification use `%adminCommunities`, `%operatorCommunities`, and `%generalCommunities`.

The default memberships to the logical users, groups, and communities are defined in the file `agent-acls-d.dat` and are specified as:

CODE EXAMPLE 10-3 Default Memberships to Logical Users, Groups and Communities

```
%adminUsers =
%operatorUsers =
%generalUsers =

%adminGroups = esadm
%operatorGroups = esops
%generalGroups = ANYGROUP

%adminCommunities =
%operatorCommunities =
%generalCommunities = public
```

These default values can be overridden by copying this file to `/var/opt/SUNWsymon/cfg` and modifying it. The `esadm` and `esops` UNIX groups are created during installation. The keyword `ANYGROUP` is not a true UNIX group, but rather is a special keyword that means that any user that was allowed to log into Sun Management Center.

Note – The UNIX groups `esadm` and `esops` are only required to be defined in the UNIX domain where the Sun Management Center server layer is running. That is, the Sun Management Center server layer resolves all logical user and group lists.

Security Levels

The possible security levels that can be specified for <*securityLevel*> in the ACL specification are:

- `noauth`—non-authenticated, authenticated, and encrypted requests are permitted.
- `auth`—authenticated and encrypted requests are permitted.
- `priv`—only encrypted requests are permitted.
- `none`—no access, regardless of security level of the request.

Default ACLs

CODE EXAMPLE 10-4 Default ACL settings for All Nodes

```
userAccess(%adminUsers,read)            = auth
userAccess(%operatorUsers,read)         = auth
userAccess(%generalUsers,read)          = auth

userAccess(%adminUsers,write)           = auth
userAccess(%operatorUsers,write)        = auth
userAccess(%generalUsers,write)         = none

groupAccess(%adminGroups,read)          = auth
groupAccess(%operatorGroups,read)       = auth
groupAccess(%generalGroups,read)        = auth

groupAccess(%adminGroups,write)         = auth
groupAccess(%operatorGroups,write)      = auth
groupAccess(%generalGroups,write)       = none

userAccess(%adminCommunities,read)      = noauth
userAccess(%operatorCommunities,read)   = noauth
userAccess(%generalCommunities,read)    = noauth

userAccess(%adminCommunities,write)     = noauth
userAccess(%operatorCommunities,write)  = noauth
userAccess(%generalCommunities,write)   = none
```

These specifications define the following behavior:

- admin / operator / general users and groups have read access using SNMP requests with at least an authenticated security level.
- admin / operator / general communities have read access using SNMP requests with an unauthenticated security level.
- admin / operator users and groups have write access using authenticated SNMP requests.
- admin / operator communities have write access using unauthenticated SNMP requests.
- general user, groups, and communities have no write access.

Examples–Specifying ACLs

The following examples demonstrate how ACLs can be specified and what impact they will have.

- To permit the UNIX user `fly` to perform SNMP `get` and `set` operations with authenticated and encrypted requests:

CODE EXAMPLE 10-5 Specifying Authenticated/Encrypted SNMP `get` and `set` Requests

```
userAccess(fly, read) = auth
userAccess(fly, write = auth
```

- To permit the UNIX user `fly` to perform SNMP `get` operations with authenticated and encrypted requests but do not allow SNMP `set` operations for the UNIX user `fly`:

CODE EXAMPLE 10-6 Specifying Requests without SNMP `set` operations for UNIX User

```
userAccess(fly, read)  = auth     read) = auth
userAccess(fly, write) = auth     write) = none
```

- To permit the admin and operator logical users and groups, defined by Sun Management Center site administrators, to perform SNMP `get` and `set` operations with authenticated and encrypted requests:

CODE EXAMPLE 10-7 Permitting admin/operator to Perform SNMP `get` and `set`

```
userAccess(%adminUsersread)          = auth
userAccess(%adminUserswrite)         = auth
userAccess(%operatorUsersread)       = auth
userAccess(%operatorUserswrite)      = auth

groupAccess(%adminGroupssread)       = auth
groupAccess(%adminGroupsswrite)      = auth
groupAccess(%operatorGroupsread)     = auth
groupAccess(%operatorGroupswrite)    = auth
```

Using SNMP Table Management Commands

For modules that support SNMP management of tables, ad hoc commands can be added to manage table rows from the Sun Management Center console. These commands can be used to add, remove, edit, disable and enable rows.

▼ To Add a Row

The commandSpec to add a row to a table is:

```
consoleHint:commandSpec(<command>) = launchUniqDialog
%windowID .templates.tools.rowadder objectUrl=snmp://
%targetHost:%targetPort/mod/<path to table entry>#%targetFragment
```

where:

<path to table entry> is a slash (/) delimited path to the table node that inherits from the MANAGED-OBJECT-TABLE-ENTRY primitive.

▼ To Remove a Row

The commandSpec to remove a row from a table is:

```
consoleHint:commandSpec(<command>) = requestTableRowOperation
%windowID snmp://
%targetHost:%targetPort/mod/<path to rowstatus \
node>#%targetFragment unload
```

where *<path to rowstatus node>* is a slash (/) delimited path to the table node that inherits from the ROWSTATUS primitive. This action sets the state of the rowstatus node for this row to destroy.

▼ To Edit a Row

The commandSpec to edit an existing row in a table is:

```
consoleHint:commandSpec(<command>) = launchUniqDialog
%windowID .templates.tools.roweditor objectUrl=snmp://
%targetHost:
%targetPort/mod/<path to table entry>#%targetFragment
```

where *<path to table entry>* is a "/" delimited path to the table node that inherits from the MANAGED-OBJECT-TABLE-ENTRY primitive.

▼ To Disable a Row

The `commandSpec` to disable a row in a table is:

```
consoleHint:commandSpec(<command>) = requestTableRowOperation
%windowID snmp://%targetHost:%targetPort/mod/<path to rowstatus \
node>#
%targetFragment disable
```

where:

<path to rowstatus node> is a slash (/) delimited path to the table node that inherits from the `ROWSTATUS` primitive. This action sets the state of the rowstatus node for this row to `notInService`.

▼ To Enable a Row

The `commandSpec` to enable a row in a table is:

```
consoleHint:commandSpec(<command>) = requestTableRowOperation
%windowID snmp://
%targetHost:%targetPort/mod/<path to rowstatus node>#
%targetFragment enable
```

where:

<path to rowstatus node> is a slash /) delimited path to the table node that inherits from the `ROWSTATUS` primitive. This action sets the state of the rowstatus node for this row to `active`.

▼ To Load a Module Instance

For modules that can be loaded multiple times, the `consoleHint:commandSpec` qualifier requires the `%targetInstance` specification.

For example, if the Solaris example module can be loaded multiple times, then the `consoleHint:commandSpec` for the top probe query is:

```
consoleHint:commandSpec(top) = probeview
%windowID snmp://
%targetHost:
%targetPort/mod/solaris-example+
%targetInstance/system load?runadhoccommand.top
```

Example: Adhoc SNMP Table Management

CODE EXAMPLE 10-8 Adhoc SNMP Table Management Commands

```
# Examle for using adhoc snmp table management commands
# Additional data model realization specifics pertaining to the
# mySystems object in the solaris-example-d.x file.
mySystems{ [ use templates......]

        myTable = {
            myEntry = {

               consoleHint:tableCommands = enable disable unload addrow editrow
                 consoleHint:commandLabel(enable) = enable
                 consoleHint:commandSpec(enable) = requestTableRowOperation
%windowID snmp://%targetHost:%targetPort/mod/fscan+%targetInstance/fscanstats\
/myTable/myEntry/rowstatus#%targetFragment enable
                 consoleHint:commandLabel(disable) = disable
                 consoleHint:commandSpec(disable) = requestTableRowOperation
%windowID snmp://%targetHost:%targetPort/mod/fscan+%targetInstance/fscanstats\
/myTable/myEntry/rowstatus#%targetFragment disable
                 consoleHint:commandLabel(unload) = unload
                 consoleHint:commandSpec(unload) = requestTableRowOperation
%windowID snmp://%targetHost:%targetPort/mod/fscan+%targetInstance/fscanstats\
/myTable/myEntry/rowstatus#%targetFragment unload

                  consoleHint:commandLabel(addrow) = addrow
                    consoleHint:commandSpec(addrow) = launchUniqueDialog
%windowID .templates.tools.rowadder objectUrl=snmp://%targetHost:%targetPort\
/mod/fscan+%targetInstance/fscanstats/myTable/myEntry#%targetFragment
                    consoleHint:commandSpec(addrow) = \  launchUniqueDialog
%windowID .templates.tools.rowadder objectUrl=snmp://%targetHost:%targetPort\
/mod/fscan+%targetInstance/fscanstats/myTable/myEntry#%targetFragment
                   consoleHint:commandLabel(editrow) = editrow
                consoleHint:commandSpec(editrow) = launchUniqueDialog %windowID
.templates.tools.roweditor objectUrl=snmp://%targetHost:%targetPort/mod/\
fscan+%targetInstance/fscanstats/myTable/myEntry#%targetFragment
                 consoleHint:tableHeaderCommands = addrow
                 rowstatus = { [ use _procedures ]
                     setrowService()  = fscanstats

                     setrowActions(active)  = on
                     setrowCommand(on)  = activatePattern %index %rowname

                     setrowActions(notInService)  = off
```

CODE EXAMPLE 10-8 Adhoc SNMP Table Management Commands

```
                    setrowCommand(off)  = deactivatePattern %index %rowname

                    setrowActions(createAndGo)  = add
                    setrowActions(createAndWait)  = add
                    setrowCommand(add)  = addPattern %index %rowname %newvalue

                    setrowActions(destroy)  = remove
                    setrowCommand(remove)  = removePattern %index %rowname
                }
            }
        }
}
```

Example: Additional Objects to the Solaris Example File

CODE EXAMPLE 10-9 Additional Objects to the Solaris Example Model `d.x` File

```
#Additional object to the solaris-example-model-d.x file
mySystems = { [ use MANAGED-OBJECT ]
    mediumDesc            = example
        myTable = { [ use MANAGED-OBJECT-TABLE ]
            mediumDesc = myTable
            consoleHint:mediumDesc = myTable
            myEntry = { [ use MANAGED-OBJECT-TABLE-ENTRY ]
                mediumDesc         = my Entry
                index              = idnum
                consoleHint:mediumDesc =  my Entry
      rowstatus = { [ use GLOBROWNODE ROWSTATUS MANAGED-PROPERTY ]
                    shortDesc   = row status
                    mediumDesc  = Row Status
                    fullDesc    = The row status
                    consoleHint:hidden = true
                    consoleHint:mediumDesc = Row Status
                }
                    idnum = { [ use INT MANAGED-PROPERTY ]
                        shortDesc       = number
                        mediumDesc      = ID Number
                        fullDesc        = ID number

                    consoleHint:mediumDesc = ID Number
                    }
                    name = { [ use STRING MANAGED-PROPERTY ]
                        shortDesc       = name
                        mediumDesc      = Users Name
                        fullDesc        = Users Name
                    required = true

                    consoleHint:mediumDesc = Users Name
                    }
                    hobby = { [ use STRING MANAGED-PROPERTY ]
                        shortDesc       = hobby
                        mediumDesc      = Users Hobby
                        fullDesc        = Users hobby

                    consoleHint:mediumDesc = Users Hobby
                    }
```

CODE EXAMPLE 10-9 Additional Objects to the Solaris Example Model `d.x` File

```
                }
        }
}
```

Sending Traps from the Agent

When the value of one or more managed properties changes in the module, the agent can notify the console immediately of the changes by sending a refresh trap. The refresh trap command should be executed in the context of the managed property (that is, the MIB node) that is being refreshed. These commands can be included either in the `.prc` file or the `.flt` files.

The following are the different ways of sending refresh traps:

- **refreshValueAndTrap**

  When this command is executed in the context of a MIB node, it refreshes the value of the node by running the appropriate refreshCommand, sends a trap to the server, thus notifying the console of the changes in that node, and the console updates the GUI with the new values.

- **setTrapInfo refreshOID**

  When this command is executed in the context of a MIB node, it sends a trap to the server indicating that the values of that node has been refreshed. The console upon receiving the refreshed values updates the GUI.

Example: Agent File

CODE EXAMPLE 10-10 Example of the Agent File

```
Example: Agent File
..
..
..
createNode = { [ use _procedures ]
         type            = passive
         access = rw
         setActions      =  cmd
         setService(cmd) = _internal
         setCommand(cmd) = runShellCmd %value %rowname
}
AnotherNode = { ..
    ..
        type            = active
        initInterval = 2
        refreshService  = _services.sh
        refreshCommand  = get-data.sh
        refreshInterval = 300
}
..
..

# The corresponding procedure file
..
..
..
proc runShellCmd{ val rowname} {
    # Do some processing
    ..
    ..
    set file [ open /tmp/$rowname w ]
        puts $file “Refresh in progress”
        close $file
    toe_send [ locate anotherNode ] setValue 0 $val
    toe_send [ locate anotherNode ] setTrapInfo refreshOID
    set file [ open /tmp/$rowname w ]
    puts $file “Refresh Successful”
    close $file
```

CODE EXAMPLE 10-10 Example of the Agent File

```
    ..
    ..
}
```

Using the `mib2x` Tool

The `mib2x` tool allows you to:

- Generate a <*module*>`-models-d.x` file from an SNMP MIB.
- Generate a <*module*>`-oids.dat` file from an SNMP MIB.

The <*module*>`-models-d.x` file defines the data model of the corresponding module. It also define the OIDs corresponding to the data modeled.

Using this tool, you can generate the data model file of an SNMP MIB with ease, and you can plug in additional qualifiers for the data as required by your module. This speeds up the data model creation part of the module development, when you are developing the module from an already existing SNMP MIB.

`mib2x` Syntax

The `mib2x` syntax is:

```
/opt/SUNWsymon/sbin/es-run mib2x  -f <filename> [-r [<prefix>:]<root>] \
[-m <mode>]
            [-b <base mibs directory>] [-i <index directory>]
            [-a <additional mib files to import>]
```

The following table describes the syntax and its options.

TABLE 10-2 `mib2x` Syntax and Options

Syntax	Description	Req'd	Default
`-f <filename>`	The absolute path to the ASN.1 MIB file read by `mib2x` (specify the file's directory name along with the file name)	Yes	None

`-r [<prefix>:]<root>`	Specifies the starting node of the output of `mib2x`. `mib2x` only prints out information on the mib tree under this node. If the optional <prefix> is specified, node names are prefixed with the path from <prefix> to <root>. For example, if iso:sun is specified, then the output is:
	`iso = 1`
	`iso/org = 1.3`
	`iso/org/dod = 1.3.6`
	`iso/org/dod/internet/private = 1.3.6.1.4`
	`iso/org/dod/internet/private/enterprises = 1.3.6.1.4.1`
	`iso/org/dod/internet/private/enterprises/sun = 1.3.6.1.4.1.42`
	`<the entire tree under sun>`
		No	`iso`

`-m <mode>`	The output format of `mib2x`. `<mode>` can be: `syntax|oid|module`: • `syntax` prints out variable name to syntax mapping • `oid` prints out variable name to OID mapping • `module` converts ASN.1 MIB text files to agent module definition file. Must specify the `-r` option	No	oid

TABLE 10-2 mib2x Syntax and Options *(Continued)*

Syntax	Description	Req'd	Default
-b \ <base mibs directory>	The directory that contains the basic MIB file required by mib2x.	No	<Install_Dir_of_Sun Management Center>/util/cfg
-i <index directory>	The directory that contains the machine dependent directory with frozen images of the MIB files.	No	/tmp
-a \ <list of mib files>	The list of mibs files is used to resolve import clauses in <filename>. For example: -a "file1 file2 file3"	No	-

The output of mib2x is always written to the standard output. You can redirect this output to an appropriate file.

Examples of mib2x

- **To generate the models file for the MIB file** mymib.txt **and to store the output in** mymib-models-d.x**, enter the following command:**

```
/opt/SUNWsymon/sbin/es-run mib2x -f /<directory>/mymib.txt -m \
module -r <root_of_module> >mymib-models-d.x
```

- **To generate the OIDs for the MIB file** mymib.txt **and to store the output in** mymib-oids.dat**, enter the following command:**

```
/opt/SUNWsymon/sbin/es-run mib2x -f /<directory>/mymib.txt > \
mymib-oids.dat
```

Note – When using the mib2x utility, you may sometimes see messages, such as:

```
mib2x: unknown macro NOTIFICATION-TYPE Using branch.
mib2x: unknown macro MODULE-COMPLIANCE. Using branch.
mib2x: unknown macro OBJECT-GROUP. Using branch.
```

These messages are harmless. The mib2x utility models the nodes using the above macros as branches in the module.

CHAPTER 11

Agent Interactive Mode

This section covers the following topics:

Sun Management Center software users can run the agent in the interactive mode. In this environment, they can communicate with the agent using Tcl/TOE commands. Any commands available in the agent can be executed in interactive mode dependent on the context in which they are executed.

Running the agent in the interactive mode allows users to debug their agents at runtime, which allows the users to perform the following tasks:

- Navigate the object tree interactively.
- Create or destroy objects.
- Define, lookup, or display certain object attributes.
- Invoke an operation in a particular object context.
- Set a group of object attributes from a file or export the object information to a file.
- Generate SNMP MIB from a module.

Working in the Agent Interactive Mode

This chapter includes examples with relation to working within the agent interactive mode.

▼ To Work Within the Agent Interactive Mode

- **Start the agent in the interactive mode:**

```
shell% sbin/es-start -ia
```

▼ To Exit the Environment

```
agent> exit
```

Tcl/TOE Commands

This section describes a set of commands available to all nodes. These commands allow users to create or destroy objects in the agents, apply object interaction, apply dictionary operation, navigate object tree, import data from a file to the agent, export agent to an file and load classes or packages at runtime.

Object Creation

The following commands can be used to create a new TOE object and destroy an existing TOE object.

```
toe_create [PARENT ...][-superior SUPERIOR]
```

Creates a new TOE object. An optional list of the *parent* object can be specified to define the objects inheritance tree. The optional *superior* object can be used to specify the object's superior in the sense of an object tree. This permits trees of objects to be created where the objects' ancestry is independent from the organizational tree structure.

```
toe_destroy OBJECT ...
```

Destroys the specified OBJECT(s).

Object Relationship

These commands allow users to trace or establish the relationship among objects.

```
toe_parents [PARENT PARENT ...]
```

With no arguments, this command lists the parents of the current TOE object. If any parent objects are specified, then the parent list of the object is changed to the new object list. This is useful for adding or removing class relationships dynamically, but should be used with extreme caution.

```
toe_superior [SUPERIOR]
```

With no argument, this command returns the superior object of the current TOE object. If `SUPERIOR` is specified as the argument, the superior object of the current object is set to `SUPERIOR`. As with `toe_parents`, users must use caution when changing the superior state of an object after it has been fully constructed.

```
toe_child NAME OBJECT
```

Establishes OBJECT as inferior of the current TOE object. This essentially does three things:

- Sets OBJECT's name to NAME.
- Sets OBJECT to be a subobject of the current object.
- Sets the superior of OBJECT to be the current object.

This is the primary means of establishing tree relationships.

```
locate NAME [-noscope] [-create PARENTS]
```

Locates an object in the object tree by name and returns its TOE identification. If `-noscope` is specified, the search does not traverse up the object tree. If `PARENTS` are specified using the `-create` option, the object are created if it does not already exist, and its parents are set accordingly.

```
inherit OBJECT ...
```

Adds the specified OBJECT(s) to the parent list of the current TOE object. This is similar to `toe_parents`, except the inherit command is implicitly safer as it cannot remove parents.

Object Interaction

The following commands allow operation apply to a certain object context instead of the current one:

```
toe_begin OBJECT
```

Sets the current TOE object context to OBJECT. The interpreter maintains a stack of object contexts, and this pushes the specified object onto the context stack. The current object is the base for all command and data references.

```
toe_end
```

Pops the current object off the context stack and reestablishes the previous context.

```
toe_self
```

Returns the TOE identification of the current TOE object context.

```
toe_send OBJECT COMMAND
```

Evaluates `COMMAND` in the context of `OBJECT`. This is equivalent to beginning the object (using `toe_begin`), evaluating the command (using `eval`), and ending the object (using `toe_end`).

```
toe_csend VAR OBJECT COMMAND
```

Performs a *catch send* function, which is identical to a combination of `toe_send` and the Tcl catch command. Any results or error messages from COMMAND are placed in the variable named VAR, and the entire command returns 0 for success, 1 for error.

```
toe_supersend OBJECT COMMAND
```

The supersend operation invokes the method of another object (usually a class) in the context of the current object. This is usually used within object constructors, to perform an initialization operation defined in a particular class on the current instance.

```
toe_recurse pre|post COMMAND
```

Traverses the object tree below the current object and runs COMMAND on each object. Whether the COMMAND is run before or after the subtrees are traversed depends on whether “pre” or “post” is specified.

```
toe_name [-full][OBJECT]
```

Returns the tree-based name of the specified TOE object. If no OBJECT is specified then the name of the current object is returned. If the `-full` option is used then the object’s full name, in absolute terms from the root of the object tree, is returned.

```
toe_dump [OBJECT]
```

Outputs the parents, superior, methods and data keys of the specified object. If no object is specified, the current object is dumped.

Dictionary Operations

These commands provide capability to allow users to apply operations on a certain entry, in the current object, such as define / undefined an entry, find the value of an entry or copy the value of a certain entry to a new entry.

```
define SLICE KEY VALUE [-t TYPE]
```

Creates a new dictionary entry in the current TOE object. A dictionary entry is identified by two names, SLICE and KEY. The slice is the category or general type of information, and the key is the specific name. This allows an object's data space to be partitioned into logical groups (for example, runtime data versus static configuration information).

The optional TYPE can be used to associate a data type to the key. The most commonly used data types are "string", "int" and "float". The default data type is "string". Data types are usually used for type checking data.

```
undefine SLICE [KEY]
```

Removed a dictionary entry named KEY from SLICE. If no KEY is specified, then all entries in the specified slice are removed.

```
entries [SLICE]
```

With no arguments, this returns all slices in the current TOE object. If a SLICE is specified, then all the keys in that slice are returned.

```
exists SLICE KEY
```

Reports whether a particular dictionary entry exists, returning 1 for true, 0 for false.

```
lookup [-d DEFAULT][-t TYPE] SLICE KEY
```

Returns the value of the dictionary entry named by SLICE and KEY. An error results if the dictionary entry does not exist in the current object or its inherited parents. This error is suppressed if the optional DEFAULT is provided, in which case the DEFAULT is returned. If a TYPE is specified then the value is converted to that type. Failure to convert the data type results in an error.

```
ilookup {SLICES} KEY [DEFAULT]
```

The `instance lookup` command is the same as the standard `lookup` command except only the current TOE object is referenced. This is more efficient than lookup and can be used in cases where the parent classes of the object need not be referenced (for example, looking up runtime data or state information).

```
slookup {SLICES} KEY [DEFAULT]
```

The “scoped lookup” is like the standard lookup except that it also reference superior objects and their parents. This allows data to be retrieved from higher levels of the object tree in a manner similar to variable scoping in programming languages such as C. This is useful for accessing resource information pertinent to an entire tree of objects, but is the least efficient of the three lookup commands (`lookup`, `iloopup`, and `slookup`).

```
sdump  {SLICES}
```

Dumps all the key/value pairs associated with the specified slice to `stdout`.

```
promote SLICE KEY
```

Promotes the value of the dictionary entry named by SLICE and KEY into the current TOE object. This is used to pull dictionary values from parent classes into the instance before modifying them. This is equivalent to:

```
define SLICE KEY [lookup SLICE KEY]
slicelength SLICE
```

Returns the number of keys defined in SLICE in the current TOE object.

```
slicecopy SLICE1 SLICE2
```

Copies all keys from SLICE1 to SLICE2 in the current TOE object.

```
slicediff SLICE1 SLICE2
```

Compares two dictionary slices (in the current object) and returns three lists: added keys, removed keys and changed keys.

Object / Dictionary I / O

These two commands allow user to import and export the agent's data from and to a file or a certain source.

```
import string|interface SLICE SPEC [-class PARENTS][-exact]
```

Reads a configuration file and imports its contents into the current TOE object. For string imports, SPEC is the body of the configuration file. For interface imports, SPEC is the interface specification (usually the file name). The SLICE specification indicates that any keys in the configuration file not qualified by a slice name should be placed in SLICE.

If the configuration file contains hierarchical object specifications, these objects is created as inferiors of the current object (that is, a tree of objects is constructed below the current object). If PARENTS are specified then all new objects inherit from this object. If the `-exact` option is specified, then any subobjects in the configuration file specification must exactly match the existing object tree structure. Otherwise, it results in an error .

```
export string|interface SLICE SPEC [-minimal][-dot]
```

Exports the pertinent dictionary information from the current TOE object and any of its inferiors. The SLICE specification indicates which slice(s) are to be exported, and a blank slice specification causes all slices to be exported. Any subobjects are represented by a curly-brace-enclosed block in the resulting configuration file. The `-minimal` option suppresses as the output of subobjects not containing any pertinent dictionary entries (that is, empty config blocks). The `-dot` switch causes any subobject dictionary keys to be output in dot notation, which does not use curly braces.

Interactive Object Tree Navigation

These command enable users to interactively navigate the object tree. They are similar to UNIX shell commands for file system navigation.

```
cd name
```

Mimics the UNIX shell `cd` command when interactively navigating the TOE object tree. Navigating the TOE object tree is similar to UNIX shell `cd`. The functionality is actually more like the C shell `pushd` command since an object context stack is maintained. The special name ‘‘..’’ is used to pop the stack and return to the previous object. The ‘‘*’’ can be used as a shortcut to an unique object in the TOE object tree.

```
pwd
```

Returns the full name of the current object.

```
ls [-al] [names]
```

Lists the names of the subobjects under the current object. The `-a` option shows what (objects or data) and who (name) are under this object. The `-l` option outputs detailed information about each object, such as the object’s TOE identification. If names are specified, the subobject names (or detail information) are listed for each named subobjects; otherwise, it means to look into all subobjects.

Class Definition

These commands can be used to define class, create class, or destroy a class at runtime. They are not so useful in terms of debugging.

```
newclass name {parent classes} {body}
```

Creates a new class definition called name. The new class is subclassed from the parent classes (using multiple inheritance), and base classes can be created with empty set of parent classes. The body specification is a block of code, which defines the methods and the data of the new class, and is evaluated in the context of the class at the time of class creation.

```
defineclass name {body}
```

Extends the definition of a class. This command can be used to add methods or data to an existing class, and is useful for splitting large class definitions over multiple places.

```
method name {args} {body}
```

Creates a new method called name in the current class. The method command is identical to the Tcl “proc” command, but it also creates an association between the method and the class it is defined in.

```
instantiate class
```

Creates a new instance object of class. A new object is created which inherits from the class definition object, and the constructor is invoked on the new object.

```
uninstantiate object
```

Destroys the object instance. Before removing the object, any inferior objects in the object scope tree are uninstantiated, and the object’s destructor is invoked.

Class / Package Loading

These commands allow users to load classes or binary packages to an agent at runtime.

```
requires class name
```

Indicates to the interpreter that a particular class is required. The interpreter loads the class if it has not been loaded, and it automatically pulls in the entire ancestor hierarchy of the named class.

```
.class.name
requires package name
```

Indicates to the interpreter that a particular binary package is required. The interpreter loads the package if it has not been loaded, and it automatically loads in all packages the are required by the named package.

```
requires template name
```

This loads an X file as a template and makes it available as a reference tree. This is useful for loading tree specifications that are used by multiple other object trees. Reference tree can be inherited to produce usable object tree.

The loaded template is placed in the object tree at:

```
.template.name
```

Agent Interactive Mode Usage Examples

This section includes the following procedures:

- To Define a Module
- To Find the Attribute Value of a Certain Object
- To View the Result of an Operation on a Certain Object
- To Import and Export a Set of Object Attributes
- To Generate SNMP MIB From a Module

Note – Since the agent interactive mode is basically used for debugging, you can define new object attributes and introduce new modules. However, anything done in this mode is permanent.

▼ To Define a Module

The rest of this chapter uses a hypothetical example consisting of three files: `hellosunmc-v01-m.x`, `hellosunmc-v01-d.x` and `hellosunmc-v01-d.prc`. The contents of the files are given below:

```
hellosunmc-v01-m.x:
.........
# Location of the module
param:location =
.iso.org.dod.internet.private.enterprises.sun.prod.sunsymon\
.agent.modules.demo.HelloSunMCV01
param:oid = 1.3.6.1.4.1.42.2.12.2.2.1000.2
..........
hellosunmc-v01-d.x:
[ load hellosunmc-v01-m.x ]
HelloSunManagedObject = { [ use templates.hellosunmc-v01-models\
-d.HelloWorldManagedObject ]
        [ source hellosunmc-v01-d.prc ]
        type            = active
        refreshMode     = sync

        refreshService  = _internal
        refreshCommand  = getHS2String
        initInterval    = 2
        refreshInterval = 300
}
hellosunmc-v01-d.prc:
proc getHS2String { } {
    set count [lookup -d "0" visitCount count]
    incr count
    define visitCount count $count
    return "Hello World! This data is updated $count time(s)"
}
```

▼ To Find the Attribute Value of a Certain Object

If the TOE object is HelloSunManagedObject, its location is specified in the `hellosunmc-v01-m.x` file. To find its attribute:

1. **Navigate to the specific object.**

   a. Determine the current object:

```
agent> toe_self
```

   This displays the current TOE object id:

```
toe2
```

   b. Determine the objects directly under this object:

```
agent> ls
```

   This lists all objects under the current object, for example:

```
classes templates _config config services iso shadow contexts rules
```

   c. Determine the current object:

```
agent> pwd
```

   This gives you the full name of the objects.

   d. Go to the specified object:

   - To go to a certain object directly under the current object:

```
agent> cd iso
.iso
```

- For a short cut to an object whose name is unique in the current context:

```
agent:ios> cd *modules

.iso.org.dod.internet.private.enterprises.sun.prod.sunsymon.\
agent.modules

agent:modules> cd *HelloSunMCV01

.iso.org.dod.internet.private.enterprises.sun.prod.sunsymon\
.agent.modules.demo.HelloSunMCV01

agent:HelloSunMCV01> ls

availability enabled HelloSunManagedObject
agent:HelloSunMCV01> cd HelloSunManagedObject

.iso.org.dod.internet.private.enterprises.sun.prod.sunsymon\
.agent.modules.demo.HelloSunMCV01.HelloSunManagedObject
```

Using the `cd` command, several times, takes you beyond the top object. If this happens, you can use the following command:

```
agent> toe_begin toe2
```

This brings you back to the node you need to enter.

2. View all slices (logic groups) in the current object:

```
agent:HelloSunManagedObject> entries
```

This lists all slices in this object.

3. Enter the following to view the attribute value in a certain slice:

```
internal object peer timer timer-timer timer-attribute\
-timer target oid service serviceparam visitCount
agent:HelloSunManagedObject> sdump visitCount
```

This lists all key attributes/value pair in this slice:

```
count: 20
```

▼ To View the Result of an Operation on a Certain Object

1. **Invoke the method on the object:**

   a. **Go to the object domain where the method has been defined:**

```
agent> cd *HelloSunManagedObject
```

   The results are as follows:

```
.iso.org.dod.internet.private.enterprises.sun.prod.sunsymon\
.agent.modules.demo.HelloSunMCV01.HelloSunManagedObject
```

   b. **View current value of an attribute:**

```
agent:HelloSunManagedObject> sdump visitCount
```

   The results are as follows:

```
count: 24
```

   c. **Invoke the method:**

```
agent:HelloWorldManagedObjec> getHS2String
```

   The results are as follows:

```
Hello World! This data is updated 26 time(s)
```

d. View a certain attribute if it is set as a result:

```
agent:HelloSunManagedObject> lookup visitCount count
```

This returns the value of the set attribute, for example:

```
25
```

2. Apply the method on the object:

a. Apply operation on a certain object directly:

```
agent:modules> toe_send [locate *HelloSunManagedObject]
getHS2String
```

This applies the `getHS2String()` method to the specified object context and return the result:

```
Hello World! This data is updated 26 time(s)
```

b. View certain attributes if it is set as results:

i. Go to the specified object context:

```
agent:modules> cd *HelloSunManagedObject
```

ii. View the attribute values:

```
agent:HelloSunManagedObject> sdump visitCount
count: 26
```

▼ To Import and Export a Set of Object Attributes

To import and export a set of object attributes to and from a certain TOE object, do the following:

1. **Write all attributes (key/value pair) of a file in .x file format:**

```
shell% cat helloworld_v03_update.x
initValue  = 10
status     = "OK"
```

2. **Go to the target object:**

```
agent> cd *HelloSunManagedObject
.iso.org.dod.internet.private.enterprises.sun.prod.sunsymon\
.agent.modules.demo.HelloSunMCV01.HelloSunManagedObject
```

3. **View the current value of an attribute:**

```
agent:HelloSunManagedObject> sdump visitCount
count: 28
```

4. **Input data under this object to slice visitCount:**

```
agent:HelloSunManagedObject> import interface visitCount \
helloworld_v03_update.x
```

5. **View the attributes generated:**

```
agent:HelloSunManagedObject> sdump visitCount
count:28
initValue:10
status:OK
```

6. **Define a new attribute in this slice (or apply a certain operation):**

```
agent:HelloSunManagedObject> define visitCount time 16:23:10
16:23:10
```

```
agent:HelloSunManagedObject> sdump visitCount
count:28
initValue:10
status:OK
time:16:23:10
```

7. **Export these attributes to a file:**

```
agent> export interface visitCount helloworld_v03_update_out.x
```

8. **View the output file:**

```
shell% cat helloworld_v03_update_out.x
```

This command provides the following result:

```
count=28
initValue=10
status=OK
time=16:23:10
HelloSunMCMessage = {
}
```

▼ To Generate SNMP MIB From a Module

1. **Go to the destination module.**

   As defined in `hellosunmc-v01-m.x`, when this HelloWorld module is loaded, its location is:

```
.iso.org.dod.internet.private.enterprises.sun.prod.sunsymon\
.agent.modules.demo.HelloSunMCV01

agent> cd *HelloSunMCV01

.iso.org.dod.internet.private.enterprises.sun.prod.sunsymon\
.agent.modules.demo.HelloSunMCV01

agent:HelloSunMCV01> ls

availability enabled HelloSunManagedObject
```

2. **Generate SNMP MIB for these modules**:

```
agent:HelloSunMCV01> mibExport
```

3. **View the SNMP MIB files.**

   The mib files are located under the directory `$VAROPTDIR,` which is defined in the following: `$INSTALL_DIR/sbin/sm_setup_mib2.sh.` You can locate the `hellosunmc-v01-mib.txt` file at: the following location:

```
/var/opt/SUNWsymon/cfg/
```

CHAPTER 12

Developer Environment Tools

This chapter describes the following SNMP commands:

The following sections present the information in the format of man pages.

`snmpset`

Name

`snmpset`—Change information in manageable nodes with SNMP

Synopsis

```
/opt/SUNWsymon/util/bin/sparc-sun-solaris$VERSION/snmpset -h host \
[-p port] [-t timeout] [-i requestID] [-c community] name1 type1 \
value1 ... nameN typeN  valueN
```

Description

With `snmpset` you can change information on a manageable node if you have sufficient access privilege and the node is writable. You specify what information you want to change with the objectid. Then you specify the type of data.

Options

`-h` *host*

host is the hostname of the machine on which the agent is running. The hostname can be either the domain hostname or the IP address in dot notation.

`-p` *port*

Normally the remote port 161/UDP is used. If you run your snmp agent on an another port you can specify which port here. The default value is taken from environment variable *SNMP_PORT*. If this variable is not set then the value is taken from services file's entry snmp/udp.

`-t` *timeout*

Change the timeout time for each retry. Time is given in seconds. The default timeout is 5 seconds.

`-i` *requestID*

This integer value is used to differentiate different requests. The default requestID is 0.

`-c` *community*

This string will be used as community name. The default community is public.

name

name specifies the name of the managed property whose value is to be set. More than one name can follow. The values of all the names are set. If one of the specified names is invalid, then the request for all of the names fails.

type

type is the type of the value in the value field. Types can be:

```
None

Integer

OctetString

IPAddr

Opaque

Counter

Gauge

TimeTicks

ObjectId

Null
```

value

This is the new value of the node. More than one *name*, *type*, and *value* combination can follow.

Exit Status

Exit status is 0 if successful, In all other cases it is nonzero.

Examples of `snmpset`

```
CommandLine> snmpset -h ignite -p 161 -t 5 -i 100 -c foo "1.3.6.1.2.1.1.4.0"
"OctetString" "Administrators Name"

OutPut>Request Id: 100
    Error: noError
    Index: 0
    Count: 1

    Name: 1.3.6.1.2.1.1.4.0
    Kind: OctetString
    Value: "System Administrator"
```

```
CommandLine> snmpset -h ignite -p 161 -c foo 1.3.6.1.2.1.1.4.0 OctetString
"System Admin"

OutPut> Request Id: 0
    Error: noError
    Index: 0
    Count: 1

    Name: 1.3.6.1.2.1.1.4.0
    Kind: OctetString
    Value: "System Admin"
```

snmpget

Name

snmpget—Get information from manageable nodes with SNMP

Synopsis

```
/opt/SUNWsymon/util/bin/sparc-sun-solaris$VERSION/snmpget -h host \
[-p  port] [-t timeout] [-i requestID][-c community] [-q ] name1 \
... nameN
```

Description

With snmpget you can get information from a manageable node. You specify what information you want with the objectId. After doing this, enter the hostname where the agent is running. If the host (or other network connected device) can be reached, then you will get an answer.

Options

-h *host*

host is the hostname of the machine on which the agent is running. The hostname can be either the domain hostname or the IP address in dot notation.

-p *port*

Normally the remote port 161/UDP is used. If you run your snmp agent on another port you can specify which port here. The default value is taken from environment variable *SNMP_PORT*. If this variable is not set then the value is taken from services file's entry snmp/udp.

-t *timeout*

Change the timeout time for each retry. Time is given in seconds. The default timeout is 5 seconds.

`-i` *requestID*

This integer value is used to differentiate different requests. The default requestID is 0.

`-c` *community*

This string will be used as community name. The default community is public.

`-q` *name*

name specifies the name of the managed property whose value is returned. More than one name can follow. The values of all the names are returned. If one of the specified names is invalid, then the request for all of the names fails. If names are given without `-q` option then full detail output is listed and if `-q` option is specified then only value is listed.

Exit Status

Exit status is 0 if successful, In all other cases it is nonzero.

Examples of `snmpget`

```
CommandLine>snmpget -h symoncool -p 161 -t 5 -i 100 -c public \
-q 1.3.6.1.2.1.1.1.0

OutPut>    "Sun SNMP Agent, Ultra-1"

CommandLine>    snmpget -h ignite 1.3.6.1.2.1.1.1.0

OutPut>         Request Id: 0
                Error: noError
                Index: 0
                Count: 1

                Name: 1.3.6.1.2.1.1.1.0
                Kind: OctetString
                Value: "SUNW,Ultra-Enterprise"
```

```
CommandLine>  snmpget -h symoncool -p 161 -t 5 -i 100 -c public \
-q 1.3.6.1.2.1.1.2.0

OutPut>     1.3.6.1.4.1.42.2.1.1
```

```
CommandLine>  snmpget -h symoncool -q 1.3.6.1.2.1.1.1.0

OutPut>     "Sun SNMP Agent, Ultra-1"
```

```
CommandLine>  snmpget -h 129.146.53.224 -p 161 -t 5 -i 100 -c
public -q 1.3.6.1.2.1.1.1.0

OutPut>     "Sun SNMP Agent, Ultra-1"
```

```
CommandLine>  snmpget -h symoncool -p 161 -t 5 -i 100 -c public \
-q 1.3.6.1.2.1.1.1.0 1.3.6.1.2.1.1.2.0 1.3.6.1.2.1.1.3.0
1.3.6.1.2.1.1.4.0 1.3.6.1.2.1.1.5.0

OutPut>     “Sun SNMP Agent, Ultra-1”
             1.3.6.1.4.1.42.2.1.1
             9330516
             “System administrator” “symoncool”
```

```
CommandLine>  snmpget -h symoncool -p 161 -t 5 -i 100 -c public  \
-q 1.3.6.1.2.1.1.10.0

OutPut>     “”
```

```
CommandLine>  snmpget -h symoncool -p 161 -t 5 -i 100 -c public  \
-q 1.3.6.1.2.1.1.1.0 1.3.6.1.2.1.1.2.0 1.3.6.1.2.1.1.3.0
1.3.6.1.2.1.1.4.0 1.3.6.1.2.1.1.10.0

OutPut>     “”
            “”
            “”
            “”
            “”
```

snmpnext

Name

snmpnext—Get information from manageable nodes with SNMP

Synopsis

```
/opt/SUNWsymon/util/bin/sparc-sun-solaris$VERSION/snmpnext -h host
[-p port] [-t timeout] [-i requestID] [-c community]  name1 ... nameN
```

Description

With snmpnext you can get information from a manageable node. You specify what information you want with the objectId. Then enter the name of the host on which the agent is running.

If the host (or other network connected device) can be reached, you will get the information about the next object id.

Options

-h *host*

host is the hostname of the machine on which the agent is running. The hostname can be a domain hostname or an IP address in dot notation.

-p *port*

Normally the remote port 161/UDP is used. If you run your snmp agent on another port you can specify the port here. The default value is taken from environment variable SNMP_PORT. If this variable is not set, then the value is taken from the services file entry snmp/udp.

-t *timeout*

Change the timeout time in seconds for each retry. The default timeout is 0 seconds, which means there is no timeout.

-i *requestID*

This integer value is used to differentiate different requests. The default requestID is 0.

-c *community*

This string is used as the community name. The default community is public.

name

name specifies the name of the previous managed property for which a value is returned. Multiple names can follow. The value of the next managed object of all the names is returned. If one of the names is invalid, the request fails for all of the names.

Exit Status

Exit status is 0 if successful. In all other cases it is nonzero.

Examples of `snmpnext`

```
CommandLine>snmpnext -h symoncool -p 161 -t 5 -i 1 -c \
public 1.3.6.1.2.1.1.1.0

OutPut>Request Id: 1
Error: noError
Index: 0
Count: 1

Name: 1.3.6.1.2.1.1.2.0
Kind: ObjectId
Value: 1.3.6.1.4.1.42.2.1.1
```

```
CommandLine>snmpnext -h symoncool 1.3.6.1.2.1.1.0

OutPut>Request Id: 0
    Error: noError
    Index: 0
    Count: 1

Name: 1.3.6.1.2.1.1.1.0
     Kind: OctetString
     Value: “Sun SNMP Agent, Ultra-1”
```

```
CommandLine>snmpnext -h symoncool -p 161 -t 5 -i 1 -c  public 1.

OutPut>Request Id: 1
    Error: noError
    Index: 0
    Count: 1

    Name: 1.3.6.1.2.1.1.1.0
    Kind: OctetString
    Value: “Sun SNMP Agent, Ultra-1”
```

snmptrap

Name

snmptrap—Send an SNMP TRAP message to a host

Synopsis

```
/opt/SUNWsymon/util/bin/sparc-sun-solaris$VERSION/snmptrap \
-h fhost [-p fport] [-c community] enterprise agent_addr \
generic_trap specific_ trap timestamp \
[name1 type1 value1 ... nameN typeN valueN ] ...
```

Options

-h *host*

host is the hostname of the machine to which TRAP is to be sent.

-p *port*

Port to which the TRAP is to be sent. The default value is taken from environment variable SNMP_PORT. If this variable is not set, then the value is taken from the services file entry `snmp-trap/udp`.

-c *community*

This string is used as the community name. The default community is public.

enterprise

This option can point to a mib sub tree or identify a product for which the TRAP has been defined.

agent_addr

Change the address from which the trap reports it is being sent. By default, `snmptrap` uses the address of the sending host.

`generic_trap`

An integer that specifies the type of trap message being sent. Trap types are defined below.

`specific_trap`

An integer that specifies the enterprise specific trap. For this the generic trap field must be 6.

`timestamp`

Time elapsed between the last (re)initialization of the network entity and the generation of the trap.

`name`

Object Id (OID) of the extra information to be sent with TRAP.

`type`

Type of the value in the value field. Types can be:

```
None

Integer

OctetString

IPAddr

Opaque

Counter

Gauge

TimeTicks

ObjectId

Null
```

`value`

Value for the OID given in the name field

Exit Status

Exit status is 0 if successful. In all other cases it is nonzero.

Trap Type Information

The following table presents trap types and what they signify. The meaning of the numbers is illustrated within parenthesis.

TABLE 12-1 Trap Type and What it Signifies

Trap Type	What the Sending Protocol Entity Signifies
0 (coldStart)	Is reinitializing itself. The agent's configuration or the protocol entity implementation may be altered.
1 (warmStart)	Is reinitializing itself. Neither the agent's configuration nor the protocol entity implementation is altered.
2 (linkDown)	Recognizes a failure in one of the communication links represented in the agent's configuration.
3 (linkUp)	Recognizes that one of the communication links represented in the agent's configuration has come up.
4 (authenticationFailure)	Is the addressee of a protocol message that isn't properly authenticated.
5 (egpNeighborLoss)	Had an EGP neighbor as an EGP peer, and the neighbor has been marked down so that the peer relationship no longer exists.
6 (enterpriseSpecific)	Recognizes that some enterprise-specific event has occurred. The specific trap field identifies the particular trap that occurred.

Examples of `snmptrap`

```
CommandLine>snmptrap -h symoncool -p 2000 -c \
public "1.3.6.1.4.1.42" 129.146.53.224 6 3 12345
```

This example sends a trap on port 2000 on host symoncool with community = public, agent = 129.146.53.224, timestamp = 12345, enterprise = 1.3.6.1.4.1.42, generic trap = 6, and specific trap = 3.

```
CommandLine>snmptrap -h symoncool -p 2000 -c foo \
“1.3.6.1.4.1.42” 129.146.53.224 6 3 12345
```

This example sends a trap on port 2000 on host symoncool with community = foo, agent = 129.146.53.224, timestamp = 12345, enterprise = 1.3.6.1.4.1.42, generic trap = 6, and specific trap = 3.

snmpwalk

Name

snmpwalk—Query for a tree of information about a network entity

Synopsis

```
/opt/SUNWsymon/util/bin/sparc-sun-solaris$VERSION/snmpwalk -h fhost
[-p fport] [-t timeout] [-i requestId] [-c community]name1 ... nameN
```

Description

With snmpwalk you can get information from an manageable node. You specify what information you want with the objectId. Then, enter the name of the host on which the agent is running. If the host (or other network connected device) can be reached, then you will get information about all the entries in the subtree below the objectId specified in *name*. snmpwalk walks the subtree in lexicographical order.

Options

-h *host*

host is the hostname of the machine on which the agent is running. The hostname can be a domain hostname or IP address in dot notation.

-p *port*

Normally the remote port 161/UDP is used. If you run your snmp agent on another port, you can specify which port here. The default value is taken from environment variable SNMP_PORT. If this variable is not set, then the value is taken from the services file entry `snmp/udp`.

-t *timeout*

Change the *timeout* time in seconds for each retry. The default *timeout* value is 10 seconds.

-i *requestID*

This integer value is used to differentiate different requests. The default *requestID* is 0.

-c *community*

This string is used as a community name. The default community is public.

name

The portion of the object identifier space that will be searched, using GET NEXT requests. The *snmpwalk* utility queries all variables in the subtree below the specified variable and displays their values.

Exit Status

Exit status is 0 if successful. In all other cases it is nonzero.

Examples of `snmpwalk`

```
CommandLine snmpwalk -h symoncool -p 161 -c public 1.3.6.1.2.1.1
OutPut>       Sun SNMP Agent, Ultra-1
              1.3.6.1.4.1.42.2.1.1
               36472928
               System administrators office
               72
```

Note – If there is no subtree with the specified OID or if the OID is invalid, then `snmpwalk` returns without printing anything.

```
CommandLine>snmpwalk -h symoncool -p 161 -t 1 -c foo  1.3.6.1.2.1.2

OutPut>Request Timed Out: snmpwalk -h symoncool -p 161 -t 1 -c foo
1.3.6.1.2.1.2
```

Note – If you do not have sufficient access privilege and there is no timeout, `snmpwalk` just returns without printing anything.

snmpwalktable

Name

`snmpwalktable`—Query for a table of information about a network entity

Synopsis

```
/opt/SUNWsymon/util/bin/sparc-sun-solaris$VERSION/snmpwalktable\
-h fhost [-p fport] [-t timeout] [-i requestId] [-c community]\
[-n numcolumns] [-w width] name1 ... nameN
```

Description

With `snmpwalktable` you can get information from the table node. You specify which table which you want information about by specifying the objectId. Then enter the hostname where the agent is running. If the host (or other network connected device) can be reached, you will get an answer. This command walks from the start of the table to the end of the table in lexicographical order.

-h *host*

host is the hostname of the machine on which the agent is running. The hostname can be a domain hostname or an IP address in dot notation.

-p *port*

Normally remote port 161/UDP is used. If you run your snmp agent on another port you can specify which port here. The default value is taken from environment variable SNMP_PORT. If this variable is not set, then the value is taken from the services file entry snmp/udp.

-t *timeout*

Change the timeout time for each retry in seconds. The default timeout is 10 seconds.

-i *requestID*

This integer value is used to differentiate different requests. The default requestID is 0.

-c *community*

This string is used as the community name. The default community is public.

-n *numcolumns*

This integer value shows the number of columns to be printed. The default value is 1.

-w *width*

This integer value indicates the width of every column. The default width is 10.

name

The ObjectId of the table

Exit Status

Exit status is 0 if successful, In all other cases it is nonzero.

Examples of snmpwalktable

```
CommandLine>   snmpwalktable -h symoncool -p 161 1.3.6.1.2.1.4.20

OutPut>           127.0.0.1
                            129.146.53.224
                            1
                            2
                                255.0.0.0
                                255.255.255.0
                                   1
                                   1
                                65535
                                65535
```

```
CommandLine>  snmpwalktable -h symoncool -p 161 -n 5 -w 3 \
1.3.6.1.2.1.4.20

OutPut>       127 1   255 1   655
              129 2   255 1   655
```

```
CommandLine>    snmpwalktable -h symoncool -p 161 -n 5 \
1.3.6.1.2.1.4.20

OutPut> 127.0.0.1  1       255.0.0.0  1       65535    129.146.53
2          255.255.25 1          65535
```

```
CommandLine>     snmpwalktable -h symoncool -p 161 -n 5 -c foo \
-t 1  1.3.6.1.2.1.4.20

OutPut>          Request Timed Out: snmpwalktable -h symoncool \
-p 161 -n 5 -c foo  -t 1 1.3.6.1.2.1.4.20
```

Note – If the access privilege is insufficient and no timeout occurs, `snmpwalktable` returns without printing anything. Also, the command does not print anything if the given OID is not a table.

PART II Programmer's Reference to Console Integration and Client API

This volume includes the following sections:

CHAPTER 13

Console Integration

This chapter covers the following topics:

- Extending the Console—page 241
- Integrating Sun Management Center Software With Other Management Tools—page 246
- Compilation and `makefile` Guidelines—page 255

This chapter contains instructions and examples for integrating the Sun Management Center Console with other applications. Two significant aspects of integration are considered:

- enabling the Console to launch user-specific applications
- enabling other management tools to utilize features provided by the Console

This chapter covers the following topics:

- Extending the Console—page 241
  - Integration Levels—page 242
  - Configuration Files—page 243
  - Update Utilities—page 246
- Integrating Sun Management Center Software With Other Management Tools—page 246
- Compilation and `makefile` Guidelines—page 255

Extending the Console

The Sun Management Center Console can be extended to include additional functionality as users deem necessary. Support for the integration of applications providing such functionality is through two configuration files (`console-tools.cfg` and `console-host-apps.cfg`) and two utility scripts

(`es-tool` and `es-apps`) that update the environment based on the content of the configuration files. Integration levels and syntax for entries in the configuration files are discussed in this section.

The general process for extending the Console is as follows:

1. Place the Java class files or the `.jar` files for the user applications in the standard Sun Management Center location:

   ```
   /opt/SUNWsymon/apps/classes/
   ```

   If there are any auxiliary files, such as, property files or images, they should also be placed in either this directory or a `.jar` file.

2. Select the appropriate integration level to determine the appropriate configuration file and utility.

3. Modify the appropriate configuration file to describe the desired extension.

4. Run the appropriate update utility.

5. Restart the Console. Depending upon your site's configuration, it may be necessary to restart the Server as well.

Integration Levels

Sun Management Center 2.1 supports the following levels of application integration:

- *External Java application with no API or access to the current host selection*: This level applies to external, stand-alone applications written in Java that the user wishes to launch from the Console. The application must be written in Java; non-Java applications are accommodated by the `SMSystemCommand` Java wrapper class that executes other programs or shell scripts in a separate process. For a description of `SMSystemCommand`, refer to the section on "Syntax for Entries in the console-tools.cfg File" on page 243.

  The application does not require any host selection context or Client API handle. The application has no further interaction with Sun Management Center once it has been launched. The name of the application (as specified in the `console-tools.cfg` file) will be appended to the Tools Menu in the Console below the standard menu items provided by the Sun Management Center. Support for this integration level is provided through the `console-tools.cfg` file and the `es-tool` update utility.

- *External Java application with API and host selection context*: This level applies to custom Java applications that the user wishes to launch from the Console. The application runs under the same JVM as the Console. The application requires a pre-authenticated Client API handle and additional global user information. The application may also require access to the host selection context. The name of the

application (as specified in the `console-tools.cfg` file) will be appended to the Tools Menu in the Console below the standard menu items provided by the Sun Management Center. The GUI for the application is displayed in a separate window. Support for this integration level is provided through the `console-tools.cfg` file and the `es-tool` update utility.

- *Internal Java application utilizing the host details context*: This level applies to custom Java applications that the user wishes to launch for a particular host that has been selected in the Browser tab or the Topology display. The application requires access to the Host selection context. The name of the application (as specified in the `console-host-apps.cfg` file) will be added to the tree list displayed in the Applications tab in the Host Details window. The application may or may not have an associated GUI; if it does have a GUI, the GUI will be displayed within the Display portion of the Host Details window when the application is launched. Support for this integration level is provided through the `console-host-apps.cfg` file and the `es-apps` update utility.

Note – User applications may wish to implement the `SMApp` interface or extend the `SmAppBase` class in order to obtain access to the agent information. Refer to the `javadocs` for specific information.

Configuration Files

The `console-tools.cfg` and `console-host-apps.cfg` files are plain text files that can be edited with any standard text editor. The files reside on the host on which the Server is running. The files may be modified at any time (including while the Console is running), but changes introduced by editing the files will not take effect until the appropriate update utility is run on the server host and the Console is restarted (depending upon site configuration, it may be necessary to restart the Server as well). Each file consists of a series of lines, each of which describes an application; blank lines and lines beginning with the pound sign '#' are ignored. Fields within each line are separated by commas ','.

Syntax for Entries in the console-tools.cfg File

Applications listed in the `/var/opt/SUNWsymon/cfg/console-tools.cfg` file are launched from the Tools menu in the Console main window. Each application is defined by a line with the following format:

```
menu_label,class [args]
```

Entry fields are as follows:

- *menu label* — the string that will appear in the Tools menu. The string can be either *unlocalized* or *localized* text. Unlocalized text can contain embedded spaces. Localized text must be specified as a *property_file*:*key* pair, where *property_file* is the name of the file containing the localizable messages for a particular locale, and *key* is the identifier used to locate the string that will appear in the Tools menu in the property file. Note that spaces are not allowed in the key. Refer to chapter 17 for a discussion of the syntax for the property file.
- *class* — the fully qualified Java class name.
- *args* — the list of arguments to the class.

The example file below shows entries for three applications to be listed on the Tools menu: `Example GUI`, `telnet`, and `ftp`:

```
# Format:
# menu_label,class arguments
Example GUI,exampleApp.ExampleGUITool

Telnet,com.sun.symon.base.client.console.SMSystemCommand \
"/usr/openwin/bin/xterm -e telnet $host" "start telnet $host"

exampleApp.ExampleSystemCommand:ftp,exampleApp.ExampleSystemCommand \
"/usr/openwin/bin/xterm -e ftp $host" "start ftp $host"
```

A special built-in Java wrapper class `SystemCommand` is provided to enable the user to execute an arbitrary shell command. This class takes two arguments. The first argument is the shell command to execute. If a program name is specified, a full path should be given. If there are embedded spaces, the entire shell command should be enclosed in double quotes. The second argument is the command to run if on a Microsoft Windows client, in which case the first argument is ignored.

```
com.sun.symon.base.client.console.SystemCommand \
shell_command  windows_command
```

Variable substitution is performed on the arguments if special variable references are present. The allowed variables are given below:

`$host`	replaced with currently selected agent host name
`$port`	replaced with currently selected agent port number

Syntax for Entries in the console-host-apps.cfg File

Applications listed in the `/var/opt/SUNWsymon/cfg/console-host-apps.cfg` file are launched from the Applications tab in the Host Details window. Each application is defined by a line with the following format:

```
menu_label,class,[args],[help]
```

Entry fields are as follows:

- *menu label* — the string that will appear in the Applications tab. The string can be either *unlocalized* or *localized* text. Unlocalized text can contain embedded spaces. Localized text must be specified as a *property_file*:*key* pair, where *property_file* is the name of the file containing the localizable messages for a particular locale, and *key* is the identifier used to locate the string that will appear in the Applications tab in the property file. Note that spaces are not allowed in the key. Refer to chapter 17 for a discussion of the syntax for the property file.
- *class* — the fully qualified Java class name.
- *args* (optional) — the list of arguments to the class.
- *help* (optional) —The help file is specified in one of two forms. The first form is a *key*:*file* pair, where *key* is a unique string that associates the help file with the application, and *file* is the name of the HTML file containing the help for the application. (Note that this is only the file name, not the full path name of the HTML file, and that this HTML help file must be installed on the Sun Management Center help server). The second form is :*url*, where *url* is any URL specification.

Note – In the case where there are no arguments to the class but you wish to specify a help file, you must still indicate the (empty) arguments field with a comma.

The example file below lists two hypothetical applications. The first application has arguments but no help file; the second application has a help file but no arguments.

```
# example 1: arguments but no help file
example.ExampleModuleApp1:ExampleApp1, example.ExampleApp1, arg1 arg2 arg3

# example 2: help file but no arguments
example.ExampleModuleApp2:ExampleApp2, example.ExampleApp2, key:file
```

Update Utilities

The integration configuration files can be edited at any time; however, in order for the changes specified in the file to be propagated to the Server, the appropriate update utility must be run on the Server host. For each configuration file, there is a corresponding update utility: for the `console-tools.cfg` file, the update utility is `/opt/SUNWsymon/sbin/es-tool`; for the `console-host-apps.cfg` file, the update utility is `/opt/SUNW/symon/sbin/es-apps`. If you wish to use an alternative configuration file, you can specify the name as an argument to the utility

The list of user applications is generated from these configuration files each time these scripts are run. Removing an entry from a configuration file will cause it to be also removed from the list.

Integrating Sun Management Center Software With Other Management Tools

Sun Management Center software is an enterprise-wide management solution for managing Sun platforms. Any other management solution that wants to use Sun Management Center software for management of a Sun platform can achieve this using a utility Bean that is provided as a part of the Sun Management Center Developer Environment. This Bean is a Sun Management Center Object Details window that can display details without requiring the user to navigate in the topology view in the Sun Management Center main console.

The class specification of this Bean is:

```
package name: com.sun.symon.apps.details.hostdetailsBean
Class HostdetailsBean
java.lang.Object
    |
+--com.sun.symon.tools.hostdetailsBean.HostdetailsBean
public class HostdetailsBean extends java.lang.Object
```

HostdetailsBean is a Bean that launches an Sun Management Center 2.1 Host Details window for a given Sun Management Center agent system monitored by a specified Sun Management Center server. This Bean uses classes and configuration files that are part of the Sun Management Center 2.1 installation and therefore work only when the Sun Management Center 2.1 console and server installation exists. Note that the Bean must be started in a separate thread.

▼ To Invoke the HostDetailsBean

1. **Instantiate a HostdetailsBean object:**

```
HostdetailsBean theBean = new HostdetailsBean();
```

2. **Initialize Bean parameters:**

```
theBean.init(String,String,String,String,String,null);
```

3. **Optionally, subscribe for PropertyChangeEvents from the Bean:**

```
theBean.addPropertyChangeListener(this);
```

4. **Optionally, set the exit() method of the Bean:**

```
theBean.setExitAction(Object, String, Object[])
```

5. **Launch hostdetails:**

```
theBean.doLaunchHostdetails();
```

PropertyChangeEvent events are fired when exceptions from the Sun Management Center client API are caught. It is the responsibility of the Bean user to listen for these events and take appropriate action. Also, PropertyChangeEvents are fired to relay informational (status) messages that could be reflected in a GUI or used to control the Bean environment.

Field Summary

```
static java.lang.String ERROR_NO_TARGET
    Bounded property value indicating target system not found
static java.lang.String HOSTDETAILS_ERROR
     Bounded property-name for notification of Exception conditions
static java.lang.String HOSTDETAILS_STATUS
    Bounded property-name for notification of status conditions
static java.lang.String SECURITY_SCHEME
    security scheme
static java.lang.String STATUS_AUTHENTICATION_OK
    Bounded property value indicating hostdetails is exiting without error
static java.lang.String STATUS_CONNECTION_OK
   Bounded property value indicating hostdetails is exiting without error
static java.lang.String STATUS_EXITING_OK
    Bounded property value indicating hostdetails is exiting without error
static java.lang.String STATUS_STARTUP_OK
    Bounded property value indicating hostdetails started okay
```

Constructor Summary

```
HostdetailsBean()
```

Method Summary

```
void
addPropertyChangeListener(java.Beans.PropertyChangeListener listener)
```

Convenience method for subscribing an object to PropertyChangeEvents. This must be done after init() is called.

```
void doExitAction()
```

Bean exit method.

```
void doLaunchHostdetails()
```

This method communicates with the Sun Management Center server and launches the Hostdetails window.

```
void init(java.lang.String sName, int sPort,
java.lang.StringtargetHost, java.lang.String user,
java.lang.String pass, java.lang.String key)
```

This method initializes values needed for proper operation of the Bean.

```
void propertyChange(java.Beans.PropertyChangeEvent evt)
```

This method is responsible for notifying subscribers of property changes within the Bean.

```
void
removePropertyChangeListener(java.Beans.PropertyChangeListener
listener)
```

Convenience class for removing an object subscription to PropertyChangeEvents. This must be done after `init()` is called.

```
void setExitAction(java.lang.Object target, java.lang.String
method, java.lang.Object[] myArgs)
```

This method allows specification of an alternate method to execute when the hostdetails window is closed.

```
void setHostname(java.lang.String host)
```

Mutator method for setting the Sun Management Center server hostname.

```
void setHostport(int port)
```

Mutator method for setting the Sun Management Center server port.

```
void setPassword(java.lang.String pass)
```

Mutator method for setting the Sun Management Center server password.

```
void setPublicKey(java.lang.String key)
```

Mutator for Sun Management Center server public crypto key. If set to null, defaults to an internal value.

```
void setTarget(java.lang.String targ)
```

Mutator method for setting the target system.

```
void setUser(java.lang.String user)
```

Mutator method for setting the Sun Management Center server login username.

Field Detail

```
public static final java.lang.String HOSTDETAILS_ERROR
```

Bounded property-name for notification of exception conditions

```
public static final java.lang.String HOSTDETAILS_STATUS
```

Bounded property-name for notification of status conditions

```
public static final java.lang.String STATUS_STARTUP_OK
```

Bounded property value indicating hostdetails started okay

```
public static final java.lang.String STATUS_EXITING_OK
```

Bounded property value indicating hostdetails is exiting without error

```
public static final java.lang.String STATUS_CONNECTION_OK
```

Bounded property value indicating hostdetails is exiting without error.

```
public static final java.lang.String STATUS_AUTHENTICATION_OK
```

Bounded property value indicating hostdetails is exiting without an error.

```
public static final java.lang.String ERROR_NO_TARGET
```

Bounded property value indicating target system not found

```
public static final java.lang.String SECURITY_SCHEME
```

Security scheme.

```
public static final boolean WANT_ENCRYPTION
```

Encryption

Constructor Detail

```
public HostdetailsBean()
```

Empty constructor for Bean compliance

Method Detail

```
public void init(java.lang.String sName, int sPort,
java.lang.String targetHost, java.lang.String user,
java.lang.String pass, java.lang.String key)
```

This method initializes values needed for proper operation of the Bean. Mutators are available for these fields in addition to this method.

Parameters:

`sName` - Hostname of the Sun Management Center server

`sPort` - Port number used by the Sun Management Center server

`targetHost` - Sun Management Center “label name” of the system to be displayed

`user` - Sun Management Center server login name

`pass` - Sun Management Center server password for above user

`key` - Sun Management Center server public key (defaults if null)

```
public void setHostname(java.lang.String host)
```

Mutator method for setting the Sun Management Center server hostname.

Parameters:

`host` - Sun Management Center server hostname

```
public void setHostport(int port)
```

Mutator method for setting the Sun Management Center server port.Parameters:

`port` - Sun Management Center server port number (probably 2099)

```
public void setTarget(java.lang.String targ)
```

Mutator method for setting the target system; that is, the system for which you would like hostdetails displayed.

Parameters:

`targ` - target system name, as known to Sun Management Center as “label name”

This is *not* the same as hostname. It is the Sun Management Center label name.

```
public void setUser(java.lang.String user)
```

Mutator method for setting the Sun Management Center server login username.

Parameters:

`user` - username for login to Sun Management Center server

```
public void setPassword(java.lang.String pass)
```

Mutator method for setting the Sun Management Center server password.

Parameters:

`pass` - password for login to Sun Management Center server

```
public void setPublicKey(java.lang.String key)
```

Mutator for Sun Management Center server public crypto key If set null, this will default to an internal value.

Parameters:

`key` - Sun Management Center server public key.

```
public void propertyChange(java.Beans.PropertyChangeEvent evt)
```

This method is responsible for notifying subscribers of property changes within the Bean.

Parameters:

`evt` - PropertyChangeEvent to be fired

```
public void
addPropertyChangeListener(java.Beans.PropertyChangeListener
listener)
```

Convenience method for subscribing an object to PropertyChangeEvents. This must be done after `init()` is called.

Parameters:

`listener` - a PropertyChangeListener to subscribe for events

```
public void removePropertyChangeListener(java.Beans.PropertyChangeListener listener)
```

Convenience class for removing an object subscription to PropertyChangeEvents. This must be done after `init()` is called.

Parameters:

`listener` - PropertyChangeListener to remove from subscription list

```
public void doExitAction()
```

Convenience method to do exit actions.

```
public void setExitAction(java.lang.Object
target,java.lang.String method, java.lang.Object[] myArgs)
```

Throws

```
java.lang.IllegalArgumentException
```

1. This method allows specification of an alternate method to execute when the hostdetails window is closed. By default the Bean will call `System.exit(0)`.

Parameters:

target - Object which contains the method to be called

method - Method name to be called

argv - String array containing method arguments

Throws:

```
java.lang.IllegalArgumentException
```

if introspection/invocation would fail.

```
public void doLaunchHostdetails()
```

Throws:

```
java.lang.IllegalStateException
```

This method communicates with the Sun Management Center server and launches the Hostdetails window. If you did not initialize all fields correctly through `init()`, you may receive an IllegalStateException.

Throws:

```
java.lang.IllegalStateException - if any required fields are null.
```

Compilation and `makefile` Guidelines

For compilation of JavaBeans™, the Java `CLASSPATH` must be set. Typically the `CLASSPATH` should be set to the following:

```
ESROOT = /opt/SUNWsymon
CLASSPATH = \
        $(ESROOT)/jclass/chart/lib/jcchart362J.jar:\
        $(ESROOT)/classes/essrv.jar:\
        $(ESROOT)/classes/esclt.jar:\
        $(ESROOT)/classes/escon.jar:\
        $(ESROOT)/classes/escom.jar:\
        $(CLASSPATH)
```

For an example of makefile, refer one the console examples under `/opt/SUNWsymon/sdk/examples/console`.

CHAPTER 14

Client API

This chapter covers the following topics:

Refer to the `javadocs`, included in the Developer Environment CD, for details on the Client classes, their methods and descriptions, including examples. Once the product image is installed, the most up-to-date information on where the `javadocs` reside will be available in the following HTML file:

`/opt/SUNWsymon/sdk/docs/index.html`

Introduction to Client API Classes

The Sun Management Center Client Application Programming Interface (API) contains a set of public Java classes. The Sun Management Center console developers or Java applications can use these classes to retrieve data from the Sun Management Center server or the Sun Management Center agent.

The client API is packaged as a Java jar file, `esclt.jar`. This file is located in the following directories depending on the operating system or platform:

- Solaris - `/opt/SUNWsymon/classes`
- Windows - `c:\Program Files\symon2.0\esymon`

Client API classes are all part of `com.sun.symon.base.client` package.

Note – The audience for this document is programmers who have knowledge of object-oriented language and Java. This document does not explain object-oriented fundamentals.

API Usage for System Management

This chapter includes many examples that can help developers get started with the Client API. These samples introduce the concepts of Request, Response, and Data classes. This section also contains information on how synchronous and asynchronous methods are programmed. Once the concepts are explained for a particular request class, the same principle applies for the various request class categories. An explanation of the functionality of the request and response classes and interfaces follow. For more information and other details on the Client API classes, refer to “Java Language Object Class Examples” on page 262.

External Interface Requirements

The Client API is built on pure Java code. The JRE version is bundled with the Sun Management Center console packages. This is available on Solaris software as well as Windows versions. The Client API does not have any other external interface requirements besides the Java 1.2 package.

Sun Management Center Architecture

This section includes the following:

Sun Management Center Three-Tier Architecture

The Sun Management Center server stands between the Sun Management Center console on one end and the Sun Management Center agents on the other. FIGURE 14-1 illustrates this configuration:

- Sun Management Center agents provide the data required for manageability.
- Sun Management Center console components provide the system monitoring, control, and configuration user interfaces.
- The Sun Management Center server acts as a request broker between the agent and the console. It provides a set of low-level interfaces in the console receptor module. These interfaces provide an access to the agent data for read/write purposes. They are extremely generic and deal with raw untyped data. This document does not contain information on the server.

The Sun Management Center Client API is built on top of these interfaces to provide a higher level of management functionality for the Sun Management Center console. Using this API, the console applications fetch *live* or *historic* data to configure the system dynamically.

The API supports both synchronous and asynchronous data models. The data provided by the Sun Management Center server can be instantaneous, historic data values retrieved from the Sun Management Center agents.

The following illustration represents the relationship of the Client API to the product. This illustration is followed by another which represents the Sun Management Center architecture and the Client API's relationship to the Sun Management Center software.

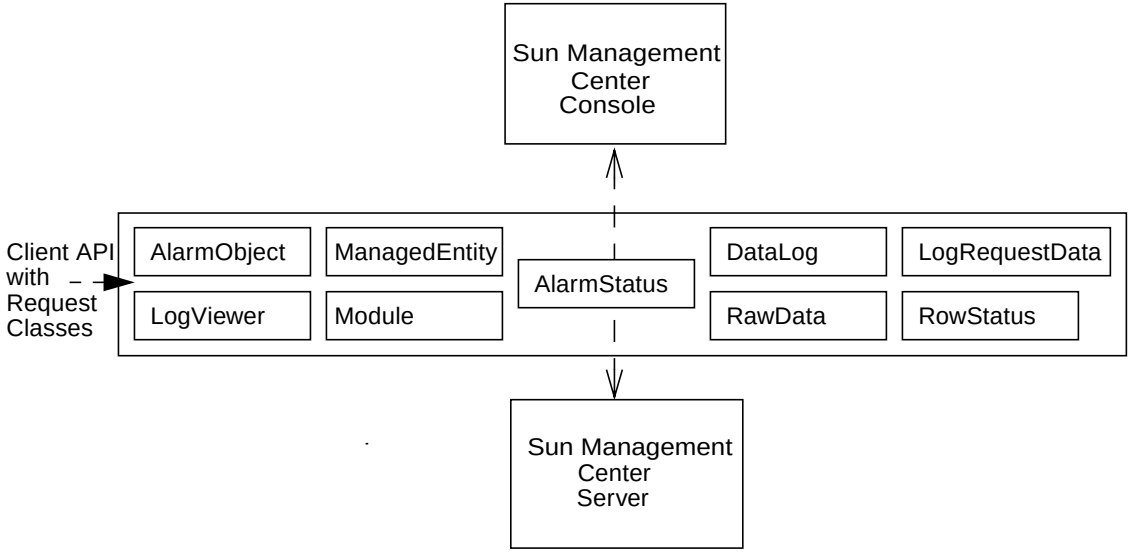

FIGURE 14-1 Client API Request Classes in Relationship With the Console and Server

The following figure shows:

- Console / GUI client using the client API to connect to the server
- Client API using RMI as a transport to the server
- Sun Management Center server components
- Sun Management Center agents in a server context with one or more modules loaded

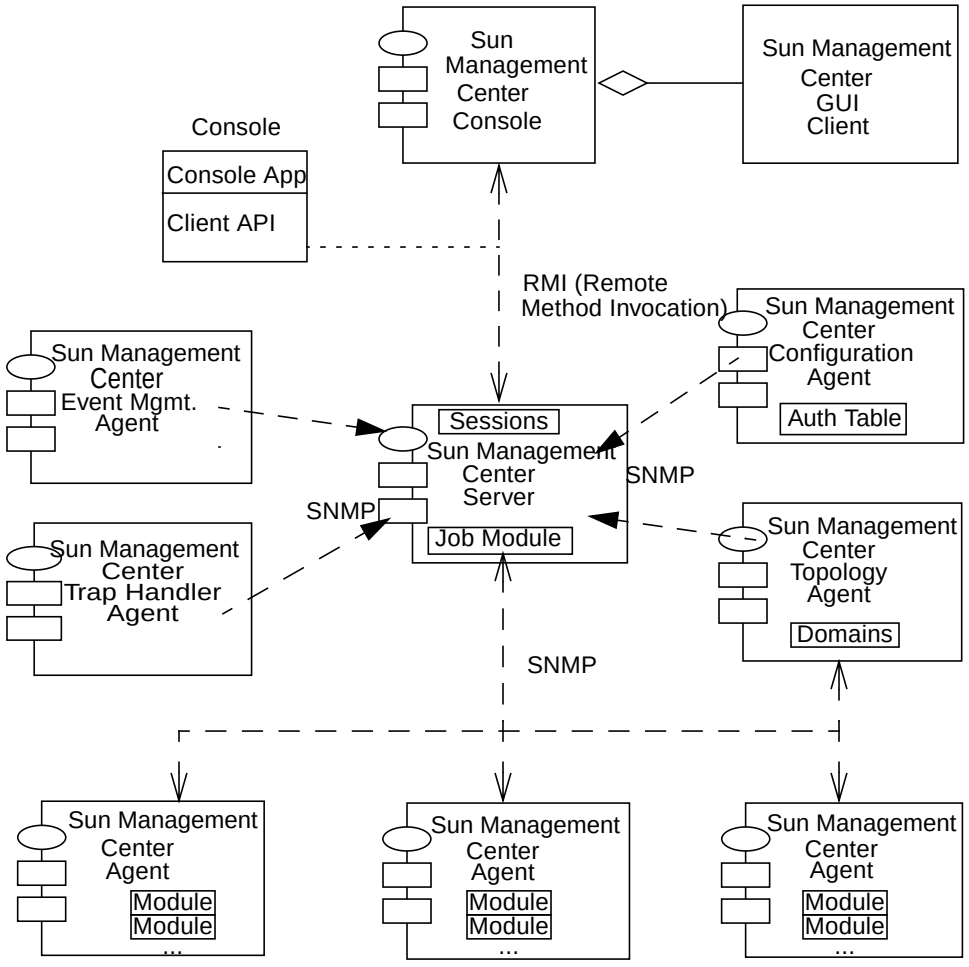

FIGURE 14-2 The Client API and the Sun Management Center Architecture

Client API Class Usage

This section covers the API Class usage for Sun Management Center Developer Environment. It includes samples to help users use the Client API. This chapter contains the following sections:

Client API Definition

The Sun Management Center console components have certain shared resources, like images for various hardware platforms that are stored on the Sun Management Center server. The console components download these resources using the Client API. The Client API is the only source of data for the Console components.

For the Sun Management Center software, the transport protocol between the Client API and the Sun Management Center server is based on RMI (Remote Method Invocation). According to Sun Management Center architecture standards, this could also be based on TCP or some other transport mechanism.

This package contains various classes each catering to a particular category of information being retrieved. There are classes to retrieve alarms, view log files, initiate discovery requests, create domains and so forth. These classes are bundled with the rest of the console-related classes in the form of a jar file.

The API provides support for both synchronous and asynchronous requests.

- The synchronous requests block till the data is retrieved from the agent/server.
- The asynchronous requests return immediately, without returning any data. The user requesting the asynchronous data provides a reference that gets called once the data is available or an error is reported depending upon the situation. This reference for the callback implements a particular interface depending on the type of data that is being retrieved.

Based on the above, the API defines the following types of classes for each category of information:

- Request class—provides both synchronous and asynchronous methods to retrieve the data.
- Response interface—caller of the asynchronous request has to implement this interface in order to register for a callback.
- Data classes—applicable for each request class where there are one or more data classes depending on the type of data that is retrieved. The data class is just an encapsulation of the data retrieved from the agent. The console component may choose to dispose the data class once the data is rendered on the console.

There are data classes that represent the Request status or Error status. The API classes and interfaces are preceded with the prefix “SM,” which distinguishes them as Sun Management Center classes.

Java Language Object Class Examples

This chapter includes examples of client API classes. Each client API class example is presented within the context of its category.

Refer to the `javadocs`, included in the Developer Environment CD, for details on the client classes, their methods and descriptions, including examples. Once the product image is installed, the most up-to-date information on where the `javadocs` reside will be available in the following HTML file:

`/opt/SUNWsymon/sdk/docs/index.html`

Note – All of the classes have the same format. They are all preceded by: `com.sun.symon.base.client.SM<`*Class or Interface Name*`>.`

The following table contains a list of the category of classes and their examples:

TABLE 14-1 Category of Classes and Examples

Category	Example
"Login API" on page 263	"Example: SMLoginTest" on page 263
"Request Status API" on page 265	"Example: SMRequestStatus" on page 265
"Raw Data API" on page 265	• "Example: SMRawDataRequest" on page 265 • "Example: getURLValue Method" on page 266 • "Example: setURLValue Method" on page 267 • "Example: createURL Method" on page 267 • "Example: getUserId Method" on page 268 • "Example: SMProbeTest" on page 269 • "Example: SMRawDataTest" on page 273 • "Example: SMRawDataAsyncTest" on page 275
"Alarm API" on page 278	• "Example: SMAlarmObjectRequest Class" on page 278 • "Example: SMAlarmAsyncTest" on page 279 • "Example: SMAlarmSyncTest" on page 282
"Managed Entity API" on page 286	"Example: SMManagedEntityTest" on page 286
"Module API" on page 292	"Example: SMModuleTest" on page 292
"Log Viewer API" on page 298	"Example: SMLogViewerTest" on page 298
"Resource Access API" on page 301	"Example: SMResourceAccessTest" on page 301
"Topology Agent API" on page 304	"Example: SMTopologyTest" on page 304
"Exception Classes API" on page 308	There are no examples for this API.

Login API

This section contains the following example:

- Example: SMLoginTest

Example: SMLoginTest

This is the first operation that the Client API programmer must perform, before using any other category of the API. The example below illustrates the usage of SMLogin class for connection establishment with the server. Subsequently, you can log in as a valid Sun Management Center user.

The user must supply the correct arguments which are server name, server port, user name and user password in order to run the program. After running this program, once the process is started, the Client API has a live connection with the server and is ready for providing other data services:

- It secures its connection between Sun Management Center console and the server.
- It uses DSA / MD5 (Digital Signature Algorithm / Message Digest) algorithms that are part of JDK to achieve this objective.)
- The publicKey variable represents the Public key in the console which corresponds to the Private key in the Sun Management Center server. The Sun Management Center console uses this public key to authenticate the server data.
- Finally, the user gets the SMRawDataRequest object handle. This handle is the live connection with the server and serves as the base class in the Sun Management Center Client API class hierarchy. The SMRawDataRequest class provides the basic data operations like reading and writing the data from the source. This can be either the Sun Management Center server or the Sun Management Center agent.
- The SMRawDataRequest object handle is then used as one of the parameters in constructing other API request classes. This is illustrated in the examples below. This is the most fundamental concept with the usage of the Client API.

CODE EXAMPLE 14-1 SMLoginTest

```
/*
 * @(#)SMLoginTest.java 1.2 99/09/15
 *
 * Copyright (c) 09/15/99 Sun Microsystems, Inc. All Rights Reserved.
*/

import com.sun.symon.base.client.*;
```

CODE EXAMPLE 14-1 SMLoginTest *(Continued)*

```
public class SMLoginTest {
    public SMLoginTest(String server_name, int server_port,
        String user, String password) throws Exception {
        try {
            // This public key is to be used as it is.
            // Please copy this key for your Login program.
            // This release of SyMON does not have provision
            // for changing the keys dynamically.

            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
                                "04cd";
            System.out.println("\n...Testing Connection \n");

            SMLogin obj = new SMLogin();
            System.out.println("Before Connection Establishment");
              obj.connect(server_name, server_port, user, password, publicKey);

            System.out.println("Successfully Connected and Authenticated");

// The following User and password should be valid for the DNS or
// the local password file on the SyMON server, depending
// on the /etc/nsswitch.conf file config. on the SyMON server host

// After authentication, the following call is used
// to obtain the SyMON server connection handle
            SMRawDataRequest req = obj.getRawDataRequest();

// The object handle "req" should be passed as one of the constructor
// arguments while instantiating any of the Client API classes.
// The handle can be used as it is, if any of the
//functionality of the SMRawDataRequest class is desired.

        }
        catch(Exception e) {
            System.out.println(e.toString());
        }
    }

    public static void main(String[] args) throws Exception {
        if ( args.length != 4 ) {
                System.out.println("usage: java" +
```

CODE EXAMPLE 14-1 SMLoginTest *(Continued)*

```
                    " SMLoginTest server_name server_port user password.");
                    System.exit(1);
            }
            else
            {
                new SMLoginTest(args[0], new Integer(args[1]).intValue(), args[2], args[3]);
                    System.exit(0);
            }
        }
}
```

Request Status API

This section includes the following example:

- Example: SMRequestStatus

Example: SMRequestStatus

For an example of this usage of SMRequestStatus Class, see the example for “Example: SMModuleTest” on page 292.

Raw Data API

This section includes the following examples:

- Example: SMRawDataRequest
- Example: SMProbeTest
- Example: SMRawDataTest
- Example: SMRawDataAsyncTest

Example: SMRawDataRequest

The SMRawDataRequest class object handle that is obtained in the login process above provides some basic type independent data operation. That is, no distinction is made whether the data is an alarm-related data or CPU data, and so forth.

This class has many overloaded methods essentially catering to various Java types. They are provided for convenience and performance-related reasons. For example, an array of objects is compared to a vector, when high performance is desired. Whereas vectors score over arrays when the amount of data being requested is dynamic.

The following are a few samples for the various functionalities for this class:

- Example: getURLValue Method
- Example: setURLValue Method
- Example: createURL Method
- Example: getUserId Method

Example: getURLValue Method

This method is a data type independent primitive to read the data for a property or a set of properties (scalar and/or vector type) from the agent or the server. The data type dependent APIs are built using this method.

TABLE 14-2 getURLValue Method

```
url = "snmp://gurudev:161/mod/solaris-standard?managedobjects";
Vector urlvect = new Vector();
urlvect.addElement(url);
data=req.getURLValue(urlvect);
for (int j = 0; j < data.size(); j++) {
row = (Vector) data.elementAt(j);
for(int i = 0; i < row.size(); i++) {
System.out.println(row.elementAt(i));
  }
}
```

The `data.size()` corresponds to the number of URLs being added to the *urlvect* variable or the number of data properties being queried. In this example, since only one URL is requested, the data.size() will return 1.

If the property is a vector or scalar, the software will return row.size().

- If a scalar property, like CPU usage, is requested, the row.size() will return 1.
- If a column in the table is queried, the row.size() will return the number of rows in that column.

Example: setURLValue Method

This method is a data type independent primitive to set the data value for a property or a set of properties (scalar and / or vector type) on the agent or the server. The data type dependent APIs are built using this method.

CODE EXAMPLE 14-2 setURLValue Method

```
String[] reqURL = new String[ 3];
   StObject[][] reqdata = new StObject[ 3][1];
for(int i = 0; i < dataURL.size(); i++) {
      String dataurl = dataURL.elementAt(i).toString();
      reqURL[i*3] = dataurl + "?historychannel";
      reqdata[i*3][0] = new StString(logURL);
      reqURL[(i*3)+1] = dataurl + "?historyinterval";
      reqdata[(i*3)+1][0] = new StString(logInterval);
      reqURL[(i*3)+2] = dataurl + "?historystatus";
      reqdata[(i*3)+2][0] = new StString("on");
   }
   handle.setURLValue(reqURL, reqdata);
```

This example illustrates how the three different attributes are set in one call. Refer to the HTML files for StObject class definition from the Sun Management Center web site.

- The getURLValue method above is a vector variant of the overloaded method.
- The setURLValue method above is an array variant of the overloaded method.

But as indicated in the above section, both the variants are available for these methods.

Example: createURL Method

Some of the samples of the above overloaded method are listed below. This method is a helper function to create a Sun Management Center URL.

Note – The Sun Management Center URL is not the same as the URL class defined in the JDK. Hence these utility method are needed. You can, however, compose their Sun Management Center URLs once you are familiar with URL structure without using these methods. The Sun Management Center agent section discusses the various types of URLs and their format. These helper methods are static methods in the SMRawDataRequest class.

CODE EXAMPLE 14-3 createURL Method

```
System.out.println("         *Complete URL*");
   try {
   url = SMRawDataRequest.createURL("gurudev", 0,
                        "modules.operatingSystem.solaris.standard" ,
                        "", "user", "primaryUser" ,"mapping", "/jiten");
   System.out.println(url);
   } catch (Exception e) {
     System.out.println("Exception:   " + e.getMessage());
   }
System.out.println("         **Testing overloaded createURL with baseURL **");
url = SMRawDataRequest.createURL(
      "snmp://gurudev:161/mod//modules.operatingSystem.solaris.standard",
      "user", "primaryUser", "mapping", "/jiten");
System.out.println(url);
System.out.println("              **Testing probe create URL**");
System.out.println("                 *Complete probe create URL*");
  url = SMRawDataRequest.createURL("gurudev", 0,
                       "modules.operatingSystem.solaris.standard", "",
                            "netstat","-a");
  System.out.println(url);
```

Example: getUserId Method

The SMRawDataRequest class object handle, as explained before, represents the user authenticated connection between the console and the server. This method allows the API to get the userId for this connection.

CODE EXAMPLE 14-4 getUserId Method

```
System.out.println("           **Testing getUserId interface**");
System.out.println("Current Login session user :    " +
req.getUserId());
```

Example: SMProbeTest

The SMProbeTest demonstrates that the console can perform an ad-hoc like query on the agent. This example assumes that the Mib2-Instrumentation module is loaded on the Sun Management Center Agent on the server host and is currently enabled. The example will either execute a '`netstat -i`' or '`ifconfig -a`', based on the selection in the example, and prints the output.

CODE EXAMPLE 14-5 SMProbeTest

```
/*
 * @(#)SMProbeTest.java 1.2 99/11/03
 *
 * Copyright (c) 11/03/99 Sun Microsystems, Inc. All Rights Reserved.
 */
import com.sun.symon.base.client.*;

import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.SocketException;

/**
 * This class gives an example of the usage of probeConnect method in
 * SMRawDataRequest class. This method returns only one TCP socket handle,
 * established with the probe application. This is to be used for read/write
 * application data. The probeConnectWithStderr is a variant of this method.
 * It returns two socket handles. One for data as above and the other for
 * errors passed from the probe application.
 */
public class SMProbeTest {

    private Socket sock;

    public SMProbeTest( String server_name, int server_port, int agent_port,
        String user, String password ) throws SMAPIException
    {
        sock = null;
        try {
            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
```

CODE EXAMPLE 14-5 SMProbeTest

```
                                "04cd";

            System.out.println("\n...Testing Connect...\n");
            SMLogin obj = new SMLogin();

            obj.connect(server_name, server_port, user, password, publicKey);
            System.out.println("Successfully Connected and Authenticated");

            SMRawDataRequest req = obj.getRawDataRequest();

            System.out.println(
               "This example assumes that the Mib2-Instrumentation module "+
                "is loaded on the Sun Management Center Agent on the server "+
              "host and is currently enabled.\n" +
              "This probe command will either execute a 'netstat -i' or "+
                "'ifconfig -a', based on the selection in this program, and "+
              "prints the output.");

            String probeURL = "snmp://" + server_name + ":" + agent_port +
                "/mod/mib2-instr/interfaces?runadhoccommand.";

            // set to true if you want 'ifconfig -a' to be executed.
            boolean useIfconfig = true;
            if (useIfconfig)
                probeURL += "ifconfig_a";
            else
                probeURL += "netstat_i";

            System.out.println("url: " + probeURL);

            sock = req.probeConnect(probeURL, null);
            System.out.println(getData());
        }
        catch (Exception e) {
                System.out.println(e.toString());
        }
        finally
        {
            closeConnection();
        }
    }

    public Object getData() throws SMAPIException {
```

CODE EXAMPLE 14-5 SMProbeTest

```
// The following timeout is added as InputStream fails to return
// -1 on end of stream. Without this timeout the sock.read blocks
//  forever even though the probe application is done, sending all of
// its response data. Alternate implementations are also possible.
// This is only an example.

        try {
            sock.setSoTimeout(2000);
        }
        catch (SocketException e) {
            throw new SMAPIException(e.getMessage());
        }

        OutputStream op;
        InputStream ip;

        try {
            op = sock.getOutputStream();
            ip = sock.getInputStream();

            // If your application requires data to be written,
            // use op.write() and op.flush().
        }
        catch(Exception e) {
            throw new SMAPIException(e.getMessage());
        }

        StringBuffer tmp = new StringBuffer();
        byte buff[] = new byte[4096];
        int len = -1;

        while (true) {
            try {
                len = ip.read(buff);
            }
            catch ( Exception e ) {
                break;
            }

            if (len == -1)
                    break;
            tmp.append(new String(buff, 0, len));
```

CODE EXAMPLE 14-5 SMProbeTest

```
        }

        if (tmp.length() == 0) // no data
            return null;

        return tmp;
    }

    public void closeConnection() throws SMAPIException {
        try {
            if (sock != null) {
                sock.close();
                sock = null;
            }
        }
        catch(Exception e) {
            throw new SMAPIException(e.getMessage());
        }
    }

    private static void usage()
    {
        System.out.println(
            "usage: java SMProbeTest " +
            "server_name server_port agent_port user password");
    }

    public static void main(String[] args) throws Exception {
        if ( args.length != 5) {
            usage();
            System.exit(1);
        }
        else {
            new SMProbeTest (args[0], new Integer(args[1]).intValue(),
                new Integer(args[2]).intValue(), args[3], args[4]);
            System.exit(0);
        }
    }
}
```

Example: SMRawDataTest

The SMRawDataTest demonstrates the usage of methods from the class SMRawDataRequest. At the later part of the example, it will retrieve the CPU Idle time and the CPU User time from the kernel reader module.

CODE EXAMPLE 14-6 SMRawDataTest

```
/*
 * @(#)SMRawDataTest.java       1.1 99/09/13
 *
 * Copyright (c) 09/13/99 Sun Microsystems, Inc. All Rights Reserved.
 */

import com.sun.symon.base.client.*;
import java.util.Vector;

public class SMRawDataTest {

    public SMRawDataTest( String server_name, int server_port, int agent_port,
        String user, String password ) throws Exception
    {
        try {
            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
                                "04cd";

            System.out.println("*** Testing Connect ****");
            SMLogin obj = new SMLogin();

            obj.connect(server_name, server_port, user, password, publicKey);
            System.out.println("Successfully Connected and Authenticated ");

            SMRawDataRequest req = obj.getRawDataRequest();

            System.out.println("*** Testing the SMRawDataRequest ***");

            // This method is used by the client application to know
            // location of the topology server
            System.out.println("** Testing getTopologyBaseURL**");
            System.out.println("Topology Base URL = " +
                req.getTopologyBaseURL());

            // This method is used by the client application to know
```

CODE EXAMPLE 14-6 SMRawDataTest *(Continued)*

```
        // location of the Event manager server
        System.out.println("** Testing getEventBaseURL**");
       System.out.println("Event Base URL = " + req.getEventBaseURL());

        // This method is used by the client application to know
        // position of the Configuration manager server
        System.out.println("** Testing getConfigurationBaseURL**");

        System.out.println("Config Base URL = " +
            req.getConfigurationBaseURL());

        // This method enables the Client API
        // to get the userid of the current server session
        System.out.println("**Testing getUserId interface**");
        System.out.println("Current Login session user :    " +
            req.getUserId());

        System.out.println("**Testing getURLValue **");
        String[] urlarr = new String[2];

        urlarr[0] = "snmp://" + server_name + ":" + agent_port +
            "/mod/kernel-reader/cpu-detail/cpu-util/cpuUtilTable/" +
            "cpuUtilEntry/cpu_idle";

        urlarr[1] = "snmp://" + server_name + ":" + agent_port +
            "/mod/kernel-reader/cpu-detail/cpu-util/cpuUtilTable/" +
            "cpuUtilEntry/cpu_user";

        Vector urlvect = new Vector();
        urlvect.addElement(urlarr[0]);
        urlvect.addElement(urlarr[1]);

        Vector dat = req.getURLValue(urlvect);

        if (dat.size() != 2)
            throw new SMAPIException("Incorrect data returned");

        for (int i = 0; i < dat.size(); i++) {
            Vector row = (Vector) dat.elementAt(i);
            for (int j = 0; j < row.size() ; j++)
            {
                if (i == 0)
                    System.out.println("% CPU Idle time for cpu(s): "
                        + row.elementAt(j));
                else
                    System.out.println("% CPU User time for cpu(s): "
                        + row.elementAt(j));
```

CODE EXAMPLE 14-6 SMRawDataTest *(Continued)*

```
                }
            }
        }
        catch ( Exception e) {
            System.out.println(e.getMessage());
        }
    }

    private static void usage()
    {
        System.out.println(
            "usage: java SMRawDataTest " +
            "server_name server_port agent_port user password");
    }

    public static void main(String[] args) throws Exception {
        if ( args.length != 5) {
            usage();
            System.exit(1);
        }
        else {
            new SMRawDataTest (args[0], new Integer(args[1]).intValue(),
                new Integer(args[2]).intValue(), args[3], args[4]);
            System.exit(0);
        }
    }
}
```

Example: SMRawDataAsyncTest

The SMRawDataAsyncTest is basically doing the same test as SMRawDataTest. This example makes use of an asynchrous request in a periodic cycle of 60 seconds. Every periodic time-out will cause the server to call back and the getURLresponse method will be invoked.

CODE EXAMPLE 14-7 SMRawDataAsyncTest

```
/*
 * @(#)SMRawDataAsyncTest.java  1.1 99/09/13
 *
 * Copyright (c) 09/13/99 Sun Microsystems, Inc. All Rights Reserved.
 */

import com.sun.symon.base.client.*;
```

CODE EXAMPLE 14-7 SMRawDataAsyncTest *(Continued)*

```
import java.util.Vector;

public class SMRawDataAsyncTest extends SMRawDataResponseAdapter {

    public SMRawDataAsyncTest( String server_name, int server_port,
        int agent_port, String user, String password ) throws Exception
    {
        try {
            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
                                "04cd";

            System.out.println("*** Testing Connect ****");
            SMLogin obj = new SMLogin();

            obj.connect(server_name, server_port, user, password, publicKey);
            System.out.println("Successfully Connected and Authenticated");

            SMRawDataRequest req = obj.getRawDataRequest();

           System.out.println("***Testing the Async. SMRawDataRequest ***");
            System.out.println("**Testing getURLValue **");

            // This is an example of how an async. cyclic URL request can be
            // placed. This request will cause a server callback every 60
            // seconds for the following URLs. If period is null then there
            // will be only a one time callback with the values for the URL.

            String[] urlarr = new String[2];

            urlarr[0] = "snmp://" + server_name + ":" + agent_port +
                "/mod/kernel-reader/cpu-detail/cpu-util/cpuUtilTable" +
                "/cpuUtilEntry/cpu_idle";

            urlarr[1] = "snmp://" + server_name + ":" + agent_port +
                "/mod/kernel-reader/cpu-detail/cpu-util/cpuUtilTable/" +
                "cpuUtilEntry/cpu_user";

            Vector urlvect = new Vector();
            urlvect.addElement(urlarr[0]);
            urlvect.addElement(urlarr[1]);

            req.getURLValue(urlvect, "60", this, this);
```

CODE EXAMPLE 14-7 SMRawDataAsyncTest *(Continued)*

```
            System.out.println("sleeping...");
            Thread.sleep(200 * 1000);
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public void getURLResponse( SMRequestStatus status, Vector dat,
        Object identifier)
    {
        int error = status.getReturnCode();
        if (error == SMErrorCode.SUCCESS) {
            if (dat.size() != 2)
            {
                System.out.println("Incorrect data returned. size =" +
                    dat.size());
            }
            else {
                for (int i = 0; i < dat.size(); i++) {
                    Vector row = (Vector) dat.elementAt(i);
                    for (int j = 0; j < row.size() ; j++)
                    {
                        if (i == 0)
                            System.out.println("% CPU Idle time for cpu(s): "
                                + row.elementAt(j));
                        else
                            System.out.println("% CPU User time for cpu(s): "
                                + row.elementAt(j));
                    }
                }
            }
        }
        else {  // Failure
            // The various error codes as in SMErrorCode may be reported here
            System.out.println("Error code = " + error + "   " +
                "Msg Text   = " + status.getMessageText() +
                " "  + "Exception  = " + (status.getException()).getMessage());
        }
    }

    private static void usage()
    {
        System.out.println(
            "usage: java SMRawDataAsyncTest " +
            "server_name server_port agent_port user password");
```

CODE EXAMPLE 14-7 SMRawDataAsyncTest *(Continued)*

```
    }

    public static void main(String[] args) throws Exception {
        if ( args.length != 5) {
            usage();
            System.exit(1);
        }
        else {
            new SMRawDataAsyncTest (args[0], new Integer(args[1]).intValue(),
                new Integer(args[2]).intValue(), args[3], args[4]);
            System.exit(0);
        }
    }
}
```

Alarm API

This section includes the following example:

- Example: SMAlarmObjectRequest Class

Example: SMAlarmObjectRequest Class

The following are examples related to the SMAlarmObjectRequest class:

- Example: SMAlarmAsyncTest
- Example: SMAlarmSyncTest

The following examples illustrate both the synchronous and asynchronous usage of the API.

Example: SMAlarmAsyncTest

The SMAlarmAsyncTest demonstrates the usage of the `client.alarm` API to query the alarm information from the server. Notice the example is doing the request asynchronously. The program will go into a sleep mode for a certain period of time and then wake up to response the callback from the server.

CODE EXAMPLE 14-8 SMAlarmAsyncTest

```
/*
 * @(#)SMAlarmAsyncTest.java    1.1 99/09/13
 *
 * Copyright (c) 09/13/99 Sun Microsystems, Inc. All Rights Reserved.
 */

import com.sun.symon.base.client.*;
import com.sun.symon.base.client.alarm.*;

import java.util.Vector;

/**
* This class is for testing the Client API.
**/
public class SMAlarmAsyncTest implements SMAlarmObjectResponse {

    private SMAlarmObjectRequest alreq = null;

    public SMAlarmAsyncTest( String server_name, int server_port, int agent_port,
        String user, String password ) throws Exception {

        try {
            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
                                "04cd";

            System.out.println("*** Testing Connection ****");
            SMLogin obj = new SMLogin();
            System.out.println("Before Connection Establishment");
            obj.connect(server_name, server_port, user, password, publicKey);
            System.out.println("Successfully Connected and Authenticated");

            SMRawDataRequest req = obj.getRawDataRequest();

           System.out.println("   ***Testing SMAlarmObjectRequest class ***");
```

CODE EXAMPLE 14-8 SMAlarmAsyncTest *(Continued)*

```
            alreq = new SMAlarmObjectRequest(req, null);
            Vector allist = null;

            // This commented section can be used if
            // Alarm retrieval is desired for a domain.

            System.out.println("** Testing getAlarms on topology**");
           alreq.getAlarms("1", "Default Domain", "{ERR} {WRN}", "{O} {C} {F}",
                "{A} {N}", null, null, null, this, new Object());

           System.out.println("** Initiating a fresh request to get alarms**");

            // get alarms on host
            alreq.getAlarms("2", server_name, null , "{ERR} {WRN}",
                "{O} {C} {F}", null, null,
                null, null, this, new Object());

            System.out.println("sleeping...");
            Thread.sleep(60 * 1000);
        }
        catch ( Exception e) {
            System.out.println(e.getMessage());
        }
    }

    public void getAlarmResponse(SMRequestStatus status, Vector data,
        Object identifier, SMAlarmIteratorAsync iter) {

        if (data == null)
            return;

        SMAlarmObjectData aldata=null;

        System.out.println("Callback:size " + data.size());

        for(int i = 0; i < data.size(); i++) {
            aldata = (SMAlarmObjectData)data.elementAt(i);
            String host =  aldata.getHost();
            System.out.println("Alarm Id: "+ aldata.getAlarmId());
            System.out.println("Rule Id: "+ aldata.getAlarmRuleId());
           System.out.println("Open Time stamp: "+ aldata.getOpenTimestamp());
            System.out.print("Alarm text: " + aldata.getAlarmShortText());
            System.out.println("Alarm Long Key : "+ aldata.getAlarmLongKey());
            System.out.println("Node URL: "+ aldata.getMoURL());
            System.out.println("Target host: " + aldata.getHost());
            System.out.println("Alarm State: " + aldata.getAlarmState());
            System.out.println("Alarm Severity: " + aldata.getSeverity());
```

CODE EXAMPLE 14-8 SMAlarmAsyncTest *(Continued)*

```
            System.out.println("Upd Time stamp: " +
                aldata.getUpdateTimestamp());
            System.out.println("Upd Reason: "+ aldata.getUpdateReason());
            System.out.println("Machine Type: "+ aldata.getMachineType());
            System.out.println("Rule Group: "+ aldata.getRuleGroup());
            System.out.println("Clo Time stamp: " +
                aldata.getCloseTimestamp());
            System.out.println("Ack Time stamp: " + aldata.getAckTimestamp());
            System.out.println("Ack operator: " + aldata.getAckOperator());
            System.out.println("Fix Time stamp: " + aldata.getFixTimestamp());
            System.out.println("Fix operator: " + aldata.getFixOperator());
        }

        try {
            if (data.size() > 0)
                iter.getNextAlarms();
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }

    public void setAlarmResponse(SMRequestStatus status, Object identifier) {}

    private static void usage()
    {
        System.out.println("usage: java SMAlarmAsyncTest " +
            "server_name server_port agent_port name password");
    }

    public static void main(String[] args) throws Exception {
        if ( args.length != 5) {
            usage();
            System.exit(1);
        }
        else {
            new SMAlarmAsyncTest (args[0], new Integer(args[1]).intValue(),
                new Integer(args[2]).intValue(), args[3], args[4]);
            System.exit(0);
        }
    }
}
```

Example: SMAlarmSyncTest

SMAlarmSyncTest basically performs the same test as SMAlarmAsyncTest except that the program does not go into any sleep mode.

CODE EXAMPLE 14-9 SMAlarmSyncTest

```
/*
 * @(#)SMAlarmSyncTest.java     1.1 99/09/13
 *
 * Copyright (c) 09/13/99 Sun Microsystems, Inc. All Rights Reserved.
 */

import com.sun.symon.base.client.*;
import com.sun.symon.base.client.alarm.*;

import java.util.Vector;

/**
 * This class is for testing the Client API.
 **/
public class SMAlarmSyncTest {

private SMAlarmObjectRequest alreq = null;

   public SMAlarmSyncTest( String server_name, int server_port, int agent_port,
        String user, String password ) throws Exception {
        try {
            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
                                "04cd";

            System.out.println("*** Testing Connection ****");
            SMLogin obj = new SMLogin();
            System.out.println("Before Connection Establishment");
            obj.connect(server_name, server_port, user, password, publicKey);
            System.out.println("Successfully Connected and Authenticated");

            SMRawDataRequest req = obj.getRawDataRequest();

           System.out.println("   ***Testing SMAlarmObjectRequest class ***");
            alreq = new SMAlarmObjectRequest(req, null);

         System.out.println("       ** Testing Sync. getAlarms with moURL **");
```

CODE EXAMPLE 14-9 SMAlarmSyncTest *(Continued)*

```
SMAlarmIteratorSync iter = alreq.getAlarms("1", null, null, "{ERR} {WRN}",
            "{O} {C} {F}", null, null, null, null);
        Vector allist = iter.getData();

        SMAlarmObjectData aldata=null;

        while (allist.size() > 0) {
            System.out.println(allist.size());
            for(int i = 0; i < allist.size(); i++) {
                aldata = (SMAlarmObjectData)allist.elementAt(i);
                System.out.println("Alarm Id       :   " +
                    aldata.getAlarmId());
                System.out.println("Rule Id        :   " +
                    aldata.getAlarmRuleId());
                System.out.println("Open Time stamp:   " +
                    aldata.getOpenTimestamp());
                System.out.println("Alarm text     :   " +
                    aldata.getAlarmShortText());
                System.out.println("Alarm Long Key :   " +
                    aldata.getAlarmLongKey());
                System.out.println("Node URL       :   " +
                    aldata.getMoURL());
                System.out.println("Target host    :   " +
                    aldata.getHost());
                System.out.println("Alarm State    :   " +
                    aldata.getAlarmState());
                System.out.println("Alarm Severity :   " +
                    aldata.getSeverity());
                System.out.println("Upd Time stamp :   " +
                    aldata.getUpdateTimestamp());
                System.out.println("Upd Reason     :   " +
                    aldata.getUpdateReason());
                System.out.println("Machine Type   :   " +
                    aldata.getMachineType());
                System.out.println("Rule Group     :   " +
                    aldata.getRuleGroup());
                System.out.println("Clo Time stamp :   " +
                    aldata.getCloseTimestamp());
                System.out.println("Ack Time stamp :   " +
                    aldata.getAckTimestamp());
                System.out.println("Ack operator   :   " +
                    aldata.getAckOperator());
                System.out.println("Fix Time stamp :   " +
                    aldata.getFixTimestamp());
                System.out.println("Fix operator   :   " +
                    aldata.getFixOperator());
```

CODE EXAMPLE 14-9 SMAlarmSyncTest *(Continued)*

```
                Vector ret = null;
                String[] id = new String[1];

                // set the following to true if you actually want
                // to modify the alarms.
                boolean modifyAlarms = false;

                if (modifyAlarms)
                {
                    // Note the following test code for
                    // ack/fix/delete is just an
                    // example of its usage
                    if (!aldata.isFixed()) {
                        System.out.println("** Testing ack alarms **");

                        if (aldata != null) {
                            id[0] = aldata.getAlarmId();
                            ret = alreq.ackAlarms(id, "just for" +
                               " the heck of it");
                        }
                        else {
                            System.out.println("No alarm to be acked");
                        }

                        if (ret == null) {
                            System.out.println("Ack Success");
                        }
                        else {
                            System.out.println("Ack Failed");
                        }

                        if (aldata.isOpen())  {
                           System.out.println("** Testing fix alarms **");
                            ret = null;

                            if (aldata != null) {
                                id[0] = aldata.getAlarmId();
                                ret = alreq.fixAlarms(id,
                                    "just for the heck of it");
                            }
                            else {
                               System.out.println("No alarm to be fixed");
                            }

                            if (ret == null) {
                                System.out.println("Fix Success");
                            }
```

CODE EXAMPLE 14-9 SMAlarmSyncTest *(Continued)*

```
                                else {
                                    System.out.println("Fix Failed");
                                }
                            }
                        }

                        System.out.println("** Testing delete alarms **");

                        ret= null;
                        if (aldata != null) {
                            id[0] = aldata.getAlarmId();
                            ret = alreq.deleteAlarms(id,
                                "just for the heck of it");
                        }
                        else {
                            System.out.println("No alarm to be deleted");
                        }

                        if (ret == null) {
                            System.out.println("Success");
                        }
                        else {
                            System.out.println("Failed");
                        }
                    }
                }
                System.out.println("** Requesting the next batch**");

                iter = iter.getNextAlarms();
                allist = iter.getData();
            }
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    private static void usage()
    {
        System.out.println("usage: java SMAlarmSyncTest " +
            "server_name server_port agent_port user password");
    }

    public static void main(String[] args) throws Exception {
        if ( args.length != 5) {
            usage();
            System.exit(1);
```

CODE EXAMPLE 14-9 SMAlarmSyncTest *(Continued)*

```
        }
        else {
            new SMAlarmSyncTest (args[0], new Integer(args[1]).intValue(),
                new Integer(args[2]).intValue(), args[3], args[4]);
            System.exit(0);
        }
    }
}
```

Managed Entity API

This section includes the following example:

- Example: SMManagedEntityRequest Class

Example: SMManagedEntityTest

This is a class that uses the services of the SMRawDataRequest class in order to get data type dependent information.

This class is used to programmatically query the schema of the properties defined in the agent. This section provides the organization of the schema on the agent.

The agent contains one or more modules. A module is made up of one or more managed objects. Each managed object contains one or more properties. And each property has qualifiers. For a detailed explanation of these terms, refer to the Sun Management Center architecture document and the agent section.

The examples in this section include a few examples of how this class is used for the purpose of obtaining the values for the above entities.

The constructor of this class takes the SMRawDataRequest object handle, which encapsulates the authenticated RMI connection handle with the server, in order to be capable of sending requests over the RMI to the server. This concept is used for every other API request class instantiation in Sun Management Center.

CODE EXAMPLE 14-10 SMManagedEntityTest

```
/*
 * @(#)SMManagedEntityTest.java 1.2 99/10/15
 *
```

CODE EXAMPLE 14-10 SMManagedEntityTest *(Continued)*

```
 * Copyright (c) 10/15/99 Sun Microsystems, Inc. All Rights Reserved.
 */

import com.sun.symon.base.client.*;
import com.sun.symon.base.client.attribute.*;
import java.util.Vector;

/**
* This class is for testing the Client API.
**/
public class SMManagedEntityTest {
    public SMManagedEntityTest( String server_name, int server_port,
        int agent_port, String user, String password ) throws Exception {
        try {
            /*Checkout SMLogin.java for Login/Authentication Test */
            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
                                "04cd";

            System.out.println("*** Testing Connection ****");
            SMLogin obj = new SMLogin();

            obj.connect(server_name, server_port, user, password, publicKey);
            System.out.println("Successfully Connected and Authenticated.");

            SMRawDataRequest req = obj.getRawDataRequest();

             // This file lists the usage of the various methods supported by
            // the SMManagedEntityRequest class.
            // The basic usage of this class is to request
            // the schema of the MIB supported on the agent.

            System.out.println("**Testing ManagedEntityRequest  **");
            SMManagedEntityRequest mreq = new SMManagedEntityRequest(req);

            System.out.println("**Testing getManagedObjectList method**");
            // The following  method is used to list all the Managed
            // objects associated with a module.

              // The following call lists all the managed objects instrumented
            // on the "kernel-reader" module.

            String[] urlarr = new String[1];
```

CODE EXAMPLE 14-10 SMManagedEntityTest *(Continued)*

```
            urlarr[0] = "snmp://" + server_name + ":" + agent_port +
                "/mod/kernel-reader";

            Vector molist = mreq.getManagedObjectList(urlarr[0]);

            for (int i = 0; i < molist.size(); i++) {
                System.out.println(molist.elementAt(i).toString());
            }

            // The SyMON agent supports managed object properties
            // which are internal to the agent/console
            // usage and should not be exposed to the SyMON console
            // browser in the form of tables etc.
            // The following call lists the visible properties
            // (can be displayed in the SyMON browser),
            // for a managed object based on a module.
            // Eg. The call below lists the visible properties for a
            // Managed object "filesystem" on a "kernel-reader" module.

// ---------------------------------------------------------------------
            System.out.println("**Testing getVisiblePropertyList method**");
            urlarr[0] = "snmp://" + server_name + ":" + agent_port +
                "/mod/kernel-reader";
           Vector plist =mreq.getVisiblePropertyList(urlarr[0], "filesystem");

            System.out.println("The visible property data for URL " +
                urlarr[0] + "/filesystem");

            for (int i = 0; i < plist.size(); i++) {
                System.out.println("Visible Property Name : " +
                    plist.elementAt(i).toString());
            }

// ---------------------------------------------------------------------
            System.out.println(
                "**Testing getVisiblePropertyDataList method**");
          // This method is similar to the method above,
          // but in addition to the name of the property, it also returns
          // the type of the property eg. Vector , Scalar etc.

            urlarr[0] = "snmp://" + server_name + ":" +
                agent_port + "/mod/kernel-reader";
            Vector pdlist = mreq.getVisiblePropertyDataList(urlarr[0],
                "filesystem");

            System.out.println("The visible property data for URL " +
                urlarr[0] + "/filesystem");
```

CODE EXAMPLE 14-10 SMManagedEntityTest *(Continued)*

```
            for (int i = 0; i < pdlist.size(); i++) {
                SMPropertyData pd  = (SMPropertyData)pdlist.elementAt(i);

                System.out.println("Visible Property Name : " +
                    pd.getPropertyName() + "  " +
                    "Property Type : " +
                    ((pd.getPropertyType()) ? "scalar" : "vector"));
            }

            // The following method is used to list all the
            // properties supported by a managed object on a
            // particular module. This includes the visible/hidden
            // properties supported by the managed object.

// ---------------------------------------------------------------------------
            System.out.println("**Testing getPropertyList method**");

            urlarr[0] = "snmp://" + server_name + ":" + agent_port +
                "/mod/kernel-reader";

            Vector vplist = mreq.getPropertyList(urlarr[0], "filesystem");

            System.out.println("The property data for URL "+
                urlarr[0]+ "/filesystem");

            for (int i = 0; i < vplist.size(); i++) {
                System.out.println("Property Name : " +
                    vplist.elementAt(i).toString());
            }

            // This method is similar to the above,
            // but in addition lists the type of the property, as a
            // Vector or a scalar.

// ---------------------------------------------------------------------------
            System.out.println("**Testing getPropertyDataList method**");
            urlarr[0] = "snmp://" + server_name + ":" +
                agent_port + "/mod/kernel-reader";

            Vector vpdlist =mreq.getPropertyDataList(urlarr[0], "filesystem");

            System.out.println("The property data for URL " +
                urlarr[0] + "/filesystem");

            for (int i = 0; i < vpdlist.size(); i++) {
                SMPropertyData vpd  = (SMPropertyData)vpdlist.elementAt(i);
```

CODE EXAMPLE 14-10 SMManagedEntityTest *(Continued)*

```
                System.out.println("Property Name : " +
                    vpd.getPropertyName() + "  " +
                    "Property Type : " +
                    ((vpd.getPropertyType()) ? "scalar" : "vector"));
            }

            // This method is used to list all
            // the tables associated with a managed object.

// ---------------------------------------------------------------------
            System.out.println("**Testing getTableList method**");

            urlarr[0] = "snmp://" + server_name + ":" +
                agent_port + "/mod/kernel-reader";

            Vector tlist = mreq.getTableList(urlarr[0], "filesystem");

           System.out.println("The tables for URL "+ urlarr[0]+"/filesystem");

            for (int i = 0; i < tlist.size(); i++) {
                System.out.println("Table Name : " +
                    tlist.elementAt(i).toString());
            }

// ---------------------------------------------------------------------
            System.out.println("**Testing getTableSchema method**");

            for (int i = 0; i < tlist.size(); i++) {
                Vector ts = mreq.getTableSchema(urlarr[0], "filesystem",
                    tlist.elementAt(i).toString());
                System.out.println("Table Schema for filesystem");

                for( int x = 0 ; x < ts.size() ; x++) {
                    System.out.println("Column("+x+")   = " +
                        ts.elementAt(x).toString());
                }
            }
// ---------------------------------------------------------------------
            // Read the contents of the table.

            System.out.println("**Testing getTableValue method**");

            for (int i = 0; i < tlist.size(); i++) {
                Vector v = mreq.getTableValue(urlarr[0], "filesystem",
                    tlist.elementAt(i).toString());
```

CODE EXAMPLE 14-10 SMManagedEntityTest *(Continued)*

```
                for(int j = 0; j < v.size() ; j++ ) {
                    Vector rowd = (Vector)v.elementAt(j);
                    System.out.println(rowd.size());
                    System.out.print("Row Data   : ");
                    for(int k = 0; k < rowd.size() ; k++ ) {
                        System.out.print(rowd.elementAt(k).toString()+ "   ");
                        System.out.print("");
                    }

                }
            }

            // For SyMON agents every Managed object property
            // has a list of attributes called Qualifiers.
            // The following method lists the qualifiers for
            // a managed object property.
// ---------------------------------------------------------------------
            System.out.println("**Testing getQualifierList method**");
            urlarr[0] = "snmp://" + server_name + ":" +
                agent_port + "/mod/kernel-reader";

            Vector qlist = mreq.getQualifierList(urlarr[0], "user",
                "primaryUser");

            System.out.println("The qualifier data for URL " +
                urlarr[0]+"/user/primaryUser");
            for (int i = 0; i < qlist.size(); i++) {
                System.out.println("Qualifier Name : " +
                    qlist.elementAt(i).toString());
            }
        }
        catch ( Exception e) {
            e.printStackTrace();
            System.out.println(e.getMessage());
        }
    }

    private static void usage()
    {
        System.out.println(
            "usage: java SMManagedEntityTest " +
            "server_name server_port agent_port user password");
    }

    public static void main(String[] args) throws Exception {
        if ( args.length != 5) {
            usage();
```

CODE EXAMPLE 14-10 SMManagedEntityTest *(Continued)*

```
            System.exit(1);
        }
        else {
            new SMManagedEntityTest(args[0], new Integer(args[1]).intValue(),
                new Integer(args[2]).intValue(), args[3], args[4]);
            System.exit(0);
        }
    }
}
```

Module API

This section includes information on the following:

- Example: SMModuleTest

Example: SMModuleTest

This example also has the usage for SMRequestStatus class, which is used for reporting errors in asynchronous methods.

The SMModuleTest demonstrates the dumping of modules information from the agents. In this example, the program will list modules and loaded modules, get the Health Monitor module's information as specified in the corresponding configuration file. It will also load, disable, enable, and unload the Health Monitor module. Finally, it demonstrates getting the module information in asynchronous mode.

CODE EXAMPLE 14-11 SMModuleTest

```
/*
 * @(#)SMModuleTest.java        1.6 99/10/15
 *
 * Copyright (c) 10/15/99 Sun Microsystems, Inc. All Rights Reserved.
 */

import com.sun.symon.base.client.*;
import com.sun.symon.base.client.module.*;

import java.util.StringTokenizer;
import java.util.Vector;
```

CODE EXAMPLE 14-11 SMModuleTest *(Continued)*

```
public class SMModuleTest implements SMModuleResponse {

    public SMModuleTest( String server_name, int server_port, int agent_port,
        String user, String password )  throws Exception
    {
        try {
            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
                                "04cd";

            // Connection establish
            System.out.println("\n...Testing Connection ...\n");
            SMLogin obj = new SMLogin();

            System.out.println("Before Connection Establishment");
            obj.connect(server_name, server_port, user, password, publicKey);

            System.out.println("Successfully Connected and Authenticated");
            SMRawDataRequest req = obj.getRawDataRequest();

            // SMModuleRequest
            System.out.println("\n... Testing the SMModuleRequest ...\n");
            SMModuleRequest modreq = new SMModuleRequest(req);

            System.out.println("\n..... List All the Modules ......\n");
            Vector allmodlist = modreq.listModules(server_name, agent_port);
            for(int i = 0; i < allmodlist.size(); i++){
                System.out.println(allmodlist.elementAt(i));
            }

            // List Loaded Modules
            System.out.println("\n..... List Loaded Modules ......\n");
           Vector loadmodlist = modreq.listLoadedModules(server_name, agent_port);
            for(int i = 0; i < loadmodlist.size(); i++){
                System.out.println(loadmodlist.elementAt(i));
            }

            // Get Loaded Module Info
            System.out.println("\n......Get Loaded Module Info......\n");
            String[][] loadmodinfo = modreq.getLoadedModuleInfo(server_name,
                agent_port);
            for(int i = 0; i < loadmodinfo.length ; i++) {
```

CODE EXAMPLE 14-11 SMModuleTest *(Continued)*

```
                // This will display the localized description for the module
                System.out.println("Module Name = " + loadmodinfo[i][0]);
                System.out.println("Module URL = " + loadmodinfo[i][1]);
            }

            // Use the health monitor module
            String module = "health-monitor";
            String modinst = null;
            String moduleName = "Health Monitor";
            String modUrl = "snmp://" + server_name + ":" +
                agent_port + "/mod/" + module;

            System.out.println("\n....... Get the " + moduleName +
                "Module's loading info. as specified in X-file ......\n");

            String moduleX = modreq.getModuleXfile(server_name,
                agent_port, module);

            System.out.println("Module X-file info = " + moduleX);

            System.out.println("\n......Load Health Monitor Module......\n");

            String modparam = getModuleParamFromX(moduleX);
            System.out.println(modparam);

            modparam = "moduleName = \"Health Monitor\"; version = \"2.0\";" +
              " console = \"health-monitor\"; enterprise = \"sun\"; " +
              " i18nModuleName = \"base.modules.health-monitor:moduleName\";"+
              " i18nModuleType = \"base.modules.health-monitor:moduleType\";"+
              " i18nModuleDesc = \"base.modules.health-monitor:moduleDesc\";"+
              " location =
\".iso.org.dod.internet.private.enterprises.sun.prod.sunsymon.agent.modules.healthMonitor\";";


            if (!modreq.loadModule(server_name, agent_port, module,
                        modinst, modparam))
                System.out.println(moduleName +
                    " module is already loaded");
            else
                System.out.println(moduleName +
                    " module is successfully loaded");

            System.out.println("\n......Check Module Loaded/Unloaded.....\n");
            if ( modreq.isModuleLoaded(server_name, agent_port, module) )
                System.out.println(moduleName+ " is loaded !");
             else
                System.out.println(moduleName+ " is unloaded !");
```

CODE EXAMPLE 14-11 SMModuleTest *(Continued)*

```
System.out.println("\n......Get Module Data.....\n");
Vector modV = modreq.getModuleData(server_name,
            agent_port, module);
SMModuleData modData;
if ( modV == null ) {
    System.out.println("no module data is available");
} else {
  for ( int i=0; i<modV.size(); i++ ) {
    modData = (SMModuleData)modV.elementAt(i);
    System.out.println(modData.getModule()+", "+
            modData.getModuleName()+", "+
            modData.getModuleInstance()+", "+
            modData.getModuleLocation());
  }
}

// Get Module Parameters
System.out.println("\n **Get Module Parameters** \n");
String mParams = modreq.getModuleParams(server_name,
            agent_port, module, modinst);
System.out.println(mParams);

// Disable Module
System.out.println("**Disable Module **");

if (modreq.disableModule(server_name,agent_port,module,modinst)) {
    System.out.println(moduleName +
        " module is successfully disabled");
} else {
    System.out.println(moduleName +
        " module is already disabled");
}

// Enable Module
System.out.println("**Enable Module **");
if (modreq.enableModule(server_name,agent_port,module,modinst)) {
    System.out.println(moduleName +
        " module is successfully enabled");
} else {
    System.out.println(moduleName +
        " module is already enabled");
}

System.out.println("**Unload Module **");
modreq.unloadModule(server_name, agent_port, module, modinst);
```

CODE EXAMPLE 14-11 SMModuleTest *(Continued)*

```
            System.out.println(moduleName +
                " module is successfully unloaded");

            // Get info of all modules
            System.out.println("\n..... " +
                "Get module loading/select info. async. ......\n");

            // The following method submits the callback
            // interface implementation to receive the info.
            // This interface is also implemented by this class

            modreq.getModuleInfoRequest(server_name, agent_port,
                "20", this);

            System.out.println("Sleeping...");
            Thread.sleep(60 *1000);
        }
        catch ( Exception e) {
            System.out.println(e.toString());
            e.printStackTrace();
        }
    }

    public void getModuleInfoResponse(SMRequestStatus status,
           SMModuleInfo[] data){

        // The callback may also be reporting an error condition,
        // hence this check is essential.

        int error = status.getReturnCode();
        if (error == SMErrorCode.SUCCESS) {    /*Success */
            System.out.println("\n**************");
            for(int i = 0; i < data.length; i++) {   // Number of modules
               System.out.println("Module Name = " + data[i].getModuleName() +
                " Module Id = " + data[i].getModuleId() +
                " Module inst. Loaded = " + data[i].getCurrentLoadCount() +
                " Can more inst. be loaded ? = " + data[i].canLoadAnother());
            }
        }
        else {                                  /* Failure */
            //The various error codes as in SMErrorCode may be reported here */
            System.out.println("Error code = " + error + "   " +
                "Msg Text   = " + status.getMessageText() +
                "    "  +"Exception  = " +
                (status.getException()).getMessage());

        }
```

CODE EXAMPLE 14-11 SMModuleTest *(Continued)*

```
    }

    /* The following methods have empty implementations.*/
    public void getLoadedModulesResponse(SMRequestStatus status,
        Vector data, Object identifier) {}

    /**
     * Gets the module parameters (suitable for use with loadModule)
     * from the return value of getModuleXfile() as input.
     * Strip out all param:name = value and concatenate them. For example:
     * <pre>
     * param:module = health-monitor
     * param:moduleName = Health Monitor
     *
     * will return: module = health-monitor; moduleName = Health Monitor;
     **/

    private String getModuleParamFromX(String x)
    {
        StringTokenizer tok = new StringTokenizer(x, "\n");
        StringBuffer s = new StringBuffer();
        String t;

        while (tok.hasMoreTokens())
        {
            t = tok.nextToken();
            if (t.startsWith("param:"))
            {
                s.append(t.substring(6)); // skip over "param:"
                s.append("; ");
            }
        }

        return s.toString();
    }

    private static void usage()
    {
        System.out.println("usage: java SMModuleTest" +
            " server_name server_port agent_port user password");
    }

    public static void main(String[] args) throws Exception {
        // args[0] is server_name
        // args[1] is server_port
        // args[2] is agent_port
        // args[3] is user
```

CODE EXAMPLE 14-11 SMModuleTest *(Continued)*

```
        // args[4] is password

        if (args.length != 5)
            usage();
        else
            new SMModuleTest( args[0], new Integer(args[1]).intValue(),
                new Integer(args[2]).intValue(), args[3], args[4] );

        System.exit(0);
    }
}
```

Log Viewer API

This section includes information on the following:

- Example: SMLogViewerTest

Example: SMLogViewerTest

The SMLogViewerTest dumps the syslog from the specified URL in the program and then it will go into sleep mode. During the sleeping time frame, if there is a wrong password supplied to the 'su', the method logSearchResponse will be invoked from the server callback.

CODE EXAMPLE 14-12 SMLogViewerTest

```
/*
 * @(#)SMLogViewerTest.java     1.2 99/09/15
 *
 * Copyright (c) 09/15/99 Sun Microsystems, Inc. All Rights Reserved.
 */

import com.sun.symon.base.client.*;
import com.sun.symon.base.client.log.*;

/*
 The following class implements the SMLogViewerResponse interface to
 receive a callback from the SMLogViewerrequest.logSearch async. request
 method.
*/
```

CODE EXAMPLE 14-12 SMLogViewerTest *(Continued)*

```
public class SMLogViewerTest implements SMLogViewerResponse {

   public SMLogViewerTest( String server_name, int server_port, int agent_port,
        String user, String password ) throws Exception
    {
        try {
            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
                                "04cd";

            System.out.println("*** Testing Connection ****");
            SMLogin obj = new SMLogin();

            obj.connect(server_name, new Integer(server_port).intValue(),
                user, password, publicKey);
            System.out.println("Successfully Connected");

            SMRawDataRequest req = obj.getRawDataRequest();
            System.out.println("Successfully Authenticated");
            System.out.println(" **Testing SMLogViewerTest **");
            System.out.println(" **Testing logSearch**");

// The SMLogViewerRequest constructor creates a connection with the
// backend logscanner process on an agent machine.
            SMLogViewerRequest lvreq = new SMLogViewerRequest(req,
                server_name, agent_port);

// The following synchronous method returns matched lines for the
// query. This query is set for Syslog file(s), a max. of 20
// matches are to be reported. The query is also set for the search
// to begin from the latest message. The pattern to be matched is
// server_name. Here the user can pass any regular expression
// pattern.

            StringBuffer matchdata = lvreq.logSearch("Syslog", "",
                20, 0, 0, 0, true, server_name, 0);

            if (matchdata == null)
                System.out.println("No match for the query");
            else
                System.out.println("Match data: \n" + matchdata);

// The following asynchronous method requests Syslog file(s) to be
```

CODE EXAMPLE 14-12 SMLogViewerTest *(Continued)*

```
// searched for pattern server_name. This method also accepts the
// SMLogViewerTest class object that has the callback method
// logSearchResponse. This method will not block as the sync.
// method above before returning the data. The last parameter
// for this method could be used for corelating the request
// response ids in case there are multiple async. logSearch requests.


// The async. call is for getting incremental changes.
// To cause something to change, try 'su' to root
// with a wrong password. This should trigger the callback.

            System.out.println("**Testing logSearch Async. call**");
            lvreq.logSearch("Syslog", "", null, this, new Object());

            System.out.println("sleeping...");
            System.out.println("Waiting for additions to Syslog.");
            System.out.println("Try 'su' to root with a wrong password.");

            Thread.sleep(120 *1000);
        }
        catch ( Exception e) {
                System.out.println(e.getMessage());
                e.printStackTrace();
        }
    }

    public void  logSearchResponse(SMRequestStatus status, StringBuffer data,
        Object identifier)
    {
        System.out.println("Async. data = " + data.toString());
    }

    private static void usage()
    {
        System.out.println("usage: java SMLogViewerTest " +
            "server_name server_port agent_port name password");
    }

    public static void main(String[] args) throws Exception {
        if ( args.length != 5) {
            usage();
            System.exit(1);
        }
        else {
            new SMLogViewerTest (args[0], new Integer(args[1]).intValue(),
                new Integer(args[2]).intValue(), args[3], args[4]);
```

CODE EXAMPLE 14-12 SMLogViewerTest *(Continued)*

```
            System.exit(0);
        }
    }
}
```

Resource Access API

This section describes:

- Example: SMResourceAccessTest

Example: SMResourceAccessTest

The SMResourceAccessTest demonstrates the retrieval of generic resources from the server. It will ask the server to verify the existence of the ConsoleMain.class and domain-config.x files, output the content of the file version-j.x and load the image file `cpu16x16.gif`.

CODE EXAMPLE 14-13 SMResrouceAccessTest

```
/*
 * @(#)SMResourceAccessTest.java        1.1 99/09/13
 *
 * Copyright (c) 09/13/99 Sun Microsystems, Inc. All Rights Reserved.
 */

import com.sun.symon.base.client.*;

import java.awt.Image;

public class SMResourceAccessTest {

    public SMResourceAccessTest( String server_name, int server_port,
            int agent_port, String user, String password ) throws Exception {
        try {
            String publicKey =  "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
                                "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
                                "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
                               "470b755b0640af974e7fc70daa6191dff6efa31a09" +
                                "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
                                "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
                                "04cd";
```

CODE EXAMPLE 14-13 SMResrouceAccessTest *(Continued)*

```
            System.out.println("*** Testing Connect ****");
            SMLogin obj = new SMLogin();

            System.out.println("Before Connection Establishment");
            obj.connect(server_name, server_port, user, password, publicKey);
            System.out.println("Successfully Connected and Authenticated");
            SMRawDataRequest req = obj.getRawDataRequest();


// The SMResourceAccess class is used to retrieve generic
// resources from the server. Currently only image data/text
// files retrieval is supported.

            System.out.println("***Testing the SMResourceAccessRequest ***");
            SMResourceAccess resacc = new SMResourceAccess(req);

// The following method is used to verify the existence of the
// file on the server. Depending on the URL scheme, the server
// will use different paths to lookup for the presence of the file.
// eg. for a file xfile:/domain-config.x the server will use
// INTERFACE_PATH env. var. for a file
// cfile:/com/sun/symon/base/console/main/ConsoleMain.class,
// CLASSPATH var. is used.

            String file = "xfile:/domain-config.x";
            if (resacc.fileExists(file))
                System.out.println(file + " exists");
            else
                System.out.println(file + " does not exist");

            file =
                "cfile:/com/sun/symon/base/console/main/ConsoleMain.class";
            if (resacc.fileExists(file))
                System.out.println(file + " file exists");
            else
                System.out.println(file + " file does not exist");

// The following method could be used to get a listing of all the
// files in a particular directory. The directory path specified
// in this call will be relative to the INTERFACE_PATH env. var.
// in the server startup shell script. The following example will
// get resolved under $ESROOT/classes/base/console/cfg dir.
// This dir. is part of the INTERFACE_PATH env. var.
            String[] listfiles = null;
            listfiles = resacc.listFiles("stdimages");
            if (listfiles != null) {
```

CODE EXAMPLE 14-13 SMResrouceAccessTest *(Continued)*

```
                for(int i=0; i < listfiles.length; i++)
                System.out.println("Filename = " + listfiles[i]);
            }
            else {
                System.out.println("Either directory is empty " +
                    "or no files under the directory");
            }

// The following method could be used to read any config. file
// from the server. The server will use the INTERFACE_PATH env.
// var. to resolve the file. The pathname/filename should be
// relative to the dirs. listed in the INTERFACE_PATH env. var.

            file = "version-j.x";
            String filecontents = resacc.getConfigFile(file);

            System.out.println("Contents of the Config. file " + file +
                " = \n" + filecontents);

// The following method could be used for reading an image file
// from the server. The console applications can share such a
// resouce loaded on the server. The pathname of the file is
// resolved using the INTERFACE_PATH env. var.

            file = "stdimages/cpu16x16-j.gif";
            Image img = resacc.getImage(file);

            if (img == null)
                System.out.println(
                    file + " could not be loaded from the server");
            else
                System.out.println(file + " was loaded");
        }
        catch (Exception e) {
            System.out.println("Exception: " + e.getMessage());
        }
    }

    private static void usage()
    {
        System.out.println(
            "usage: java SMResourceAccessTest " +
            "server_name server_port agent_port user password");
    }

    public static void main(String[] args) throws Exception {
        if ( args.length != 5) {
```

CODE EXAMPLE 14-13 SMResrouceAccessTest *(Continued)*

```
            usage();
            System.exit(1);
        }
        else {
           new SMResourceAccessTest (args[0], new Integer(args[1]).intValue(),
                new Integer(args[2]).intValue(), args[3], args[4]);
            System.exit(0);
        }
    }
}
```

Topology Agent API

Example: SMTopologyTest

This example demonstrates how to create a domain, how to show all domains created in the topology agent and how to get all children information under a domain.

CODE EXAMPLE 14-14 SMTopologyTest

```
/*
* @(#)SMTopologyTest.java       1.17 98/09/11
*
* Copyright (c) 09/11/98 Sun Microsystems, Inc. All Rights Reserved.
*/

import java.io.InputStream;
import java.io.InterruptedIOException;
import java.io.OutputStream;

import java.lang.String;
import java.lang.System;

import java.net.Socket;

import java.util.StringTokenizer;
import java.util.Vector;

import com.sun.symon.base.client.*;
import com.sun.symon.base.client.topology.*;
```

CODE EXAMPLE 14-14 SMTopologyTest *(Continued)*

```
/**
* This class is for testing the Topology Client API.
* It shows how to create an domain, how to get all domains created in the
* topology agent and how to get children informaiton under each domain.
**/
public class SMTopologyTest {

public SMTopologyTest(String serverHost, String serverPort,
                      String userName, String passwd)
               throws Exception {

  try {
      String publicKey =
          "687a8398ad4a85077d33b72a94e16ffde0c4ba023e" +
          "9c9ba77b247cc25bd3cd0015bc24b7429916751e68" +
          "1fd02e5ad6eb5345eb7c75b39a1c304e0f000846aa" +
          "470b755b0640af974e7fc70daa6191dff6efa31a09" +
          "431bb5e9848b7dc4cf4b97e1dbca31792d2860ca5a" +
          "5990dfb369e1bcf296274a4e4984c8089329679dd3" +
          "04cd";

    System.out.println("*** Testing Connect establishment and Auth ****");

    System.out.println("Before Connection Establishment");
    SMLogin obj = new SMLogin();
    obj.connect(serverHost, Integer.parseInt(serverPort),
                      userName, passwd, publicKey);
    System.out.println("Successfully Authenticated");

    SMRawDataRequest req = obj.getRawDataRequest();

    SMUserDomainRequest domReq = new SMUserDomainRequest(req);

    System.out.println("******** Create a Domain *******");
    String domName = "test_domain";
    SMUserDomainData domDt = domReq.createDomain(domName);

    if ( domDt == null ) {
        System.out.println("Domain "+domName+" exists");
    } else {
        System.out.println(domDt.getDomainName()+", URL="+
                           domDt.getDomainRootUrl());
    }

    System.out.println("******** Get All Domains *******");
    SMUserDomainData domData [] = domReq.getAllConfiguredDomains();
```

CODE EXAMPLE 14-14 SMTopologyTest *(Continued)*

```
    SMTopologyRequest topoReq = new SMTopologyRequest(req);
    SMTopologyEntityData topoData[] = null;
    SMTopologyEntityData data = null;
    int j=0;
    String name = null;
    String baseURL = null;
    String str = null;
    SMTopologyData topoD = null;
    Vector dataV = null;
    String children [] = new String[2];

    for ( int i=0; i<domData.length; i++ ) {
        name = domData[i].getDomainName();
        baseURL = domData[i].getDomainRootUrl();
        System.out.println("Domain: "+name+", URL="+ baseURL);
        topoData = topoReq.getTopologyInfo(baseURL, null);
        for ( j=0; j<topoData.length; j++ ) {
            data = topoData[j];
            System.out.println("\t"+data.getDesc()+", "+ data.getPollType());
        }
    }

    System.out.println("******** Get Children under the Domains *******");
    for ( int i=0; i<domData.length; i++ ) {
        name = domData[i].getDomainName();
        baseURL = domData[i].getDomainRootUrl();
        topoReq.getHierarchyChildRequest(baseURL, "1000", true,
                        new TopoHierTest(name), null);
    }
  }catch ( Exception e) {
     System.out.println(e.getMessage());
     e.printStackTrace();
  }
  // System.exit(-1);

}

public static void main(String[] args) throws Exception {
    if ( args.length != 4 ) {
        System.out.println("usage: java" +
                " SMTopologyTest server_name server_port user password.");
        System.exit(1);
    } else {
        new SMTopologyTest(args[0], args[1], args[2], args[3]);
    }
}
```

CODE EXAMPLE 14-14 SMTopologyTest *(Continued)*

```
}


class TopoHierTest implements SMHierarchyResponse {

    private String rootName;

    public TopoHierTest(String name) {
        rootName = name;
    }

    public void getHierarchyRootResponse(SMRequestStatus status,
                            SMHierarchyViewData data, Object identifier) {
        if (status.getReturnCode() != SMErrorCode.SUCCESS) {
            System.out.println("Error getting hierarchy root " +
                        status.getMessageText());
        }  else {
            System.out.println("Root data");
        }
    }

    public void getHierarchyChildResponse(SMRequestStatus status,
                        SMHierarchyViewData[] data, Object identifier) {
        if (status.getReturnCode() != SMErrorCode.SUCCESS) {
            System.out.println("Error getting hierarchy children " +
                        status.getMessageText());
        }  else {
            System.out.println("Domain: "+  rootName);
            for ( int i=0; i<data.length; i++ ) {
                System.out.println("\t"+data[i].getName()+", "+
                        data[i].getTargetUrl());
            }
        }
    }
}
```

Exception Classes API

Note – Exception classes do not need separate examples since they are covered within the context of the examples for each class or interface. These class and interface descriptions are presented in the javadocs included as part your install of the Sun Management Center Developer Environment.

PART III Additional Material

This volume includes the following sections:

This section also includes:

CHAPTER 15

Internationalization Guidelines

This chapter covers the following topics:

Internationalization

Sun Management Center consoles and associated GUI clients operate in a global environment. To do this, a mechanism is required to isolate the language dependent code/information from the language independent code and provide a straightforward method for graphical developers to reference the language dependent information. This chapter details this mechanism, and describes specific guidelines for creating internationalized consoles for Sun Management Center.

This chapter also describes the effects of running the Sun Management Center console in non-English environments, where the console needs to handle non-ASCII user input and outlines the methodology for dealing with this sort of information.

This chapter provides implementation details and guidelines for creating internationalized consoles and managing localization information (agents and consoles) for Sun Management Center 2.*x*.

Terminology

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. Sometimes the term internationalization is abbreviated as i18n, because there are 18 letters between the first i and the last n.

Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text. The term localization is often abbreviated as l10n, because there are 10 letters between the l and the n.

Constraints

To maximize maintainability, the Sun Management Center internationalization system must be compatible with the standard JDK 1.2 internationalization mechanism and algorithms.

Assumptions and Dependencies

The descriptions in this chapter assume that the developer is familiar with the JDK 1.2 support for i18n. Concepts such as *locale, ResourceBundle,* and *`properties` file* must be understood completely.

Software Guidelines

Properties Files

The preferred method for managing internationalization information is to store all localized console information in Java properties files. According to the Java specification, properties files contain *key=value* pairs, with each entry separated by newlines. The key corresponds to the programmatic identifier for the internationalized information, which is either hardcoded in the Java program or is contained in the associated console configuration file. With the exception of spaces, the key identifier is free-form, but the proposed key format must be a dot separated list of identifiers that provide some form of hierarchical organization within a functional group. For example, the label for the exit button in the file menu might appear in the properties file as:

```
menu.file.exit.label=Au Revoir
```

Note – Following the equal sign (=), leading and trailing spaces are included in the localized value.

There are two reasons for selecting property files as the preferred mechanism for managing internationalization information. First, display text information can be modified without recompiling Java code, which simplifies the maintenance of the internationalization information. Second, this form of property / resource specification is consistent with the configuration file system used to build the Sun Management Center consoles, thus retaining a familiarity within the product development.

ResourceBundle Class Instances

The use of properties files is the preferred method for managing internationalization information, but the alternate use of *ResourceBundle* classes is also allowed. In this case, the ResourceBundle class is extended (or, more preferably, the *ListResourceBundle* class is extended), with the *handleGetObject()* method being defined to provide the appropriate internationalization information for the provided key. The suggested key format is as described previously for the property files.

Obtaining Resource Bundles / Properties Files

In order to properly obtain the property files and/or resource bundle class instances from the Sun Management Center server or the HTTP server, the appropriate ClassLoader instance for the requesting Java object must be used. For example, a console bean object that was installed in the Sun Management Center server installation area would be loaded through the RMI class loader. As a result, internationalization information for that bean, which is also be installed in the Sun Management Center server area, must be loaded with this particular class loader instance.

Investigation of the implementation of the ResourceBundle class in JDK 1.2 reveals that the static *getBundle()* methods will handle this situation properly. An internal method *(getLoader()*) determines the appropriate ClassLoader instance for the current object context (using a native method *getClassContext()*). As a result, if the bean object mentioned above requests an information bundle through the *getBundle()* method, this process correctly uses the RMI class loader that provided the bean class in the first place. It also properly hashes the cached information, preventing possible conflicts arising from multiple source definitions of the same resource bundles. Thus, we are able to utilize the standard ResourceBundle methods, without modification and maintain standard performance / caching capabilities.

Note – This level of detail for the process description has been provided in the interest of future debugging in case this process fails in the future. If the process does fail, an appropriate alternative must be provided to properly support the directed information retrieval.

Independent Client/Bean Usage

An independent Java application (one running outside of the Sun Management Center configuration file framework) will have several methods for directly obtaining localized information.

UcInternationalizer Class

This is the class that is used by the `i18n` and `i18n-specific` type converters to parse the *functional_group:key* specification, load the resource bundles, and translate the key. To use it in your Java application, make the following declaration:

```
import com.sun.symon.base.utility.UcInternationalizer;
```

This declaration presents a static method *translateKey()* that takes in the *functional_group:key* value and returns the corresponding localized value according to the rules for the i18n converter specified previously. Also, an overloaded version of this method takes a fallback resource in the event that the specified internationalization key is not found. So, for example, if you are looking for the localized form of the *mybean* label from the *ConsoleGeneric* functional group, the code might be:

```
String lbl = UcInternationalizer.translateKey(
"base.console.ConsoleGeneric:mybean.label");
```

or

```
String lbl = UcInternationalizer.translateKey(
"base.console.ConsoleGeneric:mybean.label",
"Help Me");
```

As previously described, the `com.sun.symon` part of the location can be dropped for convenience.

An additional feature of this *translateKey()* method can take a single argument that can be inserted into the final localized string value. This is a very simple interface to the Java *MessageFormat* interface. The following example illustrates the use of this simple argument facility:

```
String arg = "brown";
String lbl = UcInternationalizer.translateKey(
"example:dog.story(" + arg + ")");
```

In this case, *example:dog.story* is the key used to find the resource bundle and localized string information and *brown* is the argument value. Now, if the example `properties` file contains the line:

```
dog.story = The quick {0} fox ...
```

the result is “The quick brown fox ...”. The value provided in the argument replaces the {0} argument indicator in the localized string value.

Two important things to note about this argument form of the internationalization key:

- The argument is *not* parsed or converted to a localized value. The argument is intended for instance values that have been user specified and are not stored in properties files. If an argument of the form *functional_group:key* is given, that is how it can be substituted in the main localized string.
- If the argument substitution value {0} does not appear in the localized string in the properties file, no error will occur, but the argument value will not appear. Being able to hide the argument for certain locales may or may not be a feature.

Direct ResourceBundle Management

Developers who are creating Sun Management Center client applications or beans in Java also have the option to manage the i18n of their application as defined by Java. At any point where internationalization information is needed, the programmer can request the property bundle for the required *functional_group* and lookup the localized information within the bundle using the information *key*. As for a previous

example, if you require the internationalized resource directly for the key `menu.filebutton` for the functional group `console` you can use the following code:

```
/* Get the internationalized button label */
ResourceBundle bundle = ResourceBundle.getBundle("console");
String label = bundle.getString("menu.filebutton");
```

The *getString()* method is used regardless of whether the resource bundle information was stored in a properties file or class definition (see "Properties Files" on page 312). You will also need to handle the *MissingResourceException*, that is generated whenever the requested resource bundle or key entry is missing. Note that the *getBundle()* call must receive the full path to properties/class files associated with the functional group (that is, the `com.sun.symon` prefix must be specified, if applicable).

This method of obtaining internationalized information can also be used directly within Sun Management Center bean objects, using the code provided above. Doing this is not suggested, as the Sun Management Center infrastructure provides the AWX type converters and the UcInternationalizer class to centralize internationalization information retrieval. One advantage of this centralization is that the UcInternationalizer code can be modified to warn about undefined key values or mark values that have been properly internationalized, allowing potential internationalization problems to be identified prior to the final localization process/testing.

Formatted Messages

There can be circumstances where a variable piece of information must be embedded into an internationalized string. Examples of these embedded bits are module instance names or dynamic object counts. This embedding is available as a standard Java feature through the *java.text.MessageFormat* class. This class is not fully documented here. See the appropriate Java documentation for more details. However, there are two specific examples that cover almost all of the cases where embedding is required. These are:

- where a string value needs to be inserted into a localized string, and
- the embedding of an integer value into the localized string

The first example is where a string value (that is, a module instance name) needs to be inserted into a localized string. This corresponds to the simple argument mechanism available through the UcInternationalizer class described in “UcInternationalizer Class” on page 314. The code that accomplishes this is as follows:

```
String name = "Your Name Here";
String baseStr =
UcInternationalizer.translateKey("example:advert");
String lbl = MessageFormat.format(baseStr, new Object[] { name } );
```

In this case, the localized value for the *advert* key in the *example* functional group marks the point where the text is to be embedded by the marker “{0}”. For example, this value might be defined as:

```
advert=Space For Rent [{0}]
```

Thus, the final lbl value is “Space For Rent [Your Name Here]”.

Situations where multiple strings need to be embedded can be accommodated by passing multiple entries in the object array and marking their locations by {0}, {1}, {2}.... The values in the braces indicate the argument number, so that the embedded string values can be rearranged as required. An embedded string can be dropped without any errors by leaving the marker out of the localized definition.

The second example is the embedding of an integer value into the localized string. This can be accomplished by turning the integer value into a string and using the embedding described above. This method would not properly accommodate locales where integer values are not represented by left-to-right roman digits. There are Java classes that can be used to manually convert the integer to a string value according to the locale, but the MessageFormat class handles this automatically. The code that accomplishes this is:

```
int val = 20;
String baseStr =
UcInternationalizer.translateKey("example:found");
String lbl = MessageFormat.format(baseStr,
new Object[] { new Integer(val) } );
and the localized found value in the example properties file looks
like
found=I found {0,number,integer} grapplegrommets.
```

The only real differences between this and the string example above are that the object passed to the format() call is an integer instance and that the marker in the localized value contains additional formatting information that indicates the argument is an integer. Again, multiple integers can be used by indexing the arguments {0...}, {1...}, ... and passing multiple values in the Object array. For information on other available formats, see the appropriate Java documentation.

Handling Non-ASCII Input

The majority of this document focuses on the presentation of "static" information in the Sun Management Center console/client applications (through the use of fixed keys and localized lookup tables), but a global application also needs to properly handle user input in any locale. For a purely Java application this is not a problem, as the string class is 16-bit based and all AWT/swing input/output fields use Unicode to fully support all languages/character sets. In many cases, developers within the Sun Management Center framework are insulated by the Client API and do not need to deal explicitly with non-ASCII user input.

The Sun Management Center agents are not written in Java, nor do they properly handle Unicode or 16-bit text. In fact, the Tcl parsing mechanism (as of Tcl 7.5 that is used by Sun Management Center 2.*x* software) has difficulty handling 8-bit non-ASCII character values (>0x7F). The standard Java Unicode string conversion function (which is used by the server process to convert from string to SNMP Octet strings) uses UTF-8 encoding that does produce non-ASCII 8-bit characters. As a result, a non-ASCII Unicode string sent in raw form from a console/client application to the agent can potentially cause a problem, as the agent Tcl engine tries to parse the UTF encoded data.

Two types of data have specific handling requirements, as described in the following sections.

Data Only Stored in Agents

This type of data is stored in the agent for later use by a Sun Management Center console/client application, and is never directly used by the agent itself. Examples of this type of data are domain names, object names/descriptions, graph labels, and so forth. Typically, this data is managed by methods within the Client API (insulating the users of the data from the specifics of the agent). As a result, developers of functionality within the Sun Management Center Client API as well as developers of client applications that directly communicate with the agent (bypassing the Client API) are responsible for massaging the non-ASCII data into a form that can be safely handled by the agents.

The *UcListUtil* class contains two static methods that are used to encode / decode non-ASCII string data into an ASCII form that can safely pass through the agent Tcl parser. `UnicodeToAscii()` takes a Java string corresponding to user input and returns an ASCII encoded version of the information suitable for storage in the agent. `AsciiToUnicode()` reverses this process to regenerate the non-ASCII string information for display. The encoding scheme is designed with an escape sequence that will not interfere with Tcl escapes, does not appear in standard data (so that the decoder is safe to use generically) and is compact in order to reduce data overhead.

Data Stored in and Manipulated By Agents

This represents information that is typically used to configure the agent. For example, alarm limits and actions, filenames, and so forth, all fall into this category. In this case, the user input should be specifically restricted to prevent the entering of any non-ASCII information, as there is no encoding mechanism relevant to the agent that can be used.

No central mechanism (as the Client API was before) exists to handle this case, since every situation is different. As a result, the client or the Sun Management Center bean must handle this situation properly and provide appropriate user feedback if the user enters non-ASCII text. Ideally, the bean will refuse to accept the data and request that the user enter ASCII-only text where required.

In the Sun Management Center console, the vast majority of this type of information is handled through the attribute editor bean, which has a protocol for indicating that entries are directly used by the agent and cannot accept non-ASCII text.

Agent Internationalization

This section describes the various pieces of information that are defined by the agents themselves for console display and how the internationalization information is to be specified (either in the infrastructure or in the module definitions).

Note – This information refers to static display strings that can be internationalized. Dynamic structures such as data input or status messages are not internationalized in Sun Management Center 2.*x* software.

Objects / Classes / Properties

This internationalization information is typically used in the hierarchy / topology / table displays associated with the browser tab in the details window. Each instance of MANAGED-OBJECT, MANAGED-PROPERTY-CLASS and MANAGED-

PROPERTY needs to define an internationalization key to allow the console to lookup a localized name for the object, class and property, respectively. This key information is also defined in other agent objects that may appear in the console (such as the mibman.modules hierarchy).

The specification of the internationalized name is made alongside the *value:mediumDesc* specification (that is, in the module models file (See "Module Building"). However, in this case the *consoleHint:mediumDesc* value is defined. The value corresponds to the *functional_group:key* specification used in the Console configuration files and by the UcInternationalizer class.

For example, an entry for the user-managed object in the `solaris-standard-models-d.x` file:

```
user = { [ use MANAGED-OBJECT ]
mediumDesc = User Statistics
consoleHint:mediumDesc = base.modules.solaris:user
```

This module has a private properties file contained in the base/modules Sun Management Center directory.

The next section describes the definitions required for the MANAGED-MODULE object.

Modules

Module instance internationalization involves two areas: internationalizing the name of the module instance, and internationalizing the parameters used in loading/editing the module, which are described in the following sections.

Module Instance Naming

There are two basic types of modules:

- those that are single instance (that is, Solaris operating environment)
- those where multiple independent instances can be loaded by the user (for example, `fscan`).

First consider the single instance modules. In this case, there is no specific instance identifier, so the "name" of the module can be specified in the MANAGED-MODULE object of the module hierarchy using the *consoleHint:mediumDesc* value described above. To reduce confusion, this description specification is identical to the *i18nModuleName* parameter described below. In fact, if the *consoleHint:mediumDesc* value is not found in the module root object, the *i18nModuleName* parameter can be used in its place.

The second type of module (multiple instances) is more complicated as the name of the module has to be shown in the console in combination with the user specified instance description. In this case, the *i18nModuleName* parameter and the *consoleHint:mediumDesc* must be different (and fully specified). The *i18nModuleName* is the general (the non-instanced) name of the module. For example, for the `fscan` module this is File Scanning, which is specified as described in this section.

However, when the module is viewed in the browser hierarchy, the user specified instance of the module must also appear. In this case, the *consoleHint:mediumDesc* value is used to obtain the base internationalization key, and then the instance details are appended to this key as an argument. For example, the root object of the `fscan` module has the following specification:

```
consoleHint:mediumDesc = base.modules.fscan:moduleDetail
```

When the module description is obtained (for the System Log instance), the following key is sent to the console:

```
base.modules.fscan:moduleDetail(System Log)
```

The definition of the moduleDetail key in the `fscan.properties` file is:

```
moduleDetail=File Scanning[{0}]
```

From “Formatted Messages” on page 316, the {0} identifier indicates the location where the argument (in this case, the instance name) appears, so that the module name that appears in the console is `"File Scanning [System Log]"` for example. If the *consoleHint:mediumDesc* is not specified, the key falls back to the one specified in the *i18nModuleName*, which does not have this marker defined, so the module instance information is not displayed in the console.

Module Parameters

Several parameter specifications (contained in the `<module><-subspec>-m.x` file) must be internationalized or involve internationalization, as described below. Note that in almost every case, the internationalization value is a *functional_group:key* specification as used in console configuration files or by the UcInternationalizer class described previously.

- The first (and most obvious) piece of internationalization is the descriptions of parameters and parameter groups. The parameter group description is what appears in the “tab” label, and is specified for the param group as:

```
?param:?description = base.modules.default:moduleParam
```

  The parameter description is what appears to the left of the data entry field, and is specified as follows (for the module parameter):

```
?param:module?description = base.modules.default:module
```

- The second parameter-related internationalization involves parameter values that must be internationalized. Examples of these are the *i18nModuleName*, the *i18nModuleType* and the *i18nModuleDesc* parameters, that respectively provide the key specification for the name of the module (“File Scanning”), the type of the module (“Local Applications”) and a longer description of the module. In each case, the value of the parameter is the familiar *functional_group:key* identifier and a line of the following form must appear for each parameter to indicate to the module loader/editor that the value is to be parsed to obtain a localized value:

```
?param:i18nModuleName?i18n = yes
```

  These parameters must be read-only, otherwise the user can edit the localized value and send that back as the key (with unusual effects). There is also a parameter format specification of `i18ncomment` that does not actually indicate that the key is to be translated, but instead indicates that the value is a multiline label to appear as unbordered text (for the description).

- The third parameter internationalization point is the handling of list parameters, where a choice of several values are sent to the agent but an internationalized list is displayed to the user. In this case, the list format appears as follows:

```
?param:parameter?format = \
list:val1,fn_group:key1|val2,fn_group:key2|...
```

  Thus, the list values are separated by the pipe character (|). Each entry contains the selection value to send to the agent, followed by a comma and the *functional_group:key* specification for the internationalized text to appear in the selection box pulldown.

The final internationalization-related parameter settings involve those parameters that are input by the users that are not processed by the agent, but are only used for console display. Currently, this is only used for the instance name provided by the user for each loaded module instance.

This is different from the instance parameter, which is used as a context key in the agent. In this case, the user can enter localized text in the native language the console is currently running in. Behind the scenes, the module loader/editor can use the ASCII encoding/decoding mechanism, described in "Data Only Stored in Agents" on page 318, to translate the user entered instance name into a form that is safe to store in the agent. To indicate that this (or any other) module parameter is permitted to take non-ASCII text from the user, the following format specification is required:

```
?param:parameter?format = unicode
```

This format should not be used for any parameter that can be utilized by the agent, as there are no facilities in the agent to decompose the encoding scheme.

Attribute Editing

Developers who are adding or modifying the shadow maps that are edited through the attribute editor, or who are customizing the shadow entries that are edited by a specific object must internationalize properly the labels and possibly the values associated with these attributes. The following sections describe the three components of internationalizing attributes.

Attribute Groups

The set of shadow groups, as defined by the value:shadowGroups() settings in the edited object, is a single level Tcl list where each pair of values defines the access key and the internationalized information key. For example, a definition can appear as:

```
shadowGroups()=Info
base.console.ConsoleGeneric:editGroup.info \
  Module base.console.ConsoleGeneric:editGroup.module
```

Each pair of values has first a simple (English) key that is used to organize the attributes in that group, and second the familiar *functional_group:key* specification, that is used to determine the localized text to display in the group selection tab of the attribute editor.

Scalar Attributes

For each scalar attribute, there is a single internationalized description that is given as the third entry in the shadowSpec() definition for that attribute. For example, the object name definition can appear as:

```
shadowSpec(name) = \
"name {Object Name} base.console.ConsoleGeneric:editAtt.name
 string {} ro scalar";
```

The provided key is used to find a localized label for that attribute to display alongside the value in the attribute editor. In practice (observe base-shadowmap-d.x) this value is never specified. Instead, a blank entry {} appears in its place. If a blank internationalization name is given for a scalar variable, the key is programmatically determined by appending the shadowmap key (*name* in this case) to the string `base.console.ConsoleGeneric:editAtt.`, arriving at the result shown above for this example.

There are other internationalization issues related to the value of the attribute and how the user edits it (as specified by the format indicator in the shadow specification). For example, the value can be an internationalization key that is translated prior to displaying (read-only) in the editor, or the value can be a selection list where the value provided to the agent corresponds to an internationalized value to display in the pulldown list. The following list describes the various formats that relate to internationalized values:

`i18n`–the value is an internationalized key that is translated prior to display in the editor. These attributes are always read-only.

`i18ncomment`—identical to the i18n format, except the presentation is slightly different to accommodate multline values.

`list:...`—specifies a selection list with mapping between the agent values and the internationalized display information. See the module parameters description for more information on the exact specification of this format.

`unicode`–enables the user to enter non-ASCII text as a value for this attribute. Generally not used as the agent cannot utilize the entered value (but there are always exceptions).

Vector Attributes

With the exception of the attribute label, these attributes are identical to the scalar attributes described above as the data type and format information is identical for all members of the vector. However, each attribute in the vector can have a different description (and hence different internationalization information). This description

is obtained using the method specified in the shadowInfo() specification for the vector attribute. The name returned from this method is appended to "base.console.ConsoleGeneric.editAtt." to construct the internationalization key for that entry of the vector. For example, a numeric alarm limit might have two attributes in the vector, the bigger and smaller limits, for which the shadowInfo() method returns too-big and too-small. In this case, the `ConsoleGeneric.properties` file will have the lines:

```
editAtt.too-big=Error if bigger than
editAtt.too-small=Error if smaller than
```

Note – Rule parameters, which are technically vector attributes, are handled by their own naming mechanism as described in the next section.

Dynamic Tables (RFC1903)

For certain modules, there are tables in which rows can be dynamically added / deleted / enabled / disabled by the user (using the RFC1903 protocol). For example, the `fscan` module enables the user to add new patterns to be scanned for in the file. In this case, the row add / edit window needs two pieces of internationalization information: the column entry label and (optionally) the format of the column data. The entry label (that appears alongside the editor field) is obtained from the setting of *consoleHint:mediumDesc* as described in "Objects / Classes / Properties". The value format is obtained from the *value:dataFormat* setting, which can be any of the standard attribute editor format types. With regards to internationalization, the two formats of interest are the `list` setting and the `unicode` setting, as described in "Scalar Attributes" above. In this case the unicode setting is a common thing for the row description, which is not used by the agent and is returned to the console to label the rows in the table as the user describes them.

Rules

For each parameter defined in a rule (by the rule:<rule>-editparam setting), the attribute editor will present an entry field that needs an internationalized label for that parameter. The internationalized label key is constructed from several sources. First, the base path of the *functional_group* is specified in the rule:<rule>-keypath setting and the name of the module is appended to give the full group specification.

The key entry in the properties file is determined by combining the string “editAtt.” with the name of the rule and the name of the rule parameter. For example, consider an “example” module that has the following lines in the rules definition block:

```
rule:da_rule-editparam = "one two"
rule:da_rule-keypath = base.modules
```

The internationalized label entries for the “one” and “two” rules are contained in the functional group *[com.sun.symon].base.modules.example* and the .properties file contains the following entries:

```
editAtt.da_rule.one=Rule One
editAtt.da_rule.two=Rule Two
```

Installation / Setup Script Internationalization

Solaris software-based add on products that require their own scripts to handle installation and / or setup will need to internationalize these scripts if they are to be used globally. The basic method here is that which is used in C programming. A portable object file is created (`filename.po`) that contains the text displayed by the scripts. The `msgfmt` command is run on this file to create a message object file (`filename.mo`). As the script runs, it obtains localized messages from the message object file. This guideline does not describe all of the details of internationalizing C programs. Refer to the man pages for `msgfmt, gettext` and `textdomain` for details.

The way a script obtains localized messages from its message object file is through the TEXTDOMAIN variable. This variable is set to the name of the script’s message object file. Sun Management Center software provides a function called **setup_textdomain**, that can be used by add-on products to establish a domain for their own localized messages. You can call this function in the beginning of your script and pass it the name of your message object file. While your script runs, it obtains messages from its own message object file.

If your script must call a function within the core Sun Management Center scripts (other than **setup_textdomain**) , set TEXTDOMAIN to the core Sun Management Center setting for the duration of this call. If your script is invoked from the core Sun

Management Center script, save the current setting of TEXTDOMAIN before changing it to your domain value. When the function ends and control returns to your script, reset TEXTDOMAIN to your own setting. For example:

```
# SyMON script running... calls your script
domain_save = TEXTDOMAIN
setup_textdomain SUNW_MY_DOMAIN
# your script running...
setup_textdomain domain_save
# call core SyMON script function...
# core SyMON script function ends, returns to your script
setup_textdomain SUNW_MY_DOMAIN
# return to your script running...
```

If your script is invoked independently of the core Sun Management Center scripts, and you must call a core Sun Management Center script function, the value of the core Sun Management Center setting for TEXTDOMAIN is SUNW_ES_SCRIPTS. Simply call `setup_textdomain` with this value before making the core function call.

Of course, if you have no need to call a core Sun Management Center function, then you only need to set TEXTDOMAIN at the beginning of your script. If you are invoked from the core script, reset it to the core setting upon completion.

Note – If your add-on product has packages that can be installed using the core Sun Management Center install script, your message object file must be in `$PKG_DIR/locale/$LANG/LC_MESSAGES`.

CHAPTER 16

Graphical User Interface Guidelines

This section covers the following topics:

These UI guidelines can help the developer of Sun Management Center software-related products to create a user interface that is easy to learn, easy to use, and highly consistent with the existing Sun Management Center user interface.

Note – This chapter provides guidelines for making your software consistent with the various components of the Sun Management Center graphical user interface. However, these guidelines should not be construed to imply that support for a particular component necessarily exists in the Client API.

Sun Management Center 2.1 software is built on Java 1.2, which placed limitations on the implementation of some UI elements, including:

- Keyboard navigation
- Keyboard shortcuts
- Default buttons
- Drag and Drop

Although these elements are not fully implemented in Sun Management Center 2.1 software, they are important and are discussed in the following section. If you implement using a version of Java that supports these features, you can use these elements.

Consistency

Consistency is a broad term that can be interpreted in a number of ways. Here are three simple definitions, with examples, to keep in mind.

- When users must distinguish differences quickly and reliably, the visual cues to make that distinction should be consistent for different uses and in different locations in the application.

  Example:

  Sun Management Center software uses alarm badges to indicate trouble states on objects, and the OK state is signalled implicitly with no badge. This makes it easy for users to detect anomalies in the console view (trouble states stand out). Adding an OK badge is inconsistent with the Sun Management Center model.

- The location of information is important, as users can find information quickly and automatically if it is always in the same place (where "place" is a main window, a tree hierarchy, a dialog window, a menu, and so forth). In adding new functionality, use the established locations for the same types of information. If the established location runs out of space, make it larger before adding a new location.

  Example:

  Sun Management Center software presents detailed alarms information in the Alarms tab of the Details window. If you have additional alarms information to present, put the new information into the existing Alarms tab panel. If the main Alarms panel cannot hold the your information, try putting it into child-windows of the Alarms panel (for example, dialogs).

- Layout of information inside windows should arrange the most important or most frequently-used information consistently with existing windows. This way, users can keep their old habits of where to look and where to point.

  Example 1:

  Sun Management Center domain manager places buttons that modify table elements to the right of the table.

  Example 2:

Sun Management Center attribute editor gives important information about the object (object label, object location, variable name, variable's current value) at the top of the window. Consistency requires that new dialogs provide the same kinds of information and put it in the same place.

Information Sources

If you have questions that this guide does not address, here are some additional sources of information:

- Use Sun Management Center 2.*x* software as your guide.
- Conduct usability testing. This can range from hiring a professional for complete testing, to showing your proposed design to end-users and getting their feedback.
- Consult published guidelines. While not always consistent at the detail level because they were developed under different operating systems, they agree on the basic principles. Internet links to published guidelines include:
  - *Java Look and Feel Guidelines* by Sun Microsystems, Inc.
    `http://java.sun.com/products/jlf/dg/index.htm`
  - *Windows Interface Guidelines for Software Design* by Microsoft Inc.
    `http://mspress.microsoft.com/prod/books/963.htm`
  - *Macintosh Human Interface Guidelines* by Apple Computer, Inc.
    `http://developer.apple.com/techpubs/mac/HIGuidelines/HIGuidelines-2.html`
  - Links to consortium guidelines for UNIX user interfaces:
    `http://www.acm.org/sigchi/hci-sites/`
- Employ the services of a professional UI/interaction designer. A directory of designers is available at:
  `http://www.acm.org/sigchi/hci-sites/CONSULTANTS.html`

Main Console

FIGURE 16-1 shows the Main Console Window.

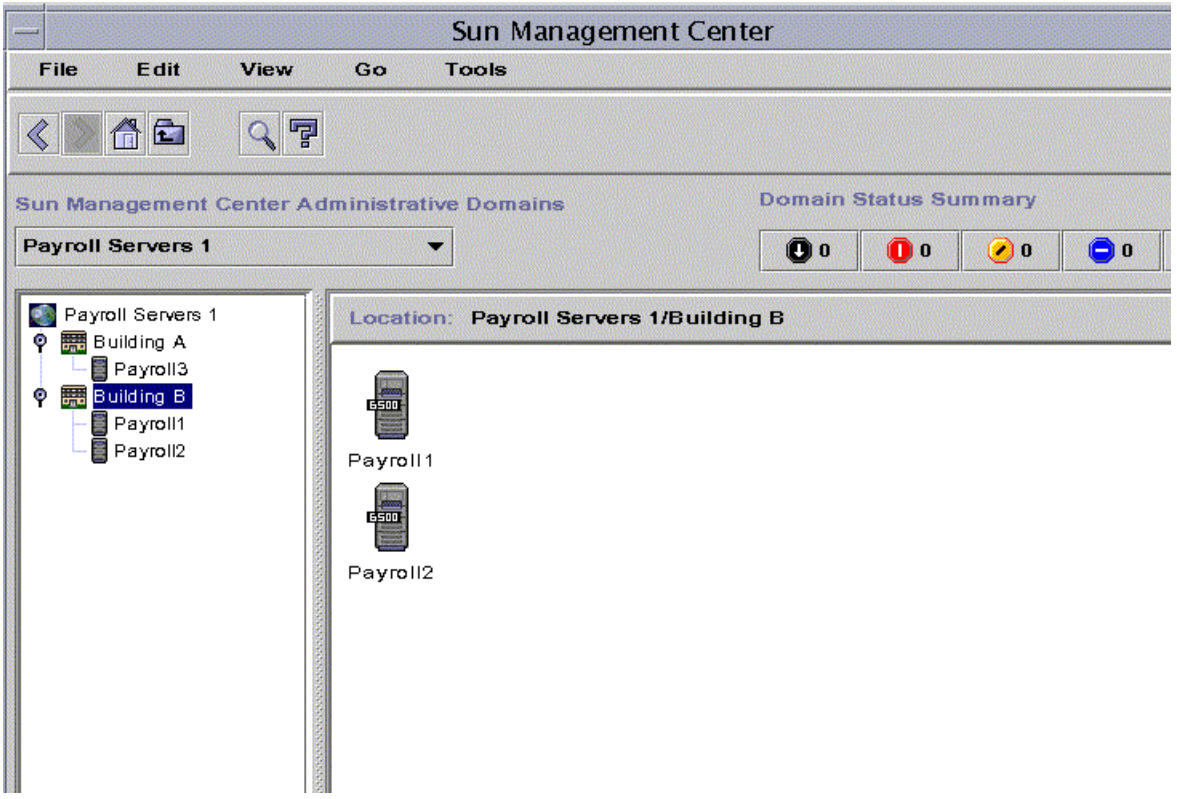

FIGURE 16-1 Main Console

The main console is divided into seven sections: pull down menus, navigation buttons, Sun Management Center Administrative Domains pulldown menu, alarm buttons, two panels and a status line.

- Menus: The pull down menus going across the top of the console window are labeled File, Edit, View, Go, and Tools.
  - File menu: The items under the File are Domain Manager, Remote Domain Manager, Set Home Domain, Sun Management Center-Console. These items operate on domains. If you plan to add Domain management functionality, it should be launched from this menu.

- Edit menu: The items under Edit are Create an Object, Create a Connection, Delete Object/Connection, Rename, Modify, Cut, Copy, Paste, Paste Into and Select All. The Edit menu items are used to modify objects, not domains.
- View menu: The items under View are Topology Layouts and Set Topology Background. If you add any view-like functionality to Sun Management Center, put it in this menu.
- Go menu: The items under Go are Back, Forward, Home, Up, History and Search. These act the same as web browser buttons and contain the same functionality as the navigation buttons located directly below the menu bar.
- Tools menu: The items under Tools are Details, Attribute Editor, Graph, Discover, and Load Module. Most add-on applications that enhance Sun Management Center software should be listed here.

- Navigation buttons: The navigation buttons act the same as web browser buttons.
  - Back takes you to your most recent topology hierarchy location.
  - Forward moves forward in the navigation history.
  - Home takes you to your home domain.
  - Up arrow on folder traverses the object hierarchy upward.
  - Search magnifier icon opens the Go To window.
  - Help "?" icon opens the Sun Management Center online Help.
  - About box: An About Box can be brought up by clicking on the Sun Management Center icon to the far right of the navigation buttons. The About Box includes information such as the product name, version number, build number, and copyright information.
- Sun Management Center Administrative domain pulldown menu: Lists all of the domains that the console can administer.
- Alarms buttons: The Alarms buttons are in a horizontal array under the Sun Management Center logo. Clicking on the alarm button opens a window with a summary of the objects that are reporting problems to the level of the alarm. The icons appear on both the alarm buttons and as badges on the hardware icons. If you add new alarm levels (and hence additional buttons) make sure that buttons are arranged in descending order of alarm severity.

Server Object Representation and Object Management

Sun Management Center software provides a host-centric user interface. That is, management is done from the perspective of the managed objects (servers and workstations). The entire main console is devoted to creating, displaying and editing

managed objects. In a large or complex enterprise server installation, the number of managed objects can be large, and monitoring for errors or anomalies requires the ability to find malfunctioning objects quickly.

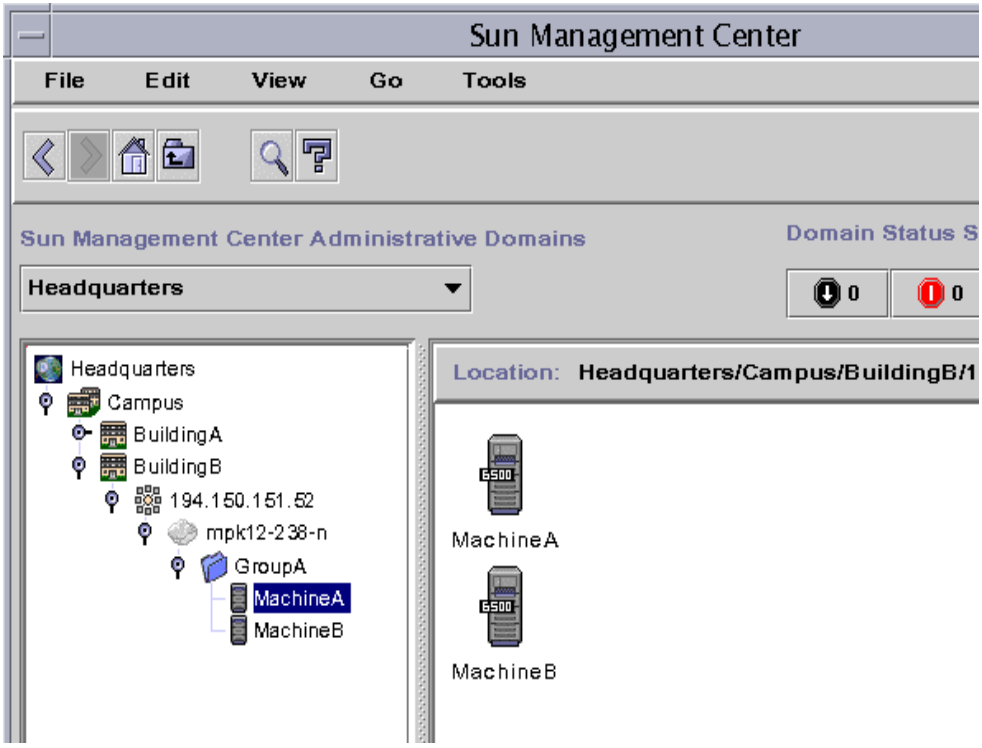

FIGURE 16-2 Main Console Window with Hierarchy and Topology Views

The server objects are represented in both a tree view list in the left panel and a layout view in the right panel (FIGURE 16-4). Object icons can be any size, but most of the existing icons are 42 x 42 pixels in the right-hand topology view and 16 x 16 pixels in the left hierarchy view. If you add new object icons, their sizes should be approximately the same as the existing icons.

The layout view, presented in FIGURE 16-4, is potentially very powerful, as it can be used to indicate the location of a managed object in real physical space, such as a server room. This enables monitoring or service personnel to pinpoint the exact location of a trouble source. Background images for this purpose can be added by selecting Set Topology Background from the View menu.

Typical Sun Management Center software customers have hundreds, and in some cases thousands, of host objects to manage and for this reason the main console, and all of the sub-consoles, must scale to large numbers. The main console has several features designed specifically for scalability:

- Main console uses both main panels for showing objects. The left panel is a hierarchical list; the righthand panel enables user-specified layout views (network view, list view, bus view, and so forth) and the ability to superimpose managed object icons on a graphical background.
- Grouping: Users can create group containers and place managed objects in them, in any way that makes sense for that user's management requirements. Groups can be moved around with their contents unchanged just like objects.
- The highest level of grouping in the Sun Management Center software console is the administrative domain. Domains can be created and deleted, populated with objects automatically or manually, and have user permissions (security) set specifically for them.
- The Sun Management Center console provides a Go To function (similar to search) that enables the user to find an object by name. This permits quick access to a particular object (including groups) within a large number of managed objects and groups.
- Actions specific to particular managed objects are quickly accessible through the right-mouse-button. These actions vary according to the object.
- Sorting and filtering permits the user to reduce the amount of information in a particular view by showing only the most relevant information. Sun Management Center software currently supports sorting and filtering at the data-table level, but does not support these functions at the topology object level.

Guidelines for Modifying Topology Views

Do not enhance or change the presentation of the Sun Management Center software agent in such a way as to interfere with the features described previously.

- Add new group and object types: Study the Create Topology Object dialog carefully (including the contents of all menus) before adding any new object types. The type you need may already exist. If not, make sure any new object types are represented in all the right places (Create Topology Object dialog, Discovery filters dialog), and that each has a full set of icons (large, small, tagged, untagged).
- Use good object management dialogs: FIGURE 16-3 is an example of the Sun Management Center Domain Manager dialog. Its central feature is a list of the objects. Along the right side, arranged vertically, are buttons that provide the main actions that can be done on the objects. Selecting an object enables all buttons whose actions can be applied to the selected object.

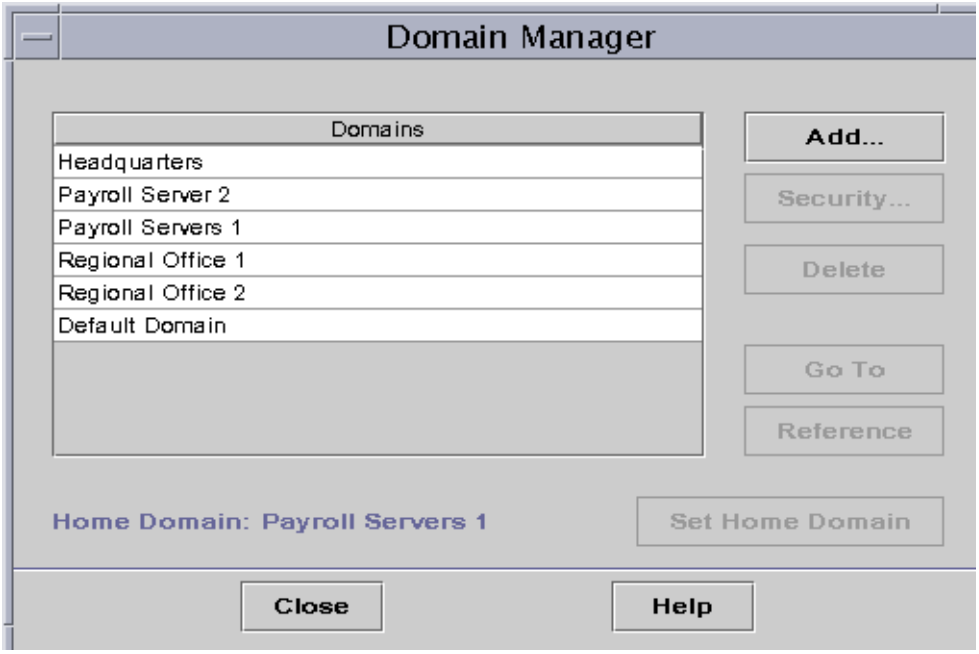

FIGURE 16-3 Domain Manager

- Add and subtract right mouse menu items freely: Ensure that they consist only of items that apply to a particular object.
- Sorting/filtering: If you are developing an enhancement or addition to Sun Management Center software that can provide sorting and/or filtering, make sure that the results of sorting/filtering are consistent with the presentation of objects by Sun Management Center software.
  - Make sure filter/sort actions are presented near the site of their action, for example, just over the list of objects.
  - Make sure the action applies equally to both console panels (layout view and list view).
  - Provide an explicit `show all` and/or `unsort` option.

Layout View

The topology view (FIGURE 16-4) displays the object selected in the hierarchy view, along with any peers that share its container.

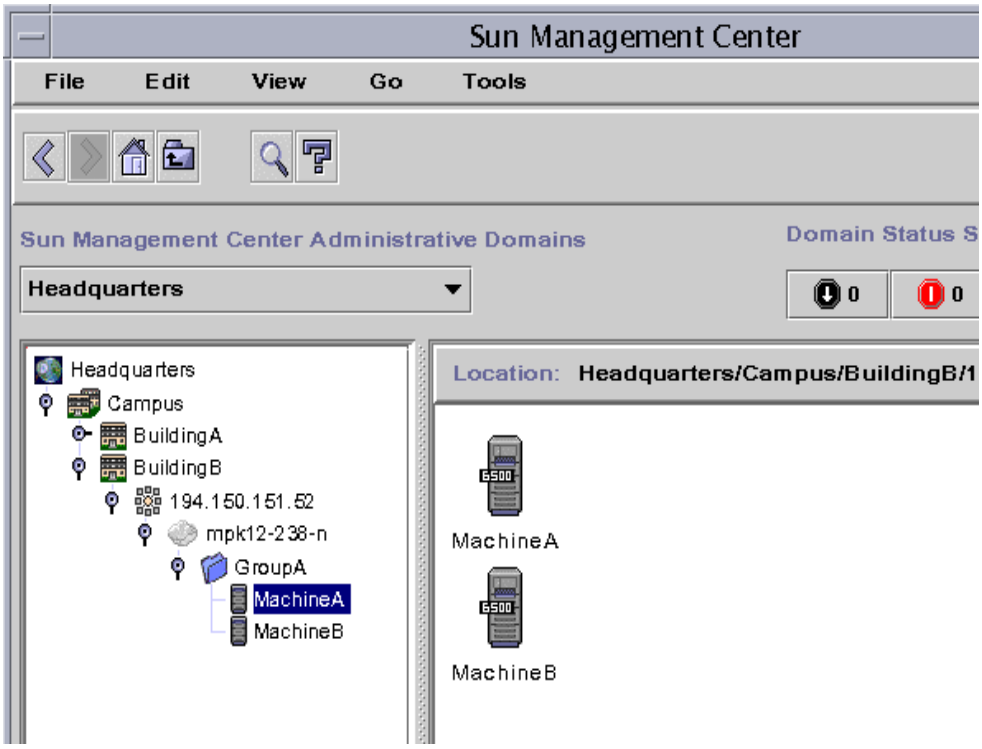

FIGURE 16-4 Main Console Window with Hierarchy and Topology Views

FIGURE 16-5 shows an example of how the user can load a background `gif` file and place the items in a physical location, in this case a server room.

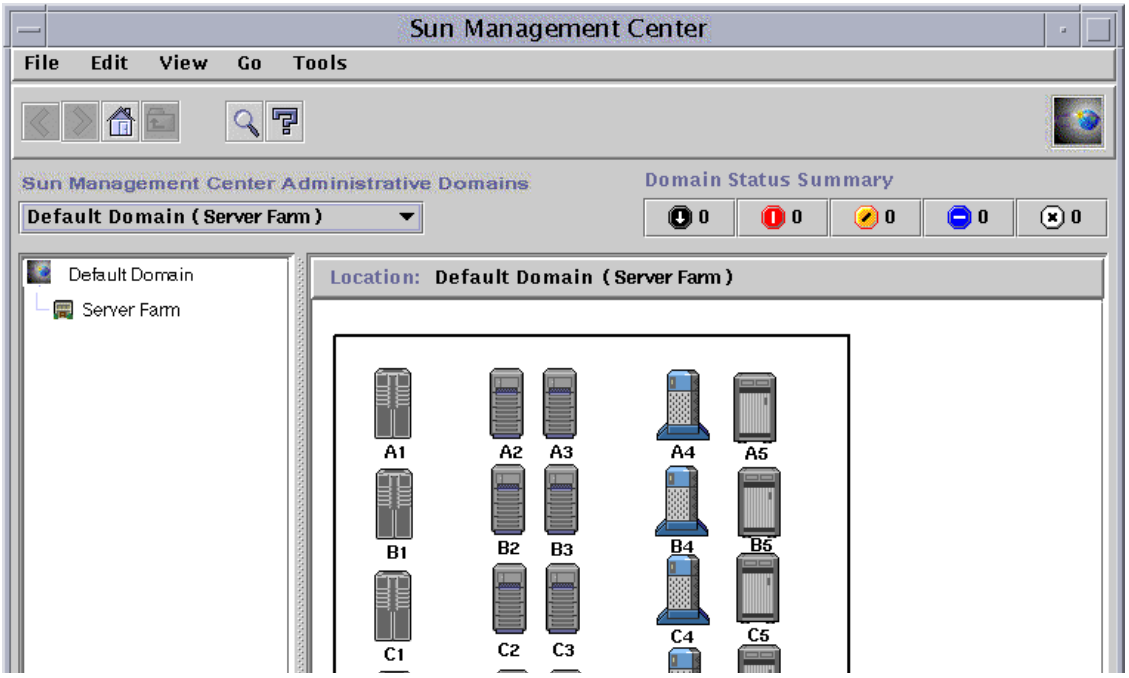

FIGURE 16-5 Topology View

Object Layouts

Only users with permission to create objects and groups are permitted to change layouts.

The choices are accessed through the Views menu:

- Network (unconstrained)
- Grid (object positions constrained to lie in a rectangular grid)
- List (objects listed vertically)
- Bus (objects linked with lines in a bus pattern)
- Star (objects linked with lines in a star pattern)
- Spoked ring (objects linked around a ring)

Layout affects only the group within which it was chosen, but it is visible by any user console in which that group can be seen.

Status line

A Status line is located at the bottom of the console window. Be sure to give the user feedback about what is going on. The Sun Management Center status line gives messages such as “Downloading physical view images, please wait.” “Paste was successful”, “Object was created.” This is a good place to put error messages, such as “Object not found.”

Status Messages

Status messages must be shown left-justified at the bottom of every window (FIGURE 16-6). Messages from a previous action must last only until the next command is requested. When a new action is initiated, the status field must clear first and then show a message indicating the ongoing status of the new action, as that becomes available. Fonts and colors for status fields should be as defined in the Fonts section.

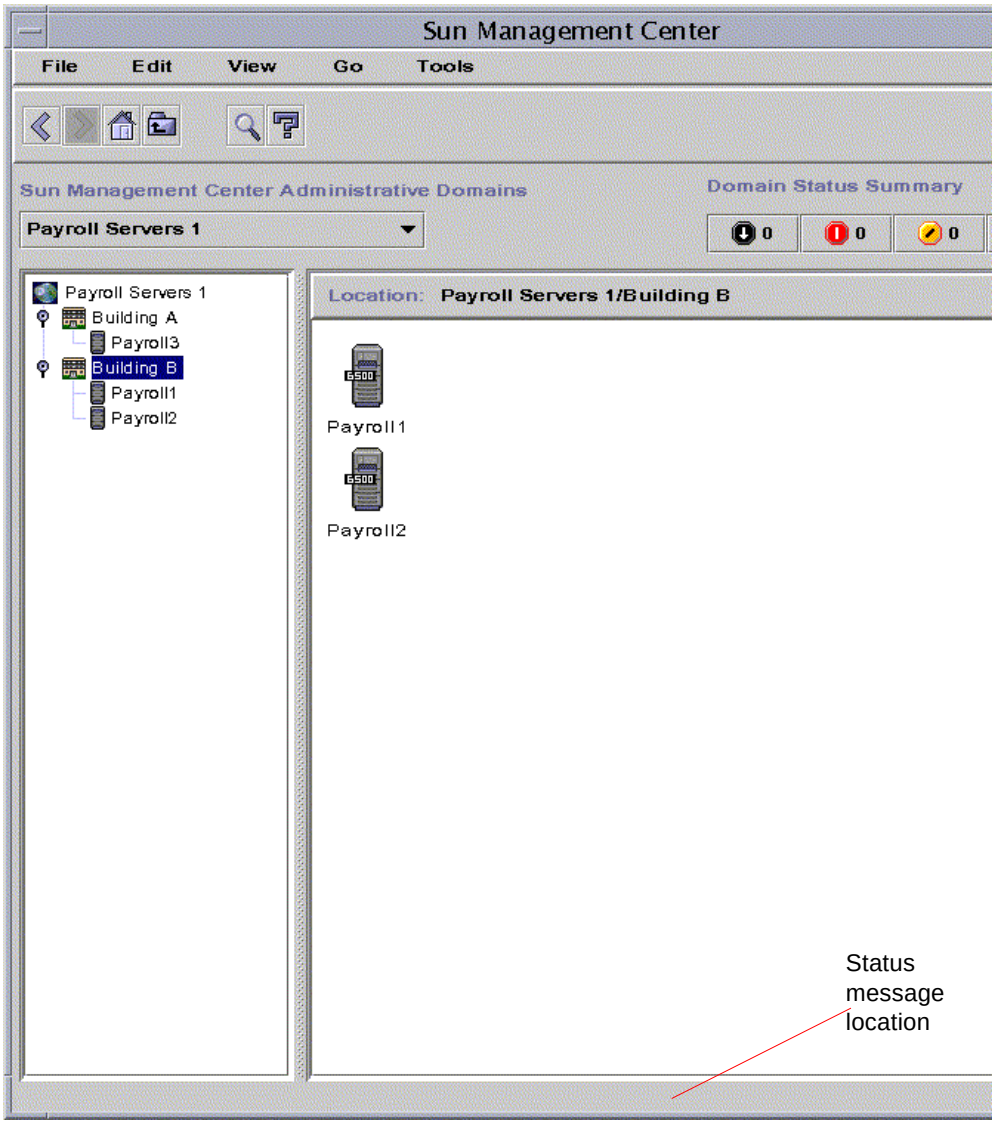

FIGURE 16-6 Status Message Location

User Input

Mouse Actions

The mouse actions are consistent across the Main Console, Details windows, and dialogs. This is an especially important consistency feature, because mouse actions are used quickly and automatically by most users. They do not want to have to stop and think about how to use the mouse. Mouse actions are defined as follows:

- Left button click:
  - Objects—If the cursor is on an object in the topology or hierarchy view, left button click highlights and selects the object.
  - Widgets—If the cursor is on a widget (for instance a checkbox or pulldown menu), a single left button click operates the widget.
  - Text fields—On entering a dialog containing text fields, the user is not required to click the mouse inside the first field. Rather, the contents of the first field are highlighted with the cursor positioned at the far right of the contents.
- Left button double-click:

  This action opens a topology object. In Sun Management Center software, open is defined as follows:

  - In the topology view:

    If the object is a host, then double-click opens the Host Details window and the object in the topology and hierarchy views maintains the selection highlighting.

    If the object is a container, then double-click opens the container to show the contents.
  - In the hierarchy view:

    If the object is a closed container, then double-click opens the container in both views (drops open the contents list in hierarchy view, navigates to and shows contents of container in layout view and the name of the container in the location field above layout view).

    If the object is an open container, then double-click closes the container in hierarchy views (snaps up contents list) and also navigates to and shows contents of container in topo view and the name of container in location field above layout view).

If the object is a node, then double-click opens the Details window and the object in the hierarchy view maintains the selection highlighting.

- Right button click:

  In all views, this action opens a popup menu to provide commands that can be executed on the selected object. The popup menu is context sensitive; the exact commands appearing there vary according to the selected object.

- Left button Click-and-Drag:

  This action enables you to drag objects to change their positions inside the righthand layout view of the Main Console. Dragging objects over other objects and dropping them has no effect. Drag-and-drop are not supported in Sun Management Center 2.*x* software.

Selection Highlighting

Selecting Objects

- In the hierarchy view, highlighting must be done with a solid medium-blue rectangle enclosing the label (with label text inverted to white).
- In the topology view, highlighting must be done with a medium-blue open rectangle enclosing the icon, and a solid medium-blue rectangle enclosing the label (with label text inverted to white).

When a selected object is put into the cut mode (by selecting the Cut item from the edit menu), the selection rectangle must go to the dashed-line form. The dashed-line rectangle must encircle the entire icon plus the text area (for example, remove text highlighting).

Multiple selection of topology objects is supported. Multiple selection cam be done two ways:

- Drag-select
- Shift+select for second and additional objects

De-selecting Objects

- Any selected object must be deselected when another object in the hierarchy or the topology is selected.
- Any selected object or group must be de-selected with a mouse click elsewhere in the window.
- Clicking again on a selected object must not de-select it. The cursor must be elsewhere to de-select. The exception is:

- When a selected object has been put into cut mode (by selecting the Cut item from the Edit menu), then removing the cut mode from the object must be done by explicitly clicking the object again. The dashed-line rectangle is then removed and the regular selection highlighting is replaced on the object.
- When multiple objects have been put into the cut mode, then clicking any object again will remove the cut mode from all the objects.

Keyboard Navigation Shortcuts

Like mouse actions, consistent keyboard action are important because users rely on keyboard navigation to be quick and automatic, requiring little thought.

Sun Management Center software follows the Microsoft guidelines for keyboard navigation (*The Windows Interface Guidelines for Software Design*, Microsoft Corp., 1995).

Note – Keyboard methods are not well supported in Java 1.*x* software, on which Sun Management Center software is built. Only minimal keyboard navigation is provided.

The appropriate exit buttons for a dialog depend on what the dialog is intended to do:

- Use OK and Cancel for confirmation dialogs and for dialogs that consist of a single discrete action that the user wants to complete quickly.

  Example (confirmation): Do you want to save this?; Launch discovery now?, Are you sure you want to delete that object?
- Use OK, Apply, and Close for dialogs in which objects are created, or properties edited, using multiple fields. The OK and APPLY buttons are only sensitized if fresh data has been entered into the window since the last click of APPLY. Close will close the window without taking any action on data currently entered in the dialog.
- Use a Close button only if the changes made in a dialog or window take effect immediately and are not accumulated over multiple input fields before being committed.
- Use Action Names (for example, Load, Save, Create) as the default action (instead of OK) in cases where the appropriate word is obvious, and short enough to fit on a button.
- When buttons are arrayed horizontally at the bottom of a dialog, then the order is:
  - Leftmost—the default action, for example: OK, Save, Create

- Middle—additional options, for example: Apply, Reset, Clear
- Rightmost—the cancellation or closing action, for example: Cancel, Close, Done

---

Note – Java 1.*x* does not provide an easy way to specify a default button activated by keypress. If you develop on a version of Java that supports this, use the Java Look and Feel guidelines.

Table Appearance and Behavior

The appearance and behavior of tables throughout Sun Management Center software must be consistent with respect to:

- Table contents
- Color
- Fonts
- Table position
- Rows
- Columns
- Growth under window resizing
- Cell, row, and column selection
- Sorting and filtering

Refer to the following illustration.

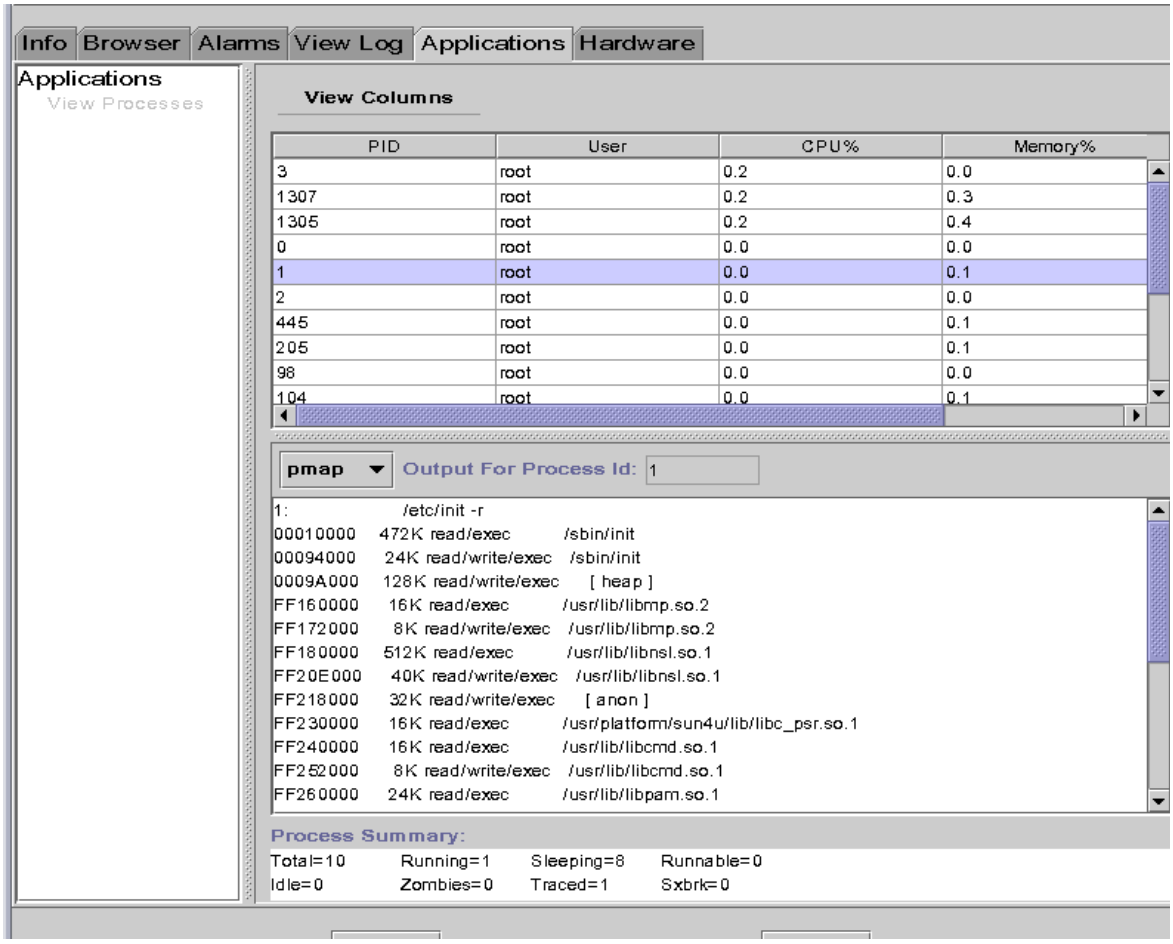

FIGURE 16-7 Table Details Window

Table Contents

- Types of contents—Tables are used to show the values of properties, for listing domains and users, and for listing alarms information. Tables can contain strings (the names of variables), numerical values on a variety of unit scales (percent, Mbytes, and so forth), and graphical icons.
- Labels—Row and column labels are optional at the discretion of the designer. When they exist, the first letter of each main word must be capitalized. The name must include, in parentheses, the unit in which the values are shown. For example, CPU Usage (%).

- Contents with rules—When a variable or any table row has a viewable and/or editable rule associated with it, the name (and unit) must have three dots (ellipsis), For example, Disk Usage (%)... appended to it.
- Table titles—If a table is the only or primary element of a window or panel, the panel label serves as a table label (for example, Processes tab labels the processes table). If a table is not the only element of a window or panel, or is a secondary element of a window or panel, the table must have a label top-left justified and outside of the table border.
- Justification of contents:
  - Row labels must be left-justified, with a 3-pixel margin between the far left edge of the cell and the first pixel of contents.
  - Column headings must be centered.
  - Text phrases and messages of varying length must be left-justified always (and preferably placed in nth column), with a 3-pixel margin between the far left edge of the cell and the first pixel of the contents.
  - Numeric values must be right-justified always (with a 3-pixel margin between the far right edge of the cell and the last pixel of the contents).
- Alphanumeric strings that are not phrases in a human language can be right- or left-justified depending on table layout needs (at the designer's discretion).
- Column widths are variable according to the typical length of information provided in the given column, with fixed minimum and maximum widths. Do not make column labels significantly longer than the longest value provided in the column, as it is wasteful of space.

Color

- Text color: Table information must be in black text on a white background. Column headers must be black text on Java-table-widget grey.
- Alarm color: When a cell in a data table has an alarm state associated with it, cell background color must change to reflect the type of state. Coloring inside the cell should not go all the way to the cell borders, but rather stop 1 pixel short on all sides. This is to allow space for a selection color to be shown for the cell as well. See below.

Note – In the Alarms Console window, Alarms are signaled with the appropriate alarm icon rather than a background color.

- Cell Selection color: When the user selects a cell or row, the selection must be indicated by a medium-blue open highlight rectangle, enclosing the cell. In the case of property tables that can have an alarms color in the cell, the selection rectangle goes outside of the alarms color area, and inside the cell border.

Table Position

- Tables that are the sole occupants of a window pane are center-justified inside the pane and remain centered irrespective of changes in the window width. Tables that appear in the same pane with other tables are left-justified with the left margin set to a value that centers the largest (widest) of the tables at the standard (default) window width.

Cell, Row, and Column Selection

- A table can be defined to be row-selectable only (for example, the Alarms Window), in which case clicking in any column selects the entire row.
- A table defined as cell-selectable must allow selection of any and all cells by clicking inside the cell. In such a table, however, clicking on the row label must select an entire row.
- Column selection is not currently used. However, for column-specific sorting this setting may be necessary.
- Cell Selection color—When the user selects a cell or row, the selection must be indicated by a medium-blue open highlight rectangle, enclosing the cell. In the case of property tables, which can have an alarms color in the cell, the selection rectangle must go outside of the alarms color area.

Colors

Sun Management Center software follows the Java software look for the colors of windows and dialog with the following extensions and exceptions.

- Status fields must be bold black on the grey background of the window.
- User-editable text fields must be black text inside an enclosing box with white background and black border. At the designer's discretion, the box can be inset.
- Non-user-editable text fields must be black text on the grey background (no enclosing box).
- List views, icon views, tables, charts and graphs must have white backgrounds.
- Menus must have a grey background when dropped open.
- Field / Widget labels must be Java Blue.

Fonts

Sun Management Center software follows the Java software font guidelines. Consult `http://java.sun.com/products/jlf/dg/index.htm` for details.

Graphing

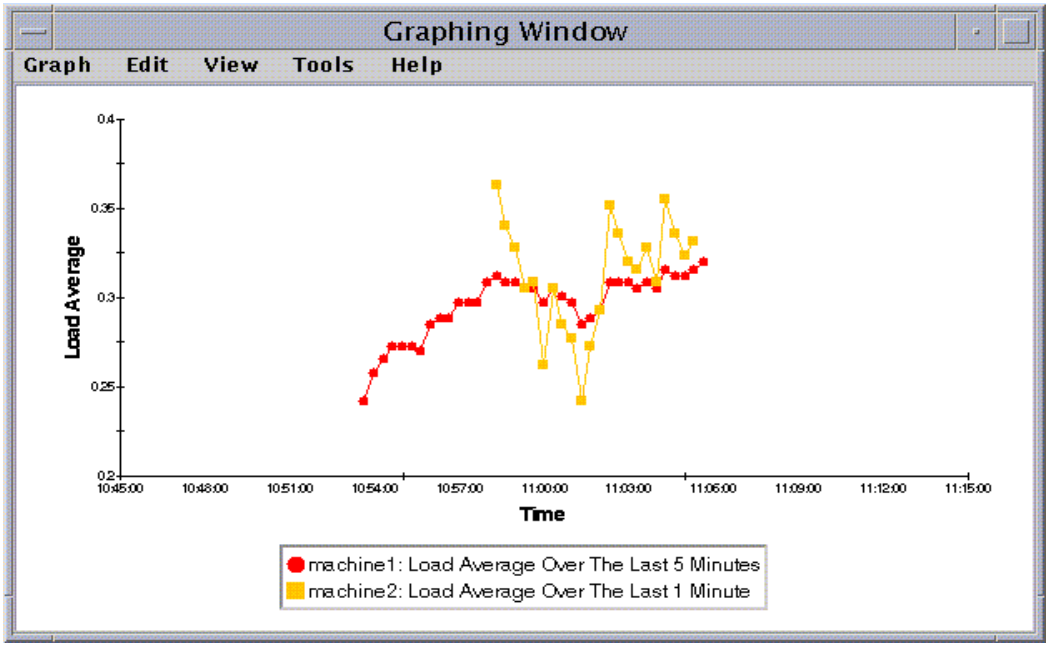

FIGURE 16-8 Graphing Window

Sun Management Center software allows graphing of any numerical data variable with respect to time. Up to five variables can be plotted on the same graph. The white background is essential for making plotted points and their corresponding axis values highly visible.

- Graphing specifications are made at the data level, from within the table showing the data.
- Graphing specifications can be saved and reinvoked (by specification, not by data) from the main console Tools menu.

- All graph labels (titles, legends, axis ticks, and so forth) are fully user-customizable both in terms of contents, position, and whether they are shown or hidden in a particular graph. Graphing features are edited in dialogs that are opened by menu items.

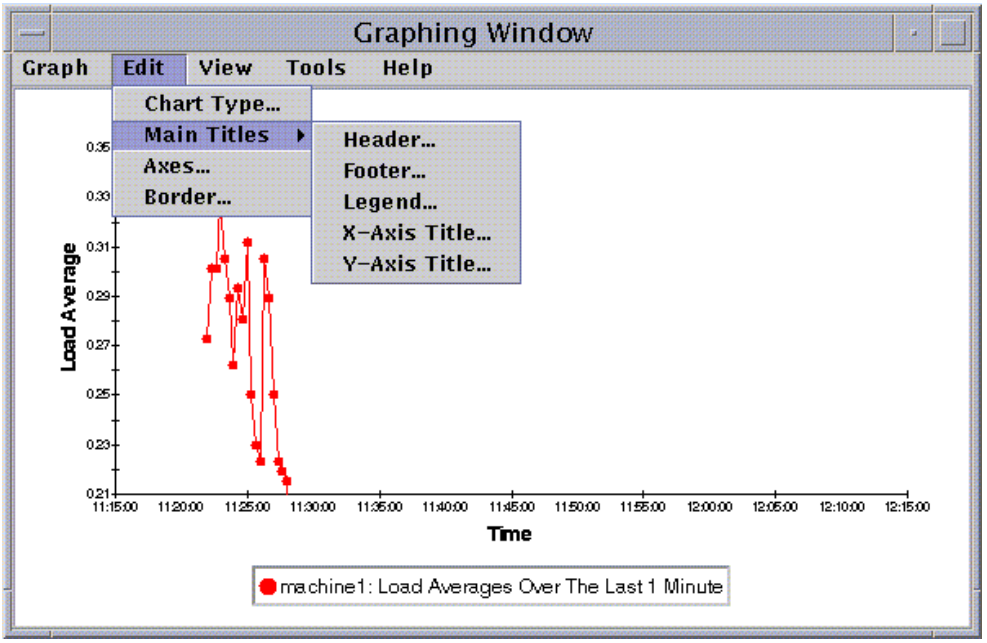

FIGURE 16-9 Graph Header Title Editing Dialog

In this example, text can be entered, styled, and positioned. The Apply button is useful here in enabling the user to see the results of the changes before exiting the dialog.

If you intend to enhance the existing graphing system or add new graphing functionality, follow these guidelines closely.

Caution – Changing the axis scale and tick setting can potentially destabilize the actual or perceived behavior of the graphing function. Make such changes carefully.

- Keep the white background, as visual detail is important in a graph.
- Use the Sun Management Center 2.x dialog structure for implementing the same or similar settings.
- Always provide an Apply button and apply the changes directly to graph so the user can see the results before exiting the dialog.

- If you wish to provide an easier or more direct method of invoking dialogs than those in Sun Management Center 2.*x* software (which invokes them from menus), the following two methods are equally good:
  - Double-click mouse on the intended element (or in intended area for a title) to open the appropriate dialog.
  - Right-mouse click on the intended element (or in intended area for a title) to open a menu of appropriate actions.

Property Setting Dialog

Sun Management Center software enables property settings for the following, at a minimum:

- Managed servers—properties such as label, description, and type
- Module parameters (at the point they are loaded)—instance name and description
- Module run-time scheduling (at any time)—cyclical, one-time only, and so forth
- Data variable alarm thresholds—value at which alarm of given severity is generated
- Alarm actions—script to run or command to execute on generation of the specific alarm
- History—where and how to log data value
- Security—access permissions to objects

Most of these settings are done in the Sun Management Center Attribute Editor (AE). The AE has a tabbed structure (folder tabs at top) that gives it extensibility. Here are some guidelines for modifying or enhancing the Attribute Editor and/or for adding new property-setting dialogs.

- If you add a new object, enable setting of the same properties that the Sun Management Center program already supports. You can also enable additional properties to be set.
- If you add properties, add them to existing dialog panels instead of creating additional property dialogs for the same object.
- If you cannot fit them into existing panels, then add a tab to the existing Attribute Editor. Make sure that the functionality is apportioned between them in a distinctive way, and that the names given reflect those differences very clearly.
- When laying out property-setting dialogs, be consistent with the Sun Management Center Attribute Editor (where most property settings are made).

In the following example, the History setting tab for a data value inside a particular managed object (host machine) is shown (FIGURE 16-10).

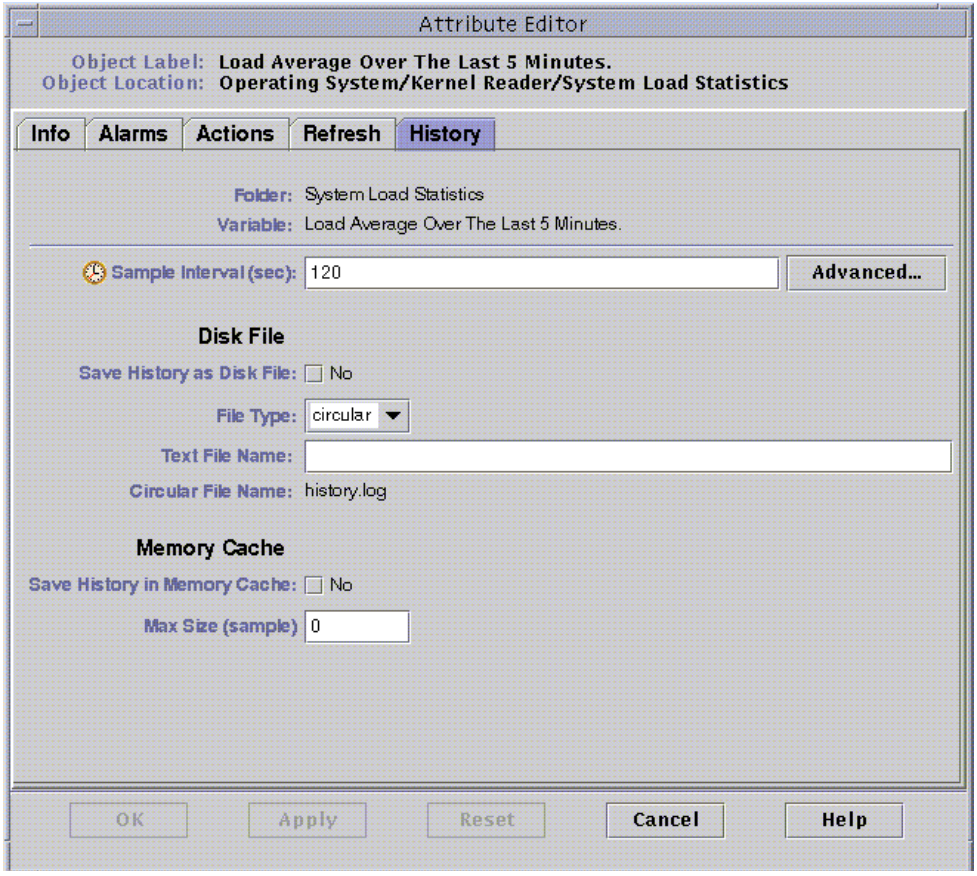

FIGURE 16-10 History Tab of Attribute Editor on a Data Variable

The information shown in the preceding figure is typical, in that all instances of the Attribute Editor follow a similar pattern. This pattern is:

- At the top, above the tabs, there is general identifying information about where the information comes from: object label and object location.
- Below this are the tabs that provide extensibility to the window.
- Typically, more detailed identifying information about the data value whose attributes are being shown goes below the tabs: folder and variable. A delineating line appears at this point, above the actual settings.
- Bold-text headings in black separate areas of slightly different functionality (Disk File versus Memory Cache, in this example).
- Every setting has a label in Java-blue text. Note that the settings are in vertical stacks, colon-justified.

- Read-only settings (history.log in this example) are shown without a box around them, against the grey background of the panel.
- Dialog buttons are discussed in detail elsewhere, but, in the case of the Attribute Editor, should *always* include:
  - Apply, which changes the setting so user can see the result in situ (inside the table where the item lives) before exiting the dialog.
  - OK, which applies and closes the window automatically.
  - Cancel, which closes window without applying any actions.
  - Help

Optional buttons

Reset is especially useful in the case of large panels with a lot of settings. Users occasionally make enough mistakes that is it better to start over. Reset reverts to original (current when dialog opened) values, not to a blank state. Use a button labelled Clear for users who want to set all widgets.

Things to avoid:

- Do not make text fields longer than needed. This is a stylistic issue but more importantly, the length of a field gives the user a cue as to valid data that can go into the field. For example, if the field holds two-digit numbers but is 72 characters long, the user can be misled.
- Do not provide text fields for complex expressions. Instead give the user explicit widgets for setting complex expressions.
- Do not use a label (for example., yes/no) to the right of a checkbox.

Time Setting

Time setting is a commonly-used function in management applications. The following guidelines stress optimal design:

- Use the same design and the same time database everywhere.
- Get as much as possible onto one dialog. Time-setting is inherently complicated and the more the user can see all in one view without relying on memory, the better.
- Put only the simplest and most frequently used functions in the main dialog. Ask the user/customer what they need, what they use most, and then place those functions frontmost. Place less used and expert features in a separate dialog

behind a button. Use sufficient labels. In time-setting more than any other functionality, it is easy for users to become confused. Generous labelling can help. For example:

- Time of day to begin recording (hh:mm):
- Date to begin recording (mm/dd/yy):

- Group similar items together under headings, rather than repeating long phrases over again. An improvement to the previous example is shown here:
  - Begin recording:

    Time of day (hh:mm):

    Date (mm/dd/yy):
- Make the time/date format explicit, either in text as shown above, or by artful use of time-setting widgets. For example, clicking arrows to change time values of the selected element (day, in the date setting).
- If possible, provide *both* widget *and* typing options.
- Compute as much as possible for the user, including the effects of daylight savings time (user simply checks/unchecks a box), leap years, holidays, time zones, and so forth.
- If a 24-hour clock is used, be sure to specify legal value(s) for midnight (24 or 0).
- For internationalization, give users a choice of date/time formats.

Alarms

Sun Management Center software has an event signalling system that spans every aspect of the product, from the main console to the individual data table cells.

If you plan to modify or augment this alarm functionality, it is important to maintain consistency with this system.

Alarm System

- Domain Status Summary buttons: Buttons at the top of the main console give summary counts and open windows with filtered views of the alarmed objects in the domain. The summary count only counts the object's highest severity event.

  Example: If a server has both a yellow alarm and a red alarm condition, the alarm will be counted in the red total, *not* in the yellow total.

- Alarm badges: These are shown in a variety of places, on buttons, affixed to hardware icons, and inside tables. When affixed to hardware icons, only one can be affixed at a time and, like the Domain Status Summary buttons, the severity corresponds to the worst alarm currently existing on the server. In the Alarms console of the Details window, all alarm severities are shown by default.
  - Alarm badges are affixed to hardware icons by centering the alarm badge at the bottom righthand corner of the HW icon. Alarm badges in the layout view (large hardware icons) are 16 x 16 pixels. Alarm badges in the hierarchy view (small hardware icons) are 12 x 12 pixels.
  - Alarm badges must be distinguishable and look good in all of the various locations that they are used.
- Alarm colors inside data table: When a data variable enters an alarm state, the corresponding data table entries become shaded in that color. These colors correspond to the colors of the alarm badges.
- Sun Management Center Alarms Console: Inside the tabbed Details window (see Details window section), this tab contains information on *all* alarms related to the object whose Details window you are looking at.

As mentioned in the main console section, when modifying Domain Status Summary buttons make sure any new / modified buttons and icons are consistent with existing ones:

- If you add new alarm severities, the corresponding badges must conform to the basic pattern of existing Sun Management Center badges, that is, identical background shape, distinctive color, and distinctive internal design.
- When adding / modifying alarm severities, remember to use the correct corresponding color inside the data table cells.
- Modifications to the alarms table in the Alarms Console must follow the Table guidelines (see Table section). Modifications to show / sort dialogs should follow Dialog guidelines (see Dialog section).

Details Window

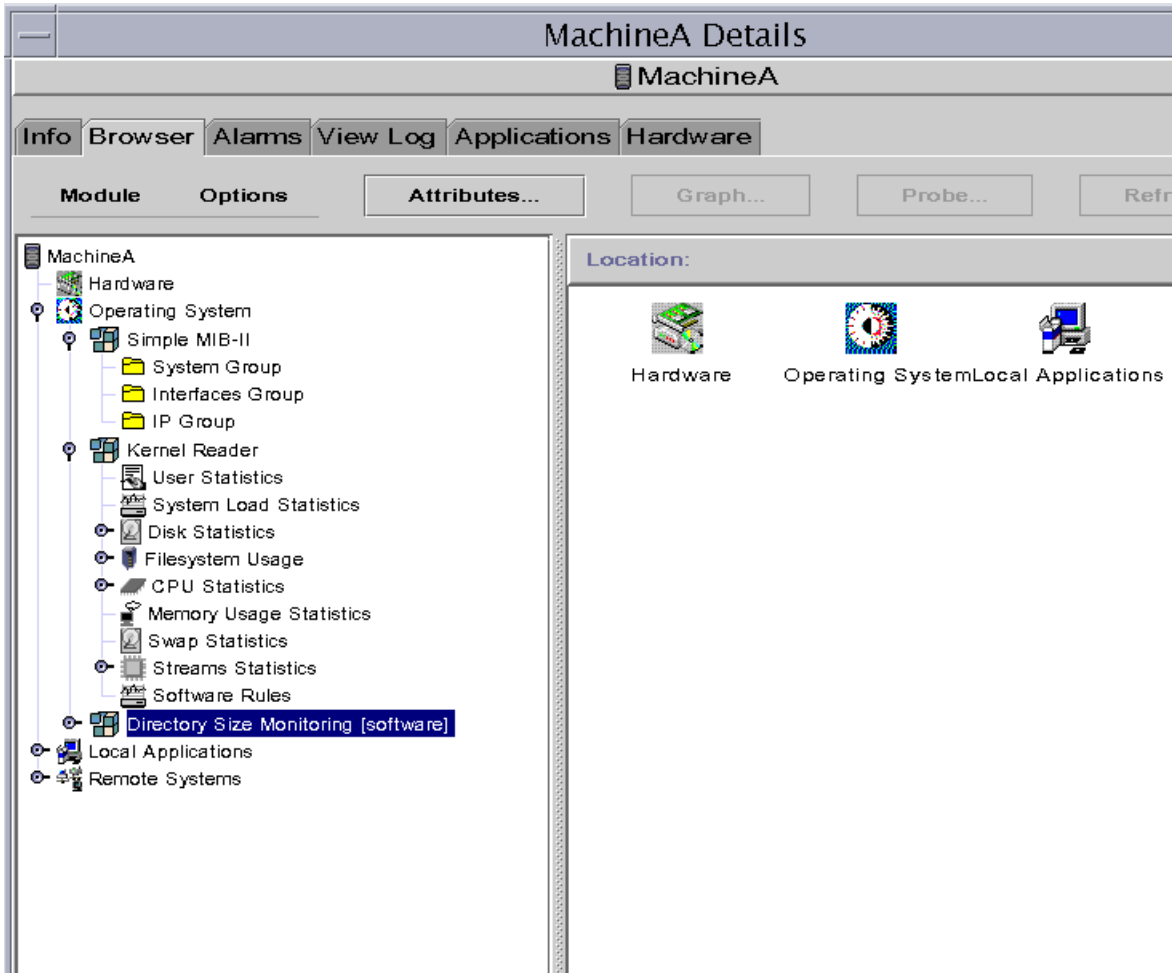

FIGURE 16-11 Browser Details Window

The Sun Management Center Details window provides a wide variety of data for a managed object.

This window has six tabs in Sun Management Center 2.1 software; Info, Browser, Alarms, Processes, Log View, and Configuration. The tabs are an extensibility mechanism that product developers can use to enhance the capabilities of the software.

- The tabs are the top level of navigation in the Details window. Menus, buttons and other navigation tools are not to be added above the tabs, but can be added below the tabs.
- When adding buttons, panel-specific buttons go at the top above the data area, and the general dialog buttons (shared by every tabbed panel) go at the bottom.
- If you plan to add functionality to the Details window, you need to decide if it belongs in one of the tabbed categories, or its own new one. It is always best to add new functionality to existing panels. If you cannot do this, then try launching a dialog (child window) from the existing panel.

  Example: You want to add the ability to annotate the Alarms window to include notes from the system administrator. Add a button on the Alarms Console that spawns a dialog box.

The Browser and Configuration panels each have a two-part structure similar to that of the main console, but the hierarchy at left contains subobjects of the host object, and the right panel contains subobjects and detailed tables. If you are adding a tab, consider creating a hierarchy of subobjects, where applicable.

The Details window has a status area at the bottom, like the main console. Use this area to give users status feedback. The Details window can be iconified and will continue to update data tables, update the logs, and register alarms.

Any subwindows spawned from this window will close when this window is closed, with the exception of the Graphing window.

CHAPTER 17

Sun Management Center 2.1 Developer Environment Packaging

This chapter covers the following topics:

The information provided in this chapter will be based on the packaging of the example modules provided as part of the Developer Environment.

The following documents are suggested reading for more detailed explanation of Solaris software packaging:

- `Pkginfo (4)`
- `Pkgmk (1)`
- `Prototype (4)`
- `Pkgmap (4)`
- `Depend (4)`
- `docs.sun.com` (*Application Packaging Developer's Guide*)
- `docs.sun.com` (Search for "Packaging")

This chapter provides examples and a list of suggested Sun Management Center software packaging practices with more details on the configuration of the packaging files.

Packaging `Helloworld_01`

The following section describes the packaging of the `Helloworld_01` module (`src/examples/modules/helloworld_01/package`).

Makefile

The makefile is set up to look for the components to be packaged in the parent directory of the package directory (`src/examples/modules/helloworld_01`).

This is accomplished by supplying the `pkgmk` utility with the '`-r [path_to_components]`' option, which in this example is '..':

```
COMPONENT_ROOT = ..
DEMOeshw1:
$(PKGMK) -o -r $(COMPONENT_ROOT) -d .
```

The contents of the package are defined in:
`src/examples/modules/helloworld_01/package/prototype`.

Prototype Entries

Here are the entries in the prototype file and a description of their functions:

- `i pkginfo: pkginfo(4)` is an ASCII file that describes the characteristics of the package along with information that helps control the flow of installation. It is created by the software package developer. In this example, the `pkginfo` is expected to reside in the same directory as the prototype file.
- `i copyright=install/copyright` is the copyright that is displayed when the package is being installed. In this example, the copyright is expected to reside in the install subdirectory. Note in the examples that the format for the prototype entries is *<component destination >=<component source>*. Component source being the location where the source can be found and component destination being the name and location of where the file will reside. Note that the names of the component source and destination may be different.
- `i depend=install/depend: depend(4)` is an ASCII file used to specify information concerning software dependencies for a particular package. The file is created by a software developer. The `helloworld_01` package depends on the `SUNWesagt` Sun Management Center agent package at runtime.
- `!default 0755 root sys` is a packaging directive to assign the components following this statement to the file attributes of read/write/execute-read/execute-read/execute, ownership equals root, and group equals sys.
- `d none SUNWsymon ? ? ?` specifies the directories that need to be created during package installation to contain the components. The first field, the file type field, of the entry '`d`', specifies that this component is a directory. The second field, the class field, 'none' specifies that this entry belongs to the package class '`none`'. The third field is the pathname of the component, in this case

'SUNWsymon'. The fields '? ? ?' are used when you know that this component has been installed be a another package and that you want this component to have the same attributes assigned to it.

- `!default 0444 root sys` is a packaging directive to assign the components following this statement to the file attributes of read-only-read-only-read-only, ownership equals root, and group equals sys.
- `f none SUNWsymon/modules/cfg/helloworld-version01-d.x=helloworld-version01-d.x` has 'f' as the first field, which denotes that the component is a 'file'. The second field denotes that this component belong to the class 'none'. The third fields denotes that destination for this component will be `SUNWsymon/modules/cfg/helloworld-version01-d.x` and that the source to this component can be found at `helloworld-version01-d.x`. Remember at the beginning of this section, that the '`-r $(COMPONENT_ROOT)`' option to `pkgmk`, allows the `pkgmk` utility to begin finding the source component `helloworld-version01-d.x` in the directory '`..`'.

Sun Management Center Software Packaging Practices

Package Naming

The Sun Management Center team uses SUNWes to denote that this package is a Sun Microsystems (SUNW) package and belongs to the Enterprise software group of packages. The next three characters are used to identify the individual packages of Sun Management Center software.

Package Versioning

The Sun Management Center team uses the VERSION, REVISION macros with the following form for Solaris software dependent packages (OS equals Solaris Release, note that 2.5.1 would be 2.5:, that is, Major.Minor, not Major.Minor.Micro):

```
VERSION=[product release version], REV=[OS Major.Minor].[YYYY.MM.DD]
```

For packages that are not Solaris Release specific (meaning the package is supported on all Solaris Sun Management Center supported releases, the [OS `Major.Minor`] string must be out of the REV string.

Component Naming

All components must have a unique name to avoid component collision at install time.

Package Dependencies

When installing modules, `SUNWesagt` is the suggested package dependency. When installing a console bean, `SUNWessrv` is the suggested package dependency.

Prototype File

The Sun Management Center team uses explicit entries in their packaging to facilitate clarity for developers when maintaining the prototype files.

Sun Management Center Module Name Practices

Sun Management Center modules are installed in a specific directory. So that there are no conflicts with the modules developed by other users, you need to ensure the uniqueness of your module filenames. It is suggested that you use the registry that is setup by Sun Management Center. Please visit the following website for more information:

```
http://www.sun.com/sunmanagementcenter/
```

Note – The above discussion is applicable for the console help files also.

CHAPTER 18

Troubleshooting

This section includes information on troubleshooting. Items covered include information on:

Module

During the Sun Management Center module development process, when you encounter problems with module loading and its additional functionalities, refer to one of the three areas where Sun Management Center software provides you with trouble shooting information.

- On the console—While loading a module from console, Sun Management Center software provides information about the status of the loading operation.
- In the agent log file—The agent log file provides information regarding module loading and module operation. The agent log file is a circular log file and can be viewed by entering the following command:

```
/opt/SUNWsymon/sbin/es-run ctail -f /var/opt/SUNWsymon/log/agent.log
```

- When in the Interactive Agent mode—Starting the agent in the interactive mode allows you to troubleshoot module loading. To run the agent in interactive mode, enter the following command:

```
/opt/SUNWsymon/es-start -ai
```

The following sections provide some examples for each one of the above categories.

Console Messages

TABLE 18-1 Example Error Messages that Display on the Console

Problem	Troubleshooting Information
Loading at the wrong place.	Check the ‘ `param:moduleType = '` value in <module>-m.x file.
Error Message: Module load failed.	Check to see if the following conditions exist: • If the agent file is under `/opt/SUNWsymon/modules/cfg` • If the models file is under `/opt/SUNWsymon/modules/cfg` • For any syntax error in module files • For valid syntax and datatypes in the models file • If the library files exist

Agent Log File Messages

TABLE 18-2 Example Error Messages That Are Found in the Agent Log File

Error Messages	Troubleshooting Information
Import interface failed	Check to see if the agent file is under `/opt/SUNWsymon/modules/cfg`.
Shutting down subagents parsing error in file: `/ /localhost/<module>-d.x flags=ro(1)`: failed to open file. aborting execution	Check to see if the models file is under `/opt/SUNWsymon/modules/cfg`.
Syntax error in file: `//localhost/<module-file>` flags=ro(42) at token '}' aborting execution	Check for syntax error in <*module-file*> around line# 42.
Parsing error in file: `//localhost/<module>-models-d.x flags=ro(17)inherit`: could not inherit ASDF. aborting execution	Check the datatype ASDF in models file.
Shutting down subagents, general parsing error, file: `//localhost/helloworld-version02-d.x flags=ro 10 couldn't load file pkgdemohw2.so":ld.so.1:esd:fatal:lib demohw2.so.1:open` failed: No such file or directory"], aborting execution	Check to see if the of library files exist.

Interactive Agent Mode Messages

TABLE 18-3 Example Error Messages Provided by the Interactive Agent

Error Messages	Troubleshooting Information
Parsing error in file: `//localhost/<module>-d.x flags=ro(1)`: failed to open file aborting execution	Check to see if the models file is under `/opt/SUNWsymon/modules/cfg`.
Syntax error in file: `//localhost/<module-file> flags=ro(42) at token '}'` aborting execution	Check for syntax error in <*module-file*> around line# 42.
Parsing error in file: `//localhost/<module>-models-d.x flags=ro(17)`: inherit: could not inherit ASDF aborting execution	Check the datatype ASDF in models file.
General parsing error `file://localhost/helloworld-version02-d.x` flags=ro 10 couldn't load file pkgdemohw2.so": .so.1: esd: fatal: libdemohw2.so.1: open failed: No such file or directory"] aborting execution	Check to see if the library file exists.

Console

The Sun Management Center console is based on a configuration file infrastructure. This infrastructure provides a scripting language that is interpreted at run time to create Java consoles. Because of this late binding, most of the errors are shown at run time.

Errors are displayed in two places:

- Inside the terminal window in which the Sun Management Center console is started: If you have made any syntax errors in your configuration file definition for your application, those errors will show up in this terminal window.
- Inside the Sun Management Center Console Messages dialog which is invoked from the file `pulldownmenu` in the main console: The error messages that are shown here are run time errors. For example, as given in the Task List, if you mention a wrong path for your `awx:component` bean, you will get an error that indicates this class was not found. It is advisable to keep the Sun Management Center Console Messages dialog up while you are doing development. You can

also look at the Sun Management Center server log to see if the communication between the Sun Management Center console and Sun Management Center server is going through.

Note – All configuration files (those with extension `.x`) are installed on the Sun Management Center server, hence any action on these files will go through that server.

For example, if you are using a [`load myConsole-j.x`] construct in your application, then in the Sun Management Center server log you should see this file being read by the console.

The Sun Management Center server log is a circular text file. To look at it in a 'tail' mode run following command:

```
/opt/SUNWsymon/sbin/es-run ctail -f /var/opt/SUNWsymon/log/server.log
```

No console log is created by the Sun Management Center console. All console debug messages are displayed in the Sun Management Center Console Messages dialog. When the Sun Management Center console comes up, it redirects all `stdout` messages to this dialog. Thus, if your Java code has `System.out.println` statements, the output of those will be displayed in this dialog.

APPENDIX **A**

Modules Appendix

This chapter covers the following topics:

Module Building Environment

This section covers the following topics:

Agent Development

The Sun Management Center agent is based on Tcl and TOE technologies. This section provides background information about the development environment of the Sun Management Center agent.

Tcl Environment

Tcl (Tool Command Language) is an interpreted command-oriented language that can be used to connect building blocks built in system programming languages like C. Commands can be added to the interpreter using a clean C interface, and these commands co-exist with built-in Tcl commands.

Tcl has both simple variables and associative arrays, and all values (including procedure bodies) are represented as strings.

For more information about the Tcl language, refer to *Tcl and Tk Toolkit*.

TOE Environment

The Tcl Object Extension (TOE) is a simple modification to the Tcl language that provides an object-oriented environment that supports a rich set of object-oriented (OO) features, and that is backward compatible with conventional Tcl code.

The premise behind the TOE modifications is simple. It was observed that all Tcl hash table access is channelled through two C macros, one to create hash entries and one to locate them.

Using this knowledge, these macros were overridden to call a set of recursive hash table operators that are capable of locating commands or data in a more sophisticated manner. This twisting of the hash table operators can be done with a one-line modification to the Tcl source code and is completely transparent to all users of these functions.

Using the modified hash table behavior, an object system was built that capitalizes on this new hash table scoping algorithm. A simple data structure, known as a TOE object, was created that is simply a pair of hash tables (one for commands, one for data) and a set of pointers to other TOE objects. The hash tables store procedures and data (properties) local to that object, while the pointers reference parent objects. Parent objects can be recursed to locate commands or data not found in the local hash tables.

To complete the system, a pointer to the current TOE object is placed in the global command hash table of the interpreter. When a command is executed, the Tcl system uses the low level Tcl hash operators to find the body of the command. These modified operators detect an active TOE context, and delegate the hash lookup to the hash tables of the current TOE object. Failure to locate the target key in that object triggers recursion into each of the parent pointers until the key is hit or all ancestors have been searched.

This transparent recursion makes all hash entries in all parents of an object appear to be local to that object. This behavior corresponds to inheritance in an object oriented environment. Other key object oriented features, such as polymorphism and dynamic binding, also fall out of the design, as the function performed by a procedure depends entirely on the object in which it was invoked.

TOE Objects

A TOE object is a data structure consisting of a command hash table, a dictionary hash table (for object property storage), parent object pointers (for ancestral relationships) and a superior object pointer (for structural relationships).

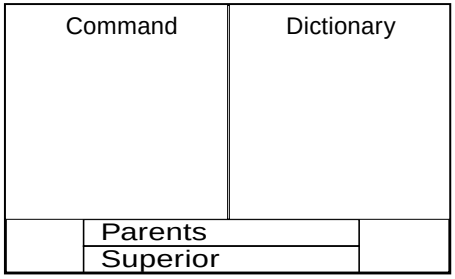

FIGURE A-1 TOE Object

Because a TOE object contains its own command and dictionary hash tables, objects can support their own command vocabulary and properties. The command names are local to the object, so commands bearing the same name can coexist in different objects. The dictionary properties are independent of the Tcl variable system, so variable use need not alter or conflict with object properties.

Object Relationships

The TOE system supports ancestral and structural object relationships.

Ancestral Relationships

These relationships define the parent/child relationships of objects. This defines the object-oriented inheritance characteristics of an object, with the child object inheriting commands and data from the parent object.

FIGURE A-2 Simple Parent/Child Object Relationship

In this relationship, child objects can see all of the commands and dictionary data in the parent object, that is implemented by referencing the parent objects on every hash table lookup. This parental referencing becomes a parent tree traversal if the parents themselves have parents.

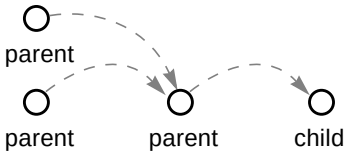

FIGURE A-3 Multiparent/Child Object Relationships

Structural Relationships

These relationships define the superior/inferior relationships of objects in a tree structure. Objects can be organized into tree structures where each object has a superior (the object up the tree) and zero or more inferiors (the objects down the tree).

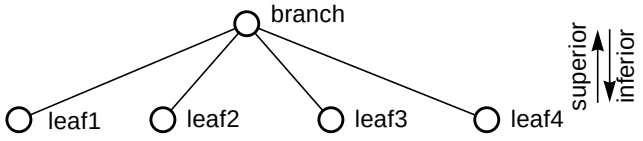

FIGURE A-4 Superior and Inferior Object Relationship

Combining Ancestral and Structural Relationships

By independently supporting these two types of relationships, trees of objects can be constructed where the structural aspects of the tree (defined by the overall purpose of the objects) is independent of the inheritance of the nodes in the tree (defined by the functions performed by the individual objects).

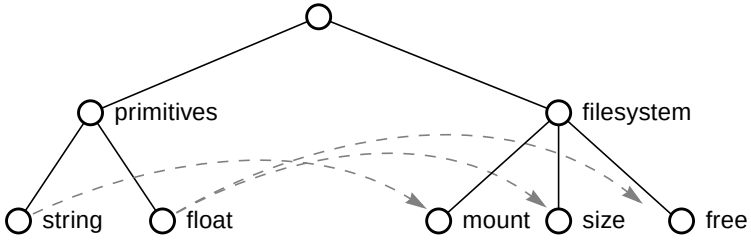

FIGURE A-5 Object Relationships of Filesystem Example

In this example, the structure of the tree is related to the overall purpose of the objects (in this case, a model of a file system), while the ancestry of each object determines what the objects do and how it behaves (in this case, the primitive data types the object represents).

Object Property Dictionaries

Every TOE object contains a set of properties. In the TOE environment, object properties are stored in a dictionary. Each object contains a dictionary that stores properties relevant to that object instance. In the implementation of TOE, a dictionary is a hash table that stores information using logical keys.

Dictionary Keys

The TOE object dictionaries use a two-key paradigm, where two logical names are used to reference any one data entity. This allows dictionaries to be partitioned into separate sections, with the division being based on the purpose, source, or volatility of the data being stored. These dictionary partitions are referred to as *slices* in the TOE system, and the pieces of data within each slice are named using what is referred to as the *dictionary key.* Dictionary slices can be thought of as property classes when used to configure object instances.

TABLE A-1 Dictionary Example

Slice	Key	Value
`value`	`refreshCommand`	`"df -kF ufs"`
`value`	`refreshInterval`	`"60"`
`alarmlimit`	`warning`	`"10000"`
`alarmlimit`	`error`	`"5000"`
`data`	`1`	`"95000"`

The object's dictionary has three partitions or slices:

- Value
- Alarmlimit
- Data

Value

The value slice contains configuration information. In this case it is the refresh command and interval of the file system entity.

Alarmlimit

The alarmlimit slice contains the error and warning level alarm limits.

Data

The data slice contains the dynamic data of the object, in this case the current floating point value of the managed property, free.

This is a typical example of data partitioning using slices, where the slices are based on the purposes and sources of the dictionary entries and are directly related to the classes of properties used by an object instance.

The dictionaries define certain operations that can be performed on entire slices. These operations include the ability to list all the currently defined keys in a slice and to undefine an entire slice. Hence maintenance of dictionary keys is simplified if the slices are properly configured, and a certain amount of accountability can be achieved if the dictionaries are partitioned along functional boundaries.

Importing and Exporting Dictionaries (Module Configuration Files)

The TOE object dictionaries have the inherent ability to import and export themselves as formatted text. The format of this representation is referred to as the .x file format. In this format, the slices and keys of a dictionary are represented using a well-defined, unambiguous syntax.

Dictionary Entry (Property) Representation

Dictionary entries can be described using the following syntax:

```
[slice:]key = value
```

Using this syntax, the dictionary entries in the preceding example table can be represented as:

```
value:refreshCommand = "df -kFufs"
value:refreshInterval = "60"
alarmlimit:warning = "10000"
alarmlimit:error = "5000"
data:1 = 95000
```

In this example, both of the slices of the object's dictionary were exported together, and all keys are prefixed by their slice name. In actuality, slices can be exported and imported individually, and if there is only one slice present, the slice prefix is optional. This can be thought of as slice relative, since the keys are placed in whatever slice is specified at the time of import. For example, the data slice of the dictionary can be exported slice relative as follows:

```
warning = "10000"
error = "5000"
```

Multi-object Dictionary Representation

The dictionaries of many objects can be exported or imported in a single operation. In such operations, the tree structure of the objects is maintained in the .x file output. The .x file syntax for an object is as follows:

```
object1 = {
key1 = "value 1"
key2 = "value 2"
}
```

In this notation, the opening of the curly brace indicates that the key-value pairs to follow belong to the object named *object1*. Such a representation is generated if an export is performed from the superior object of the *object1* object. This hierarchical

representation can be nested as deep as the object tree, supporting arbitrarily nested .x file representations. The following is an example of an .x file representation that is two levels deep:

```
object1 = {
key1 = "value 1"
key2 = "value 2"
object2 = {
key3 = "value 3"
key4 = "value 4"
}
}
```

Action Specifications

The .x file format supports the specification of actions, or logical operations, to be performed during initialization on objects described in the object tree. The general form of an action is:

```
[ action args ... ]
```

This syntax is simply a set of square braces enclosing the action command line, and optional arguments can be specified. The actual actions supported by the .x file parser depends on the application using the object tree, but several actions are always valid, such as:

- Inherit
- Load
- Source

Inherit—Adds the named object(s) to the object's parent list, thus altering the ancestral relationships of the current object. This action is the primary way of creating parent and child relationships within trees that are specified using module configuration files.

```
mount = { [ inherit primitives.string ] ... }
```

Load—Loads the named .x file into the current object. This is the primary mechanism for combining multiple module configuration files into a single object tree. In the following example, the .x file named `primitives.x` is loaded into the `primitives` object.

```
primitives = { [ load primitives.x ] }
```

Source—Loads and executes a Tcl/TOE source file into the current object. This is the primary means of extending and overriding an object's command set from an .x file. In the example, a Tcl/TOE file named `primitives.prc` is loaded and executed into the `proc` object.

```
proc = { [ source primitives.prc ] }
```

By using the nested nature of module configuration files and the *inherit* action, both the ancestral and structural aspects of an object tree can be represented.

The following .x file can be used to describe the file system subtree in FIGURE A-5:

```
filesystem = {
mount = {
[ inherit primitives.string ]
}
size = {
[ inherit primitives.float ]
}
free = {
[ inherit primitives.float ]
}
}
```

FIGURE A-6 .x file Syntax for Filesystem Example

TOE Object Classes

TOE object classes are the primary mechanism employed to extend the command vocabulary of TOE objects. TOE object classes encapsulate a set of commands that provide a well defined function. TOE objects can then inherit these classes to gain the desired functionality of the command set.

Examples of TOE object classes used by the Sun Management Center agent include the MIB node class and the SNMP class. The MIB node class enables TOE objects to gather and store data periodically and perform alarm checks on the data. The SNMP class encapsulates SNMP communication capabilities.

Agent Framework

The agent framework consists of a single tree structure within the agent that contains global services, configuration data, classes and templates that can be used by any object within the agent.

The following is a general structure of an agent's TOE object tree:

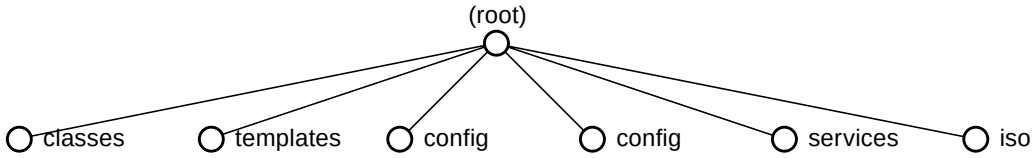

FIGURE A-7 TOE Object Tree Structure of Agent

The agent framework provides the core agent services and functions that include SNMP communications, command execution, and module management.

This framework exists to support the realization of managed objects, properties and other modeling elements that perform the actual monitoring and management functions of the agent. The managed objects, properties, and other modeling elements are encapsulated in management modules and are also loaded in this tree.

Shell Service

The shell service object (`.services.io.sh`) provides a mechanism for the Sun Management Center agent to execute commands (scripts and programs) and obtain the results of the command. This service is commonly used by module MIB nodes for data acquisition and for executing alarm actions.

The shell service supports the queuing of commands to be executed. It also supports the spawning of multiple shells to allow commands to be executed in parallel.

This service involves the agent opening pipes to one or more captive Bourne shell processes. The maximum number of shells to run is configurable.

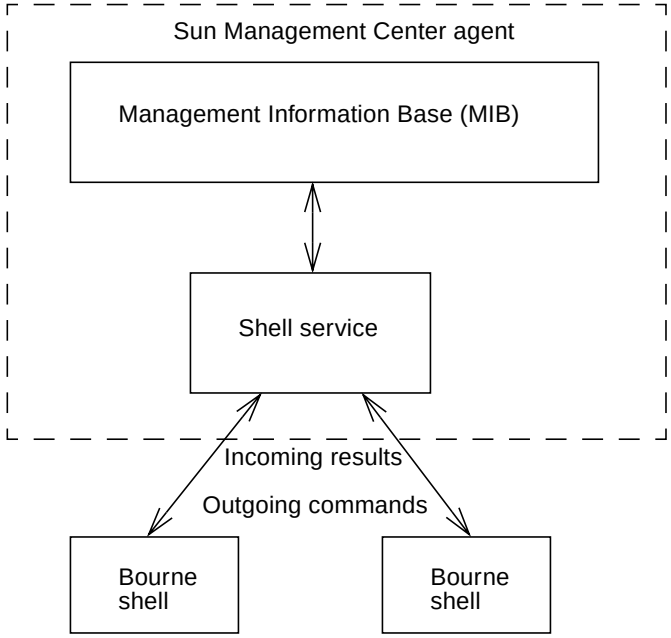

FIGURE A-8 Shell Service Data Flow

Shell Service Result Handling

When interfacing with the shell services, the caller specifies the shell command and the callback to process the command results.

The shell command to be executed can be specified with or without a full path. If the command is not specified with a full path, the command is searched for in the directories specified by the PATH environment variable of the agent.

The callback specification is comprised of a TOE object identifier and a callback method. The TOE ID specifies the TOE object context in which the callback method should be executed. The callback method must be specified with the *%result* argument (for example, callbackMethod %result) that is substituted with a result specification every time the callback is invoked.

The result specification returned to the callback is in the form of a three element list comprised of a return code, a transaction identifier, and corresponding data. The possible results are as follows:

- *wait <tid> ""* indicates that the command is being executed asynchronously and that the final result is pending.

- *data <tid> <command result>* indicates successful execution of the command and the command results are included as the third element in the list.
- *error <tid> <error message>* indicates that the execution of the command resulted in an error (that is, program does not exist or the program wrote to STDERR instead of STDOUT). The error message is included as the third element in the list.

Shell Protocol

A very simple shell protocol defines the interaction between the agent and the shell.

For each command to be executed, the agent sends the command to be executed to the shell, followed by `echo EOT`, where EOT is the terminating character. The shell executes the commands so that the command result is returned followed by EOT. The reception of the terminating character indicates the end of the transaction, implying that the next command can be sent to the shell.

Ping Service

The icmp object (`.services.io.icmp`) enables the Sun Management Center agent to ping hosts to determine whether they are up or down. Ping uses the ICMP protocol `ECHO_REQUEST` datagram to elicit an `ICMP ECHO_RESPONSE` from the specified host. A host is assumed to be up if it responds. By default, a host is assumed to be down if it does not respond after three retries, each with a timeout of 10 seconds. The number of retries and the timeout can be overridden by specifying the `maxRetries` and `retryInterval` parameters, respectively, in the `.services.io.icmp` object.

The ping service is used by the SNMP interface to determine the status of a host whose agent does not respond to an SNMP request. This service is also used by the Topology module in the Topology agent when monitoring entities as IP-based devices.

Master Event Loop (MEL) Service

The `mel` object (`.services.mel`) provides timer services to other objects. It allows other nodes to register and cancel time based events.

Default I/O Service

The default service (`.services.io.default`) is a default shell service provided for general use by any object. However, in general, modules that require a shell service should specify their own shell service to guarantee the availability of its access to a shell service.

Data Logging Registry Service

This service (`.services.history`) maintains a table of all current data logging requests. This table is automatically updated whenever the data logging specifications of a managed property changes.

This table is queried by the data logging registry module using the *listRegistry* method. This module allows console users to view information about all managed properties whose values are currently being logged.

The logging information includes the following fields:

- *state*—state of destination log file
- *module name*—name of module in which the property whose data is being logged resides
- *instance name*—module instance name of module in which the property whose data is being logged resides
- *property name*—name of property whose value is being logged
- *log interval*—logging interval
- *file status*—flag indicating whether data logging to file is currently enabled (for example, on | off)
- *logURL*—interface specification of the destination log file
- *cache status*—flag indicating whether data is currently being updated in the internal history buffer
- *cache size*—current size of the internal history buffer

Note – The data logging registry service does not perform the actual addition or removal data logging requests; it maintains a table that reflects the current data logging requests.

The configuration of data logging is supported through shadow SNMP requests to the appropriate MIB node.

File Scanning Service

This service (`.services.fscan`) allows MIB objects to subscribe for file scanning services. Conceptually, MIB objects subscribe by specifying a filename, regular expression pattern, and a callback. The service incrementally scans the file for regular expression pattern and when the pattern is detected, the callback is called with the match results. When the MIB object is no longer interested in the scanning of the pattern, it can then perform an unsubscription request.

This service is used by MIB objects whose alarm check involves log rules.

Subscribing for Patterns

To subscribe for the detection of a pattern in a file, the *fsSubscribe* method is used:

```
fsSubscribe <filename> <pattern> <callback spec> ?<node template>?
```

where:

- *filename* is the name of the file to be scanned.
- *pattern* is the regular expression pattern to scan for.
- *callback spec* is a callback specification that is dependent on the node template. For the default node template (fscan-node-d), the callback spec is comprised of a three element list consisting of a TOE object id, a row name, and a rule identifier.
- *node template* is an optional specification that defines the type of object that must be instantiated to service the subscription request. The default node template is fscan-node-d, which assumes that the caller is a rule (that is, logSubscribe) and expects the callback specification to contain a TOE object id, a row name, and a rule identifier. Currently, no other node templates are defined.

If the subscription is successful, the TOE object ID of the file scanning node is returned. If the subscription fails, –1 is returned.

Unsubscribing Patterns

To remove an existing subscription, the fsUnsubscribe method can be used:

```
fsUnsubscribe <filename> <pattern> <callback>
```

Module Management

Module management is a fundamental function provided by the Sun Management Center agent framework. It enables the agent to load and unload the management modules that define the monitoring and management functions performed by the agent.

Modules comprise of a set of managed objects and properties that focus on a particular aspect of system or application condition and performance.

The discussion of module management in the Sun Management Center agent is divided into the following topics:

- MIB Subtrees: This section describes the structure of the trees in which management modules are loaded.
- Module Loading: This section describes the mechanisms used for loading and unloading modules.
- MIB Manager: This section describes additional functions provided by the agent to manage modules.

MIB Subtrees

The Sun Management Center agent supports SNMP contexts to identify MIB modules that can have multiple instances. Each SNMP context is represented by a separate MIB subtree.

Default SNMP Context

The *.iso* subtree represents the default SNMP context (all modules that can only be instantiated once they are loaded into this subtree). The standard MIB objects that are not part of modules are also loaded into this subtree.

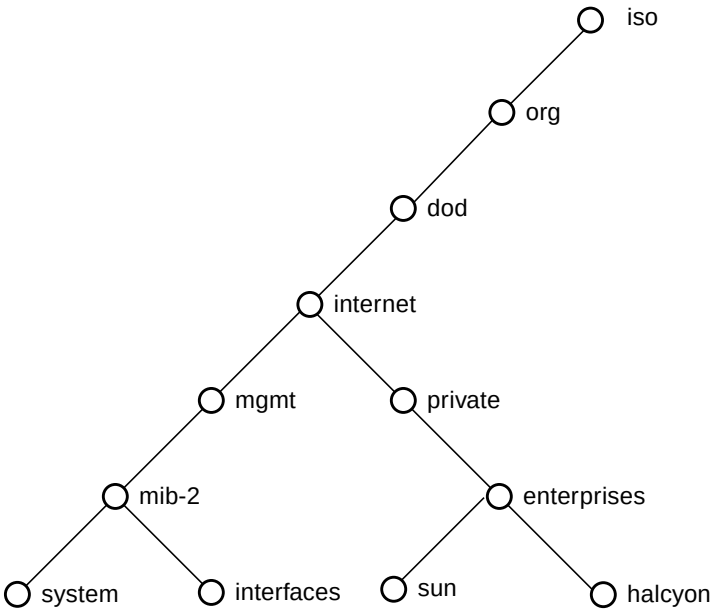

FIGURE A-9 Default Context—ISO subtree

In general, the *.iso* subtree for the default SNMP context contains two main branches, the standard management branch (*mgmt*) and the *private enterprises* branch.

The standard SNMP management MIB objects are loaded in the *mgmt* subtree. An example of a standard SNMP MIB is the MIB for Network Management of TCP/IP-based internets (MIB-II).

The *enterprises* branch contains enterprise specific subtrees.

For instance, the Sun Management Center agent always instantiates a core module loader in the `.iso*enterprises.sun.prod.sunsymon.agent.base.mibman` object in the default SNMP context. Sun Management Center modules that can only have a single instance are also loaded in under the *enterprises* branch in the default SNMP context.

Non-default SNMP Contexts

Each instance of a module that can be multi-instantiated is assigned an SNMP context. The name of the module instance corresponds to the SNMP context name. Each nondefault SNMP context is represented by a separate *<context name>*`.iso.*` subtree under the `.contexts` object.

For example, loading a Topology module whose instance name is *view-1* creates the `.contexts.view-1.iso.*` subtree that represents the *view-1* SNMP context.

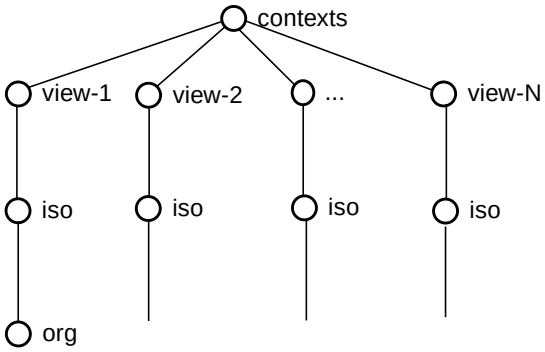

FIGURE A-10 Nondefault SNMP Contexts—Contexts Subtree

Private Enterprises

By convention, the Sun Management Center agent modules developed by Sun are loaded within the *sun* branch in the appropriate SNMP context. Similarly, Sun Management Center agent modules developed by Halcyon are loaded in a subtree under the appropriate SNMP context.

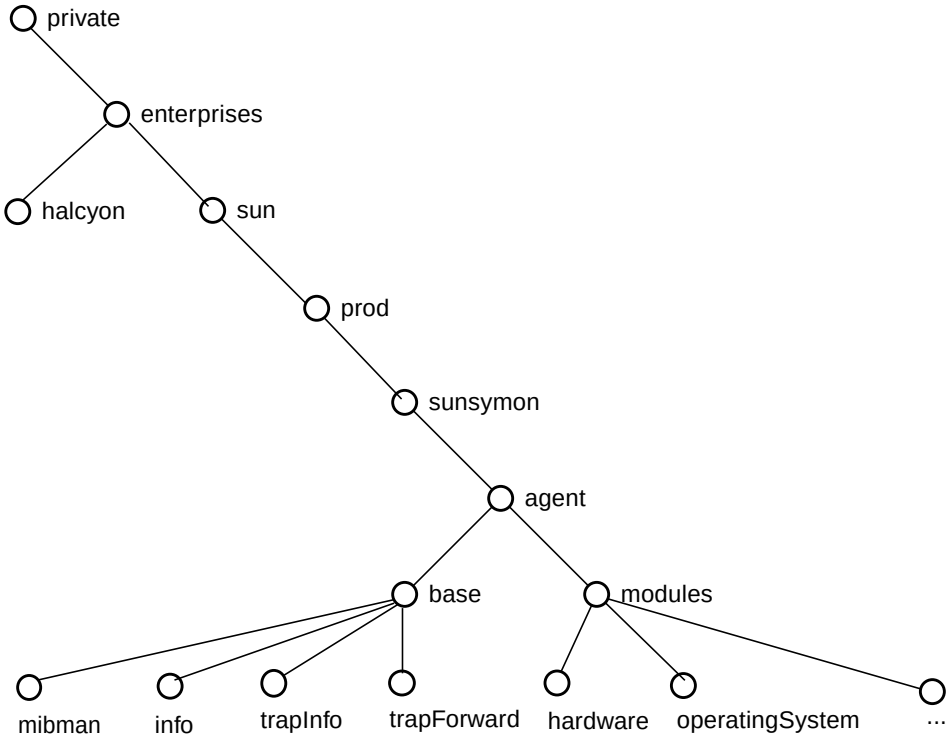

FIGURE A-11 Private Enterprise Subtree

The preferred location of Sun Management Center modules can be specified in an .x file (`base-oids-<enterprise>-d.dat`) that maps the logical object names to object identifiers. The Sun Management Center agent loads this file on start up. It can also be specified in the parameter file of the module.

The location where modules are loaded is important for hierarchical summarization and for general module management. Hierarchical summarization groups the alarm statuses of all managed child objects to generate an overall status of the managed objects for that portion of the MIB tree. Organizing modules into groups allows modules to be managed as a group.

Module Subtrees

Sun Management Center specific modules loaded by the agent are classified into the following module types:

operatingSystem—monitor operating system related entities associated with the local host system (for example, CPU usage, swap, processes, file systems, and so forth.)

- *hardware*—modules that monitor hardware related entities associated with the local host system (for example, disks, CPUs, power supplies, and so forth)
- *localApplication*—modules that monitor software applications that run on the local host (for example, custom software applications)
- *remoteSystem*—modules that monitor entities running on other host systems (for example, legacy SNMP agents on remote hosts)
- *serverSupport*—modules that perform server agent functions and are not intended to be accessed using the standard MIB browsing mechanisms (for example, domain-control, topology, cfgserver, and so forth)

Each Sun Management Center agent module is loaded into its corresponding module type branch under the appropriate *modules* subtree and SNMP context. The following diagram shows a *module* subtree.

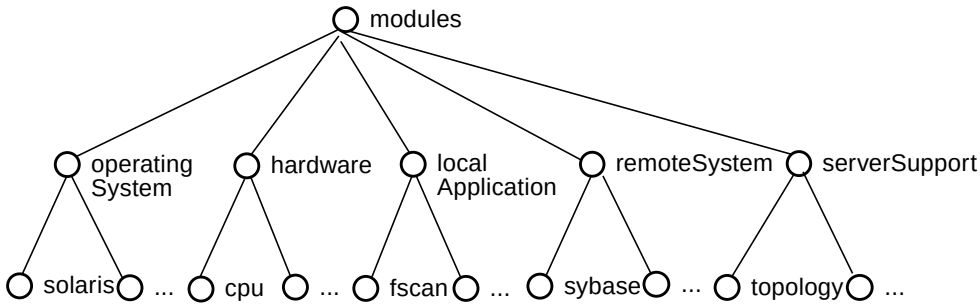

FIGURE A-12 Modules Subtree

Classifying modules by these categories is important for hierarchical summarization. This classification of modules separates the various alarm summary lists, enabling the alarm summary of managed objects in the MIB to reflect the status of the respective category.

Module Loading

When a Sun Management Center agent starts up, the agent loads the set of modules specified in its module configuration file (`base-modules-d.dat`). Once the agent is running, a Sun Management Center console user can load additional modules or unload loaded modules. The loaded modules are saved to the module configuration file (that is, `/var/opt/SUNWsymon/cfg/base-modules-d.dat`) so that the same set of modules is automatically reloaded if the agent is restarted.

The module configuration file contains entries for each module to be loaded. For each module to be loaded, its location in the MIB tree hierarchy, name, and parameters must be specified. Each entry in the file has the following format:

```
<module spec> = "<MIB location> <enterprise> <module name> <module parameters>"
```

where:

module spec specifies the module name and module instance name (if one exists) concatenated with a + sign (for example, `fscan+syslog, mib2-system`).

MIB location specifies the full TOE object path to the root node of the module. For example, the mib2 system module location is:

```
.iso.org.dod.internet.mgmt.mib-2.system.
```

enterprise specifies the name of the enterprise MIB that the module resides in. For example, a module developed by Sun should reside in the sun enterprise. A module that is not enterprise specific (for example, mib2-system) should specify a blank enterprise.

module name specifies the actual name of the module without the module instance specification (for example, `mib2-system`, `fscan`).

module parameters specifies the module parameters in the form of a list containing key-value pairs terminated by semi-colons (that is, ';'). All string values with white-spaces should be enclosed with backslashed double quotes (that is, "

"\aaa bbb\"). For example, to specify module parameters *a* and *b* whose values are *123* and "*1 2 3*", respectively; use the following specification: {*a = 123; b = \"1 2 3\";*}.

Module Parameters

The module parameters that can be specified correspond to those parameters specified in the module's parameter file (that is, <*module*>`-m.x`).

Common module parameters include:

- *module* specifies the module name.
- *moduleName* specifies the module name for display purposes.
- *version* specifies the module version.
- *location* specifies the MIB location of module.

- *enterprise* defines enterprise MIB in whcih the module resides.
- *moduleType* specifies module classification. Possible values are hardware, operatingSystem, localApplication, remoteSystem, or serverSupport.
- *desc* specifies a module description

For modules that can be instantiated multiple times, the *instance* and *instanceName* parameters should also be defined. In addition, modules can specify additional parameters that are specific for the module.

base-modules-d.dat

This file contains three module entries: `mib2-system`, `agent-stats`, and `fscan+syslog`. The `mib2-system` entry demonstrates the loading of a non-enterprise specific module. The `agent-stats` entry shows how to load a simple Sun Enterprise module. The `fscan+syslog` entry shows how to load a Sun Enterprise module that can be instantiated multiple times. This module also contains module specific parameters.

Note – Each entry must be specified on one line only. To improve readability, each entry has been divided into multiple lines in the following example.

In the following table, note that each row is a continious string of syntax.

CODE EXAMPLE A-1 base-modules-d.dat

```
#
#   File:    base-modules-d.dat
#
mib2-system =
".iso.org.dod.internet.mgmt.mib-2.system {} mib2-system
{module = \"mib2-system\"; moduleName = \"MIB2 System\"; version = \"1.0\";
console = \"mib2-system\"; location = \".iso.org.dod.internet.mgmt.mib-
2.system\"; enterprise = \"\"; moduleType = \"localApplication\"; instance =
\"\"; desc = \"The MIB2 System module monitors MIB2 system information.\"; }"
agent-stats =
".iso.org.dod.internet.private.enterprises.sun.prod.sunsymon.agent.modules.age
ntStats sun agent-stats {module = \"agent-stats\"; moduleName = \"Agent
Statistics\"; version = \"2.0\"; console = \"agent-stats\"; location =
\".iso.org.dod.internet.private.enterprises.sun.prod.sunsymon.agent.modules.ag
entStats\"; enterprise = \"sun\"; moduleType = \"localApplication\"; instance =
\"\"; desc = \"The Agent Statistics module monitors the health of the agent
installed on the local host.\"; }"
fscan+syslog =
".contexts.syslog.iso.org.dod.internet.private.enterprises.sun.prod.sunsymon.a
gent.modules.fscan sun fscan {module = \"fscan\"; moduleName = \"File
Scanning\"; version = \"2.0\"; console = \"fscan\"; location =
\".iso.org.dod.internet.private.enterprises.sun.prod.sunsymon.agent.modules.fs
can\"; enterprise = \"sun\"; moduleType = \"localApplication\"; instance =
\"syslog\"; instanceName = \"System Log\"; filename = \"/var/adm/messages\";
scanmode = \"tail\"; desc = \"The File Scanning module monitors files for regular
expressions.\"; }
```

MIB Manager

The MIB manager provides general MIB related services to external entities through SNMP. Sun Management Center agents instantiate the MIB manager in the `.iso*enterprise.sun.prod.sunsymon.agent.base.mibman` object.

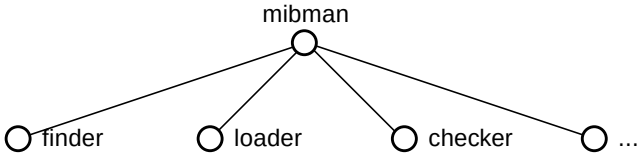

FIGURE A-13 MIB Manager Branch

The MIB manager is comprised of MIB objects that provide the following services:

- URL / OID finder
- Module loader
- Module checker
- Browser root
- Module registry
- Module tables

A procedures (that is, `_procedures`) TOE object also exists as a peer object of the MIB objects listed above. This object is not a MIB node object and only serves as a repository for MIB manager related procedures that can be inherited by the MIB nodes that need to execute the procedures.

URL/OID Finder

The `finder` object is used to resolve the SNMP URL of a currently loaded MIB object to its object identifier (OID).

When an SNMP URL is set into the finder object, the finder object locates the MIB object identified by the URL and returns its OID in the form of an OID URL.

The OID URL has the following general format;

```
snmp://<host>:<port>/oid/[<context>]/<oids>[/<subid>][?<shadow spec>]#<instance spec>
```

Subsequently, the OID can be determined from the OID URL and used to access directly the MIB object identified by the SNMP URL.

▼ To Convert an OID URL to an Actual OID

1. **Parse off the OID portion of the URL.**
2. **Extract the context if one is specified.**
3. **If the OID includes a shadow specification, extract it.**
4. **If the instance spec is a non-integer, it can be comprised of one or more comma separated instance data types (`int`, `ip`, `str`, `+str`, `oid`, or `+oid`).**

   These data types define how to convert the textual instance to a numeric instance. The '+' indicates that the actual length of the instance must be prepended to the instance since its length is not implied. The values of `int`, `ip`, and `oid` instance types are integers and so these values map directly to the subid values. The `str`

instance types indicate that the instance values are alphanumeric and must be converted to their corresponding decimal ASCII value and concatenated with a period (.) (for example, abc --> 97.98.99).

5. **If it is a shadow OID, append the instance length and append the shadow specification.**

6. **Replace all (/), (#), and (?) characters with a period (.).**

   For example, the SNMP URL for the system description property in the mib-2 system module is:

```
snmp://<host>:<port>/mod/mib2-system/sysDescr#0
```

   When this value is set to the finder node, the resulting response is the OID URL:

```
snmp://<host>:<port>/oid/1.3.6.1.2.1.1/1#0
```

   The actual OID can be extracted from the OID URL as follows:

   a. **Parse off the portion after the** `/oid/ substring` **(that is, 1.3.6.1.2.1.1/1#0).**

   b. **Substitute all '/' and '#'characters with '.' (that is, 1.3.6.1.2.1.1.1.0).**

      This OID can then be used to access the data via SNMP.

▼ To Access the fulldes Shadow Attribute of the Same MIB Property

- **Set the following URL to the finder:**

```
snmp://<host>:<port>/mod/mib2-system/sysDescr?fulldesc#0
```

The resulting OID URL is:

```
snmp://<host>:<port>/oid/2.3.6.1.2.1.1/1?7.1#0
```

▼ To Convert the Shadow OID URL to a Valid OID

The OID URL for a shadow OID contains a '?' that signifies the start of the shadow attribute index specification. The '#' signifies the start of the instance specification. To convert the shadow OID URL to a valid OID, do the following:

1. **Parse off the portion after** `/oid/` **(for example, 2.3.6.1.2.1.1/1?7.1#0).**

2. **From the parsed string, extract the shadow index specification that is enclosed by '?' and '#' and replace the '/' and '?' with a '.' (that is, shadow index specification is 7.1 and OID is 2.3.6.1.2.1.1.1#0).**

3. **Since the instance is an integer, simply append the length of the instance specification to the OID and replace the # with '.' since instance is '0', length is 1 -- 2.3.6.1.2.1.1.1.0.1.**

4. **Append the shadow index specification to the OID (2.3.6.1.2.1.1.1.0.1.7.1).**

   This OID can then be used to get the full description shadow attribute for the mib-2 system description property.

▼ To Access a Table Property in a Module

An example to get the scan pattern for a specific row (`unix_error` row instance) in the file scanning module (syslog module instance):

- **Send the following SNMP URL to the finder:**

```
snmp://<host>:<port>/mod/fscan+syslog/fscanstats/scanTable/
scanEntry/pattern#unix_error
```

The resulting OID URL is:

```
snmp://<host>:<port>/oid/syslog/1.3.6.1.4.1.42.2.12.2.2.24/1/3/1/
4#+str
```

▼ To Convert the OID URL to an OID

1. **Parse off the OID portion (that is, syslog/1.3.6.1.4.1.42.2.12.2.2.24/1/3/1/4#+str).**

2. **Extract the context (that is, syslog).**

3. **Since the instance specification is +str, the textual instance name must be converted to a numeric instance with the length prepended (`unix_error` --> 10.117.110.105.120.95.101.114.114.111.114).**

4. **Append the instance to the OID and replace the '/' and '#' with '.' (1.3.6.1.4.1.42.2.12.2.2.24.1.3.1.4.10.117.110.105.120.95.101.114.114.111.114).**

   This OID can then be used to request the data via SNMP. If using SNMPv2c or SNMPv2u, specify the context in the contextName field of the SNMP PDU. If using SNMPv1, specify the context name in the community field as <*community*>:<*context*> (for example, if the community name is `public` and the context is `syslog`, use `public:syslog` as the community field).

Module Loader

The *loader* MIB object is a leaf node that permits modules to be loaded by SNMP. Only users with sufficient security privileges are permitted to load modules (refer to the *Sun Management Center Security SDS* for more details about SNMP security).

The module loader input specifies the module parameters as key-value pairs separated by ';'. These parameters are based on the same information specified in the module configuration file described earlier.

For example, to load the mib-2 system module, the following string can be set to the loader node.

```
module = mib2-system; moduleName = "MIB2 System"; version = 1.0;
console = mib2-system; location = .iso.org.dod.internet.mgmt.mib-
2.system; enterprise = ""; moduleType = localApplication; instance
= ""; desc = "The MIB2 System module monitors MIB2 system
information.";
```

Module Checker

The *checker* MIB object is a leaf node that provides an SNMP interface for checking the status of a module. Given a module name and an optional module instance, it determines whether the module is currently loaded, not loaded, or not installed on the agent machine.

The following responses can be returned by the checker node:

- *notInstalled* string is returned if the set value corresponds to a nonexistent module name (for example, *bogus* where there is no module named 'bogus')
- *installed* string is returned under the following conditions: if the set value corresponds to an existing module name and that module is not currently loaded; if the set value is only the module name of a module that can be instantiated multiple times (for example, fscan without the instance specification); or if the set value is a module name + instance (for example, `fscan+bogus` where the bogus instance of the fscan module is not loaded) and the specified module instance is not loaded
- *loaded* string is returned under the following conditions: if the set value corresponds to a module name that is currently loaded and that module can only be instantiated once; or if the set value of a module name and (+) instance name corresponds to a loaded module with the specified instance (for example, `fscan+syslog` where the syslog instance of the fscan module is loaded)

Browser Root

The browser root MIB object is a leaf node whose value can be retrieved via SNMP. The value of the node is an SNMP URL that represents the root object of the MIB hierarchy tree. This value is used by the Sun Management Center console to determine the root of the MIB hierarchy of an agent's MIB for browsing purposes.

The default browser root URL is:

```
snmp://<host>:<port>/sym/base/mibman/modules
```

Module Registry

This MIB object is a leaf node that supports the retrieval of information about modules that are currently loaded by the agent via SNMP.

Specifically, by setting a module name to this MIB node, it returns the module name, module version, and number of loaded instances of the specified module. For example, setting the value *fscan* must return `fscan 2.0 1` where fscan is the module name, 2.0 is the version, and 1 is the number of loaded instances.

Alternatively, by setting a blank value to the MIB node, the module name, module version, and number of loaded instances for all the modules currently loaded are returned as a list of sublists ({`fscan 2.0 1`} {`mib2-system 2.0 1`}).

Module Tables

The modules object is branch MIB object that contains five module tables corresponding to the five module types: hardware, operatingSystem, localApplication, remoteSystem, and serverSupport. Each table contains the currently loaded modules, classified by their module type.

Each table contains the following columns:

- *module spec* specifies the module name + optional instance name.
- *name* specifies a description of the module.
- *i18nName* specifies a key used to lookup the internationalized description of the module.
- *version* specifies the module version.
- *URL* specifies an SNMP URL to get the overall status of the module.
- *status* specifies the current status of the module.
- *id* specifies the TOE ID of the module root (for internal use).

Additional Base MIB Branches

In addition to the *mibman* branch in the `.iso*base` subtree, every Sun Management Center agent component MIB contains the *info*, *trapInfo*, *trapForward*, and *control* branches. This section describes these MIB branches.

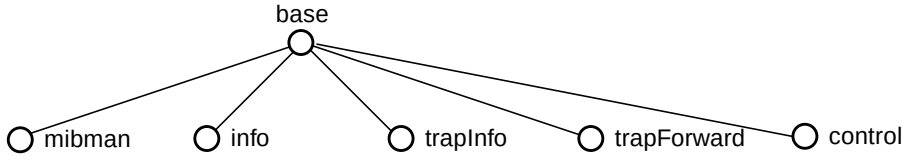

FIGURE A-14 .iso*base Subtree

System and Agent Information

The `.iso*base.info` branch contains nodes that provide general information about the host system, the agent, and modules installed on the system.

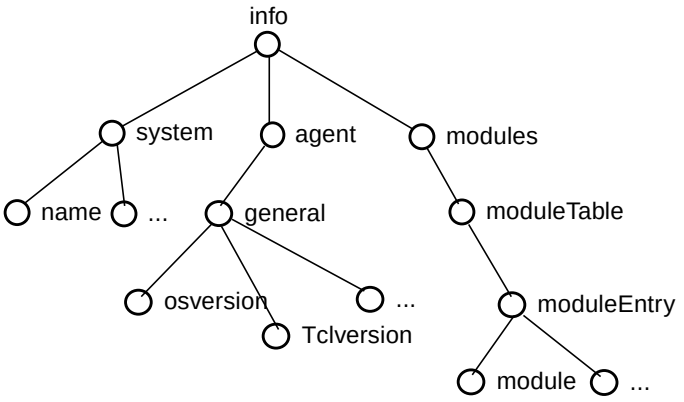

FIGURE A-15 info Branch

System Information

The system branch contains leaf nodes that provide the following information:

- Hostname
- System architecture
- Operating system version
- Hardware description

- IP address
- Trap destination
- Event destination

Agent Information

The agent branch contains leaf nodes that provide the following information:

- Software version
- Tcl version
- Tcl Patch Level

Module Information

The modules branch contains a table listing all the modules that can be loaded by the agent. The table contains the following columns:

- Module name
- Module description
- Internationalized module description
- Version
- Module count

The *module count* can be –1, 0, or some positive integer. A value of –1 implies that the module is currently loaded and cannot be instantiated multiple times. A value of 0 implies that the module is not currently loaded. A positive integer reflects the current number of loaded instances of the module and implies that the module can be loaded multiple times.

Trap Information

The `.iso*base.trapInfo` branch contains MIB objects whose values are included in the variable bindings of various enterprise specific traps that can be generated by the agent. The branch contains the following nodes:

- *statusOID* — included in variable binding of the *statusChange* trap. The value of the statusOID is set to the object identifier of the MIB node whose alarm status has changed.
- *refreshOID* — included in the variable binding of the *valueRefresh* trap. The value of the object is set to the object identifier of the node whose value has been manually refresh.

- *eventInfo* — included in the variable binding of *event* traps. The value of the object is set to a string containing the event version format, hostname, last line in the event circular log file, and the file creation time (for example, Tv0 manila 27 920442422).
- *userConfig* — included in the variable binding of *userConfig* traps. The value of the object is set to a string containing the agent's SNMP engine ID and usmUserSpinLock value.
- *moduleInfo* — included in the variable binding of the *moduleLoad* and *moduleUnload* traps. The value of the object is set to the module specification and module version (for example, fscan+syslog 2.0).

The *setTrapInfo* method is the primary interface for sending the *statusChange* and *valueRefresh* traps whose variable bindings include the *statusOID and refreshOID* objects, respectively. This method takes the trap type (statusOID or refreshOID) as an argument to specify the type of trap to send. The method must be called from the context of the MIB object whose OID must be included in the trap message variable binding.

The other traps have more specific functions and are intended to be used only by their respective users (eventInfo is used by event infrastructure, userConfig is used by usmUser MIB, and moduleInfo is used by the module load and unload methods).

Trap Forward

The `.iso*base.trapForward` branch contains nodes that support trap subscription. Specifically, this branch contains the following nodes:

- *clientRegistrar* supports trap subscription
- *jobAdder* supports incremental additions to an existing trap subscription
- *jobRemover* supports incremental deletions to an existing trap subscription

The subscription specifications supported by these nodes are described in appendix e??? xxx: xref.

Control Functions

The `.iso*base.control` branch object contains the *action* and *cache* leaf nodes. The set security access of these nodes are restricted to users with administrative security privileges.

Action Object

The *action* node supports the ability to shutdown the agent. To shutdown the agent, set the value to 2. The ability to restart the agent using this node is not supported in Sun Management Center 2.1.

Cache Object

The *cache* node is included in Sun Management Center 2.1 software. This node supports the ability to retrieve and manage the agent's current SNMP finder cache via SNMP. The current contents of the finder cache can be retrieved through a get request to the cache node.

Setting the cache node value to * clears all the entries in the finder cache. Setting the node value to a host name clears all the cache entries associated with the specified host. Setting the node value to a host name and port (`host:port`) clears all the cache entries associated with the specified host and port.

Useful Tcl Commands and Filters

The Tcl/TOE commands and procedures tha follow are available in all nodes for use as refresh commands or filters.

valueOf <*node name*>

This function takes the name of a managed property as its only argument and returns the value of the managed property. This function must be executed in the node that is the superior of <*node name*>.

getValue <*index*>

This function must be executed in a leaf node and returns the value stored for the specified <*index*>. If the leaf node is a scalar, <*index*> is always 0. If the leaf node is a vector (within a table), <*index*> can be any value from 1 to the number of rows stored in the table.

getValues

This function can be used to return all data stored in a leaf node. Like `getValue`, this function must be executed in a leaf node.

getRowData [<*rowname*>]

This function can be used to return data from a table. This function must be executed from a node that inherits from the `MANAGED-OBJECT-TABLE-ENTRY` primitive. If no <*rowname*> is specified, the function returns all the data. If <*rowname*> is specified, the data for the row reference by that name is returned.

getTableDepth

This function returns the number of rows stored in a table. This function must be called from a node that inherits from the `MANAGED-OBJECT-TABLE-ENTRY` primitive.

setValue <*index*> <*value*>

This function can be used to set the value of a managed property. <*index*> is 0 for all scalar leaf nodes, and 1 or higher for a table property indicating which element in the vector is to be set. <*value*> is the value to be set.

locate <*node name*>

Typically, this function is used together with `toe_send` to enable the evaluation of a command in the context of another object. This function recursively searches up the MIB tree for <*node name*> and returns the unique TOE ID of that object if it is found. <*node name*> can be a absolute path to the node (starting from `.iso`) or a relative path to the node <*node1*>.<*node2*>...

toe_send <*toeid*> <*command*>

This function is used to evaluat the `allows` command in the context of another object. The <*toeid*> is the TOE ID of the object in which the command is to be evaluated. The TOE ID of an object is typically determined using the `locate` command. <*command*> can be any Tcl/TOE command that is valid in the context of the node. For example, `toe_send [ locate node1 ] getValue 0`, retrieves the data value stored in `node1`.

transposeFilter

A useful data filter is the `transposeFilter`, which can be used to transpose a table of data.

rateFilter<*node name*>

This function accepts the name of a managed property and returns the rate of change per second for the managed property since the previous sample.

rateFilter64 <*node name*>

Same as `rateFilter` except for 64-bit integer values.

tableRateFilter<*node name*>

This function is similar to `rateFilter` function, except that is operates on a list of data instead of a scalar.

tableRateFilter64 <*node name*>

Same as `tableRateFilter` except for 64-bit integer values.

pctFilter<*node1*><*node2*>

This function computes the value of a named managed property as a percentage of another managed property.

This function accepts the name of two managed property peers, each of which contains the same number of values. The list of values associated with the first property is computed as a percentage of the list of values associated with the second property. The function returns a list of percentages.

linearFit<*value*>

This function is used to compute the slope of the line that best fits through a set of data values. This function accepts a single numerical argument. This value is stored along with previous values passed into this function. The number of data points stored internally is specified by the `refreshParams` qualifier

digitalFilter<*value*>

This function provides a multiply and accumulate function to provide digital filtering capabilities. This function accepts a single numerical argument that is stored along with other values passed into the function.

The `refreshParams` qualifier specifies the coefficients of the filter. The sum of the coefficients must be one so that the result does not have to be normalized. The number of coefficients indicates the number of data points to store internally.

Alarm Status Strings

A status string can be retrieved for any node via SNMP through the shadowmap.

The status string is a sequence of tab-separated fields. It is constructed out of the state and name of the node, and other relevant information.The exact format of this status string may change as Sun Management Center software development progresses.

This section contains examples of status strings as they currently exist. The purpose of these examples is to show how the node state contributes to the status string, and how the status of underlying child objects is represented in the status of a parent branch object.

Solaris Example of Status Strings—CPU Managed Object

Consider a managed object, CPU, with managed properties of idle time and busy average. If there is no alarm condition on either of the managed properties, the shadowmap status strings are displayed as:

```
Idle Time Status:
{INF-0  fly   Solaris Example  CPU Idle Time   OK
  snmp://204.225.247.154:161/mod/solaris/cpu/idle 0  882368193 }

Busy Average Status:
{INF-0  fly   Solaris Example   Average CPU Usage   OK
  snmp://204.225.247.154:161/mod/solaris/cpu/average 0 882368193}

CPU Status:
{INF-0  fly   Solaris Example    CPU Usage   OK
  snmp://204.225.247.154:161/mod/solaris/cpu 0  882368193 }
```

Now suppose that the idle time is in alarm because the system is less than 10% idle, and the busy average is in alarm because the system is more than 90% busy. Now the shadowmap status strings are displayed as:

```
Idle Time Status:
    {ERR-5  fly   Solaris Example   CPU Idle Time   < 10%
     snmp://204.225.247.154:161/mod/solaris/cpu/idle 25  882368193 }

Busy Average Status:
    {ERR-5  fly   Solaris Example   Average CPU Usage   > 50%
     snmp://204.225.247.154:161/mod/solaris/cpu/average 25 882368193}

CPU Status:
    {ERR-5  fly   Solaris Example   CPU Idle Time   < 10%
        snmp://204.225.247.154:161/mod/solaris/cpu/idle 25  882368193
}
{ERR-5  fly   Solaris Example   Average CPU Usage   > 50%
    snmp://204.225.247.154:161/mod/solaris/cpu/average 25 882368193}
```

Note – The overall CPU status is a list of the alarm statuses of the underlying properties.

In general the contents of a status string is given by a tab-separated string:

```
<alarm state>-<alarm severity>\t<host>\t<module name>\t<medium
description>\t<alarm message>\t<snmp url>\t<alarm level>\t<timestamp>
```

where:

<*alarm state*> is the alarm state value in nickname form (see TABLE A-2). This value drives the icon that is displayed in the console.

<*alarm severity*> is a value from 0 to 9 that is used to rank alarms within each state.

<*host*> is the name of the host that is generating this alarm.

<*module name*> is the name of the module that is generating this alarm.

<*medium description*> is the `mediumDesc` value of the node that is generating the alarm.

For nodes using the `rCompare` rule, <*alarm message*> is <*alarm check*> <*alarm limit*> [<*unit*>]. In the preceding examples, <*alarm check*> is > or <, <*alarm limit*> is 10 or 50, and <*unit*> is %. Other alarm rules can set the this message text explicitly.

<*snmp url*> is the SNMP URL that corresponds to the node that is generating the alarm.

<*alarm level*> is the numeric representation of <*alarm state*>-<*alarm severity*>. The conversion is <*alarm state value*> *10 + <*alarm severity*>. Fore example `ERR-5` has an <*alarm level*> of 25. TABLE A-2 lists the default values for <*alarm state value*> and <*alarm severity*>.

<*timestamp*> is the epoch time when the alarm limit was last evaluated.

TABLE A-2 Alarm Level

Alarm State	State Value	Default Severity
OK	0	0
OFF	0	1
DIS	0	1
INF	0	5
WRN	1	5

TABLE A-2 Alarm Level

Alarm State	State Value	Default Severity
ERR	2	5
IRR	2	7
DWN	2	9

Module Testing Tips

When a module is loaded into the agent and viewed through the Sun Management Center console, information is cached and also saved to files. This is done for performance and to allow the information to be persistent across restarts of the agent. As a result, there are issues to consider when testing changes to a module:

- Before reloading a modified module, restart the agent to ensure that cached files are not used when the module is reloaded.
- If the MIB structure of the module is changed and the module was previously loaded into the agent, then the Sun Management Center Java Server must also be restarted. In addition, delete the file `/var/opt/SUNWsymon/cfg/user-oids-d.dat` before restarting the agent.
- If the module parameters are modified, unload the module from the agent before restarting the agent.

File Naming Conventions

Module definition files adhere to the following naming conventions:

<module><-subspec>-<descriptor>.<extension>

where

<*module*> is the module name.

<*subspec*> is an optional qualifier for the module name.

<*descriptor*> is one of a set of standard descriptors indicating the purpose of the file.

<*extension*> is one of a set of standard file extensions indicating the file type.

By convention, the <*module*> and <*subspec*> portions of the filename are common for all files associated with a specific module. This allows related module files to be easily grouped together while eliminating the chances of filename contention with the definition files of other modules. The following are standard descriptors for module definition files:

-d	Daemon file
`-ruletext-d`	Rule message text file
-models-d	Model file
`-m`	Parameter file
`-ruletext-d`	Rule initialization file

Additional standard descriptors are:

`-j`	`Java console file`
`-s`	`Java server file`

Standard Extensions

The following are standard extensions for module definition files:

`.x`	File in module configuration file format
`.def`	.Default file
.flt	Tcl/TOE Filter file
`.prc`	Tcl/TOE Procedure file
`.tcl`	Tcl commands and procedures
`.sh`	Executable shell scripts
`.dat`	Data file
`.rul`	Tcl/TOE rule file
`.properties`	Internationalization text file

Solaris Example Module Filenames

Some of module definition files for the Solaris Example module must be named as follows:

The following are solaris example module file names:

`solaris-example-m.x`	Solaris Example Parameter file
`solaris-example-d.x`	Solaris Example `Agent` file
`solaris-example-d.def`	Solaris Example `Alarm` file
`solaris-example-d.flt`	Solaris `Filter` file

Mandatory and Optional Module Files

The following lists the required definition files.

TABLE A-3 Mandatory Module Files

<module><-subspec>`-m x`	Parameter file
<module><-subspec>`-models-d.x` *<object*>-models*`-d.x`	Model files (may be multiple files)
<module><-subspec>`-d.x`	Agent file

The following optional files can be defined for each module, depending on the module implementation requirements:

TABLE A-4 Optional Module Files

<module><-subspec>`-d.flt`	Filter file
<module><-subspec>`-d.prc`	Procedure file
<module><-subspec>`-*.sh`	Executable Shell Scripts (can be multiple files)
<module><-subspec>`-d.rul`	Rule file
`<module><-subspec>-ruleinit-d.x`	Rule Initialization file
`<module><subspec>-ruletext-d.x`	Rule Message Text file
<module><-subspec>`.properties`	Properties file
`ServerOverrideBundle.properties`	Server Override Properties file

TABLE A-4 Optional Module Files

<*module*><-*subspec*>-oids-d.dat	Module OIDs file
<*module*><-*subspec*>-traps-d.x	Traps file
<*name*>16x16-j.gif	Standard Icon file
<*name*>32x32-j.gif	Topology Icon file
<*module*><-*subspec*>-d.def	Alarm file

If binary extensions or packages are used by a module to facilitate or optimize data acquisition and alarm processing, one or more of the following files can exist also:

TABLE A-5 Binary Extension Files

<*module*><-*subspec*>-shell.tcl	Package load commands
pkg<*module*><-*subspec*>.so	Standard Tcl package shared object
lib<*module*><-*subspec*>.so	Standard UNIX shared object

Each of the files listed above is discussed in detail in the following sections.

Location of Module Files

All module files except the following must be installed in the `/opt/SUNWsymon/modules/cfg` directory of the agent host. The exceptions to this rule are:

- Shell scripts must be installed in the `/opt/SUNWsymon/modules/sbin` directory of the agent host.
- Properties files must be installed in the `/opt/SUNWsymon/classes/com/sun/symon/base/modules` directory of the host running the Sun Management Center server layer.
- Shared object files and packages must be installed in the `/opt/SUNWsymon/base/lib/<arch>` directory of the agent host.
- The serveroverride properties file is a special file that must be located in `/opt/SUNWsymon/classes` on the server host.
- Standard icon files must be installed in `/opt/SUNWsymon/classes/base/console/cfg/stdimages` directory of the host running the Sun Management Center server layer.

- Topology icon files must be installed in the `/opt/SUNWsymon/classes/base/console/cfg/topoimages` directory of the host running the Sun Management Center server layer.
- The parameter file must be installed on the agent host.

The list of modules available in the Load Module console window is determined only when the agent is first started. When a new module has been added, this list can be updated by forcing the agent to redetermine the list of available modules. This can be done by right-clicking in the Load Module window and selecting the Refresh menu option. The update of the list may take a while depending on the number of modules available. The list can be also be updated by restarting the agent.

Data Management

This section covers the following topics:

The chapter describes the concepts and techniques used in Sun Management Center software to construct models of the entities to be managed. It also describes mechanisms employed by the Sun Management Center agent to enable these models to gather data, determine status, and perform actions on the managed entities.

- Information Model defines the concepts used in Sun Management Center software when modeling entities to be managed. It also describes how these entities can be modeled using TOE objects and primitive classes.
- Operational Model describes how the Sun Management Center agent realizes the management model to manage entities. Sun Management Center agents autonomously collect data and utilize simple alarm checks and/or rules based technology to determine the status of the managed objects. The agent can then generate alarms or perform actions based on the detected conditions, thereby providing predictive failure capabilities and auto-management.
- Management Information Base (MIB) is the repository of the managed entity data and management parameters. Management modules that are loaded into the MIB are also discussed.

For more information about management modules, refer to the Chapter 5.

Information Model

This section describes how the entities to be managed by the Sun Management Center agent are modeled using TOE objects and primitive classes. It also describes how alarm conditions associated with the managed entities are represented.

General Concepts

Sun Management Center software is based on the object oriented paradigm, in which objects are used to model the various aspects of a system for the purpose of managing that system. The physical and logical components of a system that are being managed are referred to as *managed entities.* Managed entities can be disks, boards, hosts, clusters and networks. Managed entities that are host platforms are referred to as *managed nodes*.

The various types of managed entities are modeled using *managed object classes*, and these classes are combined to form a *meta model* for a particular system, the structure of which accurately models the structure of the managed entities it represents. To perform management functions, models must be realized in a process running on a managed node, at which time each managed object class in the model is instantiated into a *managed object*.

Because of the hierarchical nature of the components of a system, managed entities can be the aggregation of other managed entities. Similarly, managed objects that are instantiated during the realization of a model are considered to be the aggregation of all the subordinate managed objects in that model.

An example of this would be a host that is composed of a power supply, boards, a chassis and other components. The host and all subcomponents are considered managed entities, even though the host entity collectively includes the others. In the model of such a system, the managed object class representing the host is an aggregation of the classes representing the other entities. In a realization of this model, the host managed object is an aggregation of managed objects representing the power supply, boards, a chassis and other components.

In Sun Management Center software, models of managed entities usually take the form of a *management module*, and the tree structure of the managed objects and properties within a module is often referred to as a *Management Information Base*, or MIB.

Managed Entity Modeling

Managed entities are modeled using managed objects, which are instances of managed object classes. The managed properties of the managed entities directly correspond to the properties of the managed object classes used to build the model. In a realization of a model, it is these managed properties that contain the information pertinent to the monitoring and management of the managed entity.

When realizing a model, a tree of TOE objects is created that implements the structure and functions of the model. In this realization, a MIB node object is created for every managed object and every managed property in the model.

These objects are derived from a set of primitives, that in turn are derived from the TOE MIB node class, which implements much of the required management functionality, including timed data acquisition, alarm status checking, rule execution, and alarm creation. The object instances are therefore quite adept at general management functions, and the model that describes them is responsible for configuring them for their specific management purpose.

Using this approach, one inconsistency must be understood. The use of a TOE object to represent a managed object would be very straightforward if it were not for the fact that the properties of the managed object cannot be modeled directly by the properties of the TOE object. If this were the case, then the set of properties available to be managed by a TOE object would be limited to the set of properties not used by the TOE object internally to perform its management function. In other words, there could be contention between the object properties and those of the managed object in the model.

It is for this reason, as well as for the simplification of the TOE object implementation, that the properties of managed objects are represented using separate TOE objects. This is a natural function for these objects, which exist primarily to acquire data and take management actions. This means that objects in the realization can correspond to properties of the managed entity, and the properties of the TOE object can, in fact, correspond to qualifiers of the managed entity. This remapping in the realization is necessary given the realization mechanism used.

For example, a file system can be modeled as a managed object represented by a TOE object in the Sun Management Center agent. Conversely, the file system size would be modeled as a managed property but would be represented by another TOE object; instead of a property of the TOE object that represents the file system.

Management Model Primitives

The construction of management models involves the use of management primitives, which are object classes that exhibit specific management behavior. These primitives correspond to the following model elements:

- Managed objects
- Managed object tables
- Managed properties
- Managed property classes

Managed properties are divided into specific primitives based on the data type of the property and the types of alarm checks to be performed on that object (such as integer type with high limits or string type with regular expression checks).

Primitives are composed of several property classes. This means that the type, function, and behavior of the primitive is defined by several broad categories of properties. TABLE A-6 lists the five property classes used to define primitives.

TABLE A-6 Managed Model Primitives

Type of Property Class	Description
Structural	Object tree structure properties
Technique specific	Properties pertaining to security and communication protocols
Realization	Properties defining the data acquisition operations of the object
Management	Properties specifying operational ranges and alarm actions
Management Rules	Inference-based rule specification properties

FIGURE A-16 shows the composition of object primitives using these five property classes.

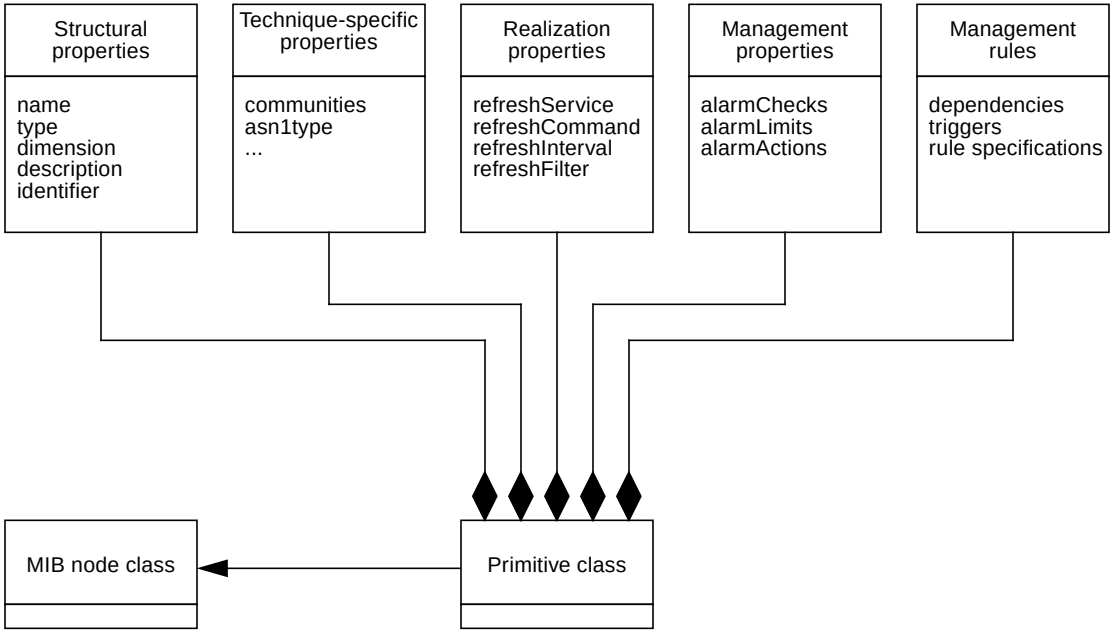

FIGURE A-16 Management Model Primitive Classes

A set of object primitives is available when constructing models that cover all the major object types and alarm check scenarios. These primitives intrinsically define all properties pertinent to SNMP access and ASN.1 description, including the ASN.1 type and the access communities.

Using primitives when constructing models will therefore define most of the properties in the *structural* and *technique-specific* property classes. Properties of the other classes, such as refresh information (for data acquisition) and alarm limits (for status determination) can then be added to the model.

All of the properties associated with a MIB object are effectively defined in the TOE object that represents the MIB object. Most of these properties are accessible through SNMP and the shadow MIB.

Alarm Representation

One of the primary purposes of management models is to detect system events. There are two types of events that can be detected, *hard events*, which are specific occurrences within the system (such as a disk crash or a process termination), and *soft events*, which correspond to a managed property going into or out of an arbitrary range. Hard events can be detected in a very objective way, usually through the presence of a message in a log file or a specific indication in a data acquisition

operation. Soft events, on the other hand, are very subjective, and their occurrence is purely a function of the operational limits associated with the related property or properties.

The nodes of a MIB tree attempt to ascertain the condition of the managed system entities with which they correspond. All changes in an entity's condition correspond to a system event, and the detection of a system event typically leads to a change in the status of a managed object or managed property. Changes in status lead to the creation of an alarm event, which is passed through the system as an indication that the event occurred. It is the creation of these alarms that is of primary importance in the monitoring process.

Alarms contain all of the information useful to clients interested in a particular event. This information includes the identity of the managed node on which the event was detected, a readable portion describing the nature of the event or of the current condition of the entity, a severity number, the time of detection of the event, and the URL of the managed object or property which detected the event. Alarms are intended to be globally valid, and thus all fields, including the readable portions and the URL, are sufficiently qualified to make them completely unambiguous in a global context.

Alarm objects can contain the following fields:

- *Ack/Alarm Label*—the current alarm state of the object and an optional acknowledgment flag. The alarm state is represented by a three letter code (for example, INF for informational, WRN for warning, ERR for error, and so forth) and the ack flag is denoted by a preceding asterisk (*).
- *Target Host*—the host on which the object in alarm exists
- *Module Instance*—the instance name of the module (if applicable)
- *Module Name*—the name of the module
- *Sub-Module Specification*—this field is not supported in the Sun Management Center product; it is always blank
- *Table Row Name*—the row instance name of the object (if the object is an entry in a table)
- *Object Description*—the medium description of the object
- *Problem Info*—the alarm message describing the current status
- *Source URL*—the URL of the object in alarm
- *Alarm Severity*—an integer representing the alarm severity of the object—the higher the number, the more severe the alarm.
- *Timestamp*—the time at which the alarm condition was detected—the time in seconds elapsed since midnight January 1, 1970 (GMT).

The fields in the alarm object are separated by tabs.

Operational Model

Sun Management Center agents manage objects by autonomously collecting and monitoring data. The agents use simple alarm checks and / or rules based technology to determine the status of the managed objects. The agent can then automatically generate alarms or perform actions based on the detected conditions, thereby providing predictive failure capabilities and automanagement. The agents make data and status of the managed objects available to the Sun Management Center server and Sun Management Center console layers.

Operation Sequence

A fully realized model will perform monitoring and management operation at regular intervals or on demand. The objects within the model perform certain operations to achieve this, and the results of these operations are well defined.

In a typical management scenario, the following sequence of events occurs for a managed object or property:

1. Data acquisition request is made.
2. Results of request are forwarded to managed object / property.
3. Data is disseminated into appropriate objects / properties (the data cascade).
4. Alarm rule checks are performed (where applicable) to determine object / property state.
5. Changes in state trigger alarm actions:
    a. Alarm propagates up object tree.
    b. Traps are sent.
    c. Status is logged.
    d. User-defined actions are taken.

Essentially, the nodes in the tree autonomously gather data, place it in the appropriate objects or properties, check limits, fire rules, and take action on state changes. In a normal scenario, no interaction is required between the manager and the agent in order to perform management operations, and the only communication required is the trapping of alarms on state changes.

Data Acquisition Scenarios

To refresh the information in the MIB tree, data acquisition operations must be performed. In Sun Management Center agents, this is generally referred to as the *refresh* operation. Typical refresh operations manifest themselves as the invocation of a *refresh command* in the context of a *refresh service*. A refresh service is an object within the agent that can be used for data acquisition. A refresh command is a service-dependent command that defines the specific operation to perform. Conceptually, the refresh command is sent to the refresh service each time a refresh is triggered.

Refresh services can be any object supporting the service interface. Typically, refresh services can include such things as:

- Objects in the MIB tree (from which you can acquire data)
- Objects maintaining pipes to subshells (such as a Bourne shell, Perl process, or another Tcl shell that can load Tcl extensions)
- SNMP stack (for performing data acquisition from other agents)
- Internal service, which allows access to built-in or dynamically loaded extensions to the agent process.

Services are discussed in detail in the *Agent Framework* chapter in this document.

Note – Since the agent is single-threaded, it is blocked when running Tcl commands in the internal service. If it is expected that a Tcl command can take a significant amount of time to return its result, a Tcl subshell service should be employed to execute these commands. The Tcl subshell process can load the required Tcl extension(s) so that it can execute Tcl commands and return the results to the agent asynchronously.

Cascade Scenarios

The *data cascade* is disseminating a buffer of data into a tree of managed objects or properties. By strictly defining the rules governing data updates, a wide variety of data acquisition scenarios are available. Data can be acquired one piece at a time and placed into managed properties, or larger amounts of information can be acquired in a single data acquisition operation and cascaded into several managed properties or even several managed objects.

In general, all data acquisition operations are initiated by an *active node*. An active node is a managed object or property that has refresh information associated with it. Active nodes can be managed objects, managed property classes, or managed properties, depending on the desired cascade scenario. Also, properties can be of either a *scalar* or a *vector dimension*, and this affects the data update operation.

Conceptually, the data cascade consists of a tree node acquiring information and either placing the information in itself (in the case of managed properties) or passing it down to inferiors in the tree, who in turn either consume it or pass it on. Data left over from one inferior node tree is passed on to the next inferior tree until all the data is consumed. Failure to consume all the information, or there not being enough information to fill the tree, constitute an overflow or underflow condition.

Overflow conditions are not detected by the agent since extra data is discarded. Underflow conditions are not directly detected by the affected nodes. However, other nodes or external clients that query the node for its value detect the absence of data and flag the condition.

Because the structure of the object tree can vary infinitely, so too can the various manifestations of the data cascade. In practice, however, there are only a few common cascade scenarios that lend themselves to several broad categories of tree structure. These cascade scenarios are described in the following sections.

Active Scalar

In this scenario, a property node (which is always a leaf of the tree) is the active node. It initiates a data acquisition (DAQ) operation, receives the results, and places the information in itself. In this scenario, the property is scalar in dimension, meaning it represents one datum, and hence the DAQ operation must return one and only one piece of data. This can be illustrated as follows:

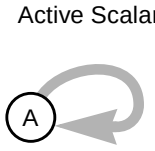

FIGURE A-17 Active Scalar Cascade

An example of an active scalar node is the system uptime managed property. The refresh command of this node computes the system uptime and the uptime value is stored in the node.

Active Vector

As in the active scalar case, active vector cascades result from a single property being the active node. In this case, however, the property is a vector, meaning it represents zero or more pieces of information. The DAQ operation must return zero or more pieces of data, all of which are placed, in order, into the managed property.

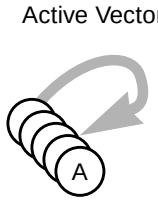

FIGURE A-18 Active Vector Cascade

An example of an active vector node is a managed property that stores the list of files in a directory. The refresh command runs the UNIX `ls` command and the list of files in the current directory are stored in the node.

Compound Scalar

In this scenario, a branch of the node tree is the active object. This branch can be a managed object, managed object table, or a managed property class, but it is never a managed property (which are always leaves). Under this branch are several scalar leaves (managed properties), each requiring one datum per refresh. The DAQ operation in the branch returns several pieces of data, with the data being passed first to the first leaf node, which consumes one piece, and then on to the subsequent leaf nodes, each of which consuming another piece. The amount of data returned by the refresh operation must match the number of leaves under the active node, or an over/underflow condition occurs.

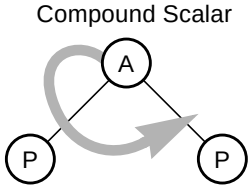

FIGURE A-19 Compound Scalar Cascade

An example of a compound scalar would be a set of nodes modeling the one, five, and fifteen minute load averages of a system. A load managed object is the active branch. Under this branch are the one, five, and fifteen minute load average managed properties. The refresh command of the active branch would return the three load average values and these values are cascaded into the three children nodes.

Compound Vector

This scenario, also known as a *table cascade*, arises when a branch of the node tree contains several property leaves, all of which are vector in dimension. In this case, data cascading down the tree is placed into the vector leaves in equal amounts, with the data order interpreted as row-major and the property leaves treated as columns of a table. If there are N leaves in the tree, then the DAQ operation must return exactly M*N pieces of data, where M is the resulting table depth.

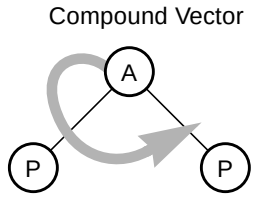

FIGURE A-20 Compound Vector Cascade

An example of a compound vector would be a set of nodes modeling a file system table that contains information for each file system partition. Possible columns in this table would be the partition mount point and size. The refresh command of the branch would then return the mount point name and corresponding size of each file system partition.

Complex Vector

The complex case represents a mixture of the preceding scenarios. In the complex case, the information is passed down through the tree using the general mechanism described above. Scalar leaves consume one piece of data, tables will consume M*N pieces of information and simple vectors consume all they are given.

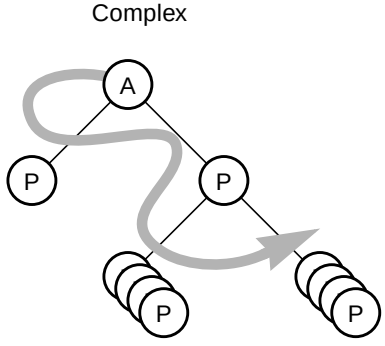

FIGURE A-21 Complex Cascade

An example of a complex cascade scenario involves augmenting the file system table example described earlier with an additional managed property that stores the number of file system partitions. In this case, the active object's refresh command returns the number of partitions, followed by the mount points and sizes of each file system partition.

Nested Heterogeneous

This is where active nodes are placed under other active nodes in the node tree. As a rule, active nodes do not accept information from higher-level cascades. Hence, in this case, the higher-level cascade bypasses the nested active node, and the nested object is responsible for refreshing itself and/or the tree of nodes below it.

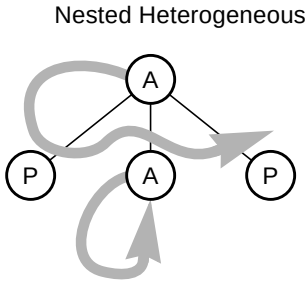

FIGURE A-22 Nested Heterogeneous Cascade

An example of a nested heterogeneous cascade is a set of nodes modeling the process usage of a system. The managed properties consist two passive nodes (number of active processes and number of sleeping processes) and an active node (maximum number of available process slots). The active branch object's refresh command returns the number of active and sleeping processes. The active leaf node's refresh command returns the maximum number of available process slots.

Derived Heterogeneous

Similar to the nested heterogeneous case, this scenario involves a derived node placed under an active node (or another derived node) in the node tree. Like active nodes, derived nodes do not accept information from above. In this case, however, the DAQ operation of the derived node may depend on, and hence be triggered by, the update of the objects around it. The firing of the refresh operation of the derived node therefore is intrinsically linked to the data cascade from the superior object.

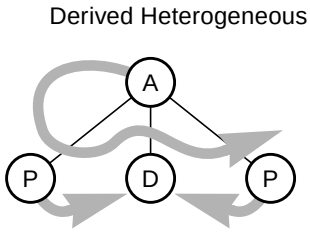

FIGURE A-23 Derived Heterogeneous Cascade

An example of a derived heterogeneous cascade is a set of nodes modeling the swap usage of a system. The managed properties consist of two passive nodes (current swap usage and the total swap) and a derived node (percentage swap used). The active object's refresh command returns the current and total swap. The percentage swap used is then computed from the two returned values.

Derived Nodes

A derived node is a member of the MIB tree that uses other MIB nodes as the service(s) for its refresh. In other words, its value is a function of the values or qualifiers of one or more other managed properties. Through the use of derived variables, it is possible to create nodes whose value represents averages, rates of change, specific digital filters (for example, high pass, low pass, or band pass) or other useful calculated information.

Derived nodes establish dependency relationships with the nodes on which they rely through the use of the *refresh triggers* specification. Nodes can be triggered off the change in value or status of one or more nodes, and refreshes automatically when any of the specified events occur. Derived nodes can also update at an interval, although this is usually unnecessary if the triggers are specified properly.

Alarm Rule Checks

After completion of the full refresh operation (the refresh request and the subsequent data cascade), a set of *refresh actions* occur. For nodes in a MIB tree, these actions include the *alarm rule checks*, which involve checking the data values of the managed properties against a set of alarm criteria.

These alarm checks determine the current status of the managed entities being monitored, as described in the information model. The alarm checks can be classified into simple comparison checks or more complex rule evaluation.

Simple Comparison Checks

Simple comparison checks apply only to single data entries of managed properties and are usually dyadic relational operations involving numeric limits, regular expressions, or comparison strings. The output of these checks is a status code, with the status produced corresponding to the state associated with the most severe alarm check that tests positive. If none of the checks are satisfied, the node is considered to be in the ok state, and nodes with no alarm checks are always considered ok.

Rule Evaluation

Rules provide a mechanism to specify customized alarm checks in place of standard alarm checks that perform simple comparisons. Rules are potentially complex expressions involving the values or status of one or more MIB nodes, and generate values or status that corresponds to the outcome of their computations. As opposed to simple comparison alarm checks, rules can embody complex comparisons, computations and relationships, and the status they produce may represent a very informed decision.

Each rule in the agent has a corresponding MIB node, and this node triggers the evaluation of the rule, maintains any rule-specific qualifiers, and acts as a repository for the resulting data or status.

Having a one-to-one correspondence between rules and MIB nodes facilitates both the triggering of the rule and the generation of alarm objects, as the identity of the MIB node generating the alarm must be placed in the alarm. The URL in the alarm can then point back to a node that represents the rule. Acknowledgment of the alarms generated by a rule and the editing of rule-specific qualifiers can be done through the use of the rule's URL.

Using this approach, the technology to evaluate rules is independent of the triggering mechanism and the alarm generation. Because the rule is fired by the standard triggering mechanisms, and because the values or status of all nodes on which the rule depends can be passed to the rule at the time of triggering, the rule needs to implement the relevant computation or comparison and return the ensuing data or status. Making use of this, a simple Tcl-based rule mechanism are available for implementing the body of a rule, and support for rules based on commercial, third party inference engines can be added easily in the future.

Alarm Actions

If a change in alarm status is detected, an alarm object (as described earlier in the information model) is generated and the following alarm actions are triggered:

- Event information is written to the event circular log file.
- Status is propagated up the MIB tree to all superior nodes.

- StatusChange and event traps are generated to inform the management layers.
- User-defined alarm actions are executed.

Status Propagation

Detection of system events causes a change in the status of the corresponding managed property in the MIB tree. This change in status must be reported to superiors in the MIB tree, as these superiors correspond to the managed objects or managed property classes to which the managed property exhibiting the status change belongs. By propagating status at the time of a status change, all managed objects and properties at any level of the MIB tree are in sync with the current state of their inferiors.

This upward passing of status information is typically referred to as *hierarchical summarization*, and is very important to the operation of both the agents and the management layer. By permitting managed objects at all levels to describe their own status, the determination of status at the server and console levels is greatly simplified.

The placement of status lists in objects at each level of the MIB tree can be diagrammed as follows:

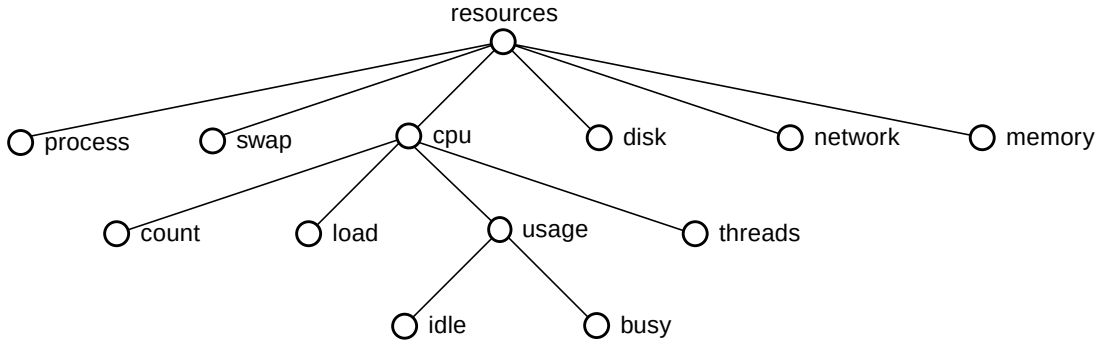

FIGURE A-24 Objects in MIB Tree

Each object in the tree can be queried for its status using shadow SNMP operations. Leaf objects such as *idle* and *busy* only contains their own statuses generated from the alarm rule checks.

Branch objects reflect the status of all its children by containing a list of all exceptional (not ok) status conditions. For example, the status of the *usage* object contains any alarm status conditions of the *idle* and *busy* properties. Similarly, the *cpu* object status are based on the *count*, *load*, *usage*, and *threads* objects. Finally, the *resources* object status contains the statuses of all the objects shown in the tree above.

The ability to query the status of any managed object in the MIB tree allows agents to logically combine the status of many disjointed, structurally unrelated managed objects into a single *logical element group*. Logical element groups can then be used to extend the managed object hierarchy beyond a single agent.

Alarm Status Change and Event Traps

Alarm objects are passed to the management layers through the transmission of SNMP traps. Specifically, two SNMP traps are generated when the status of a managed object changes:

- *statusChange* trap is sent to the Trap Handler. The trap varbind contains the *statusOID* MIB object, whose value is the OID of the managed property whose alarm status changed.
- *event* trap is sent to the Event Manager. The trap varbind contains the *eventInfo* MIB object whose value contain the event version format code, hostname, last line in the event log file, and the event log file creation time. This trap informs the Event Manager of a new event, causing the Event Manager to retrieve the event information from the agent.

These traps are used by the management layers to facilitate centralized event management and alarm correlation.

Event Propagation

When the alarm status of an object is detected, event information is written to a circular log file on the local host and an event trap is sent to the event manager.

The default event log destination is specified through agent's status channel output specification (that is *statusOutput* in the file `base-config.x`). For example, the default event log destination for the Sun Management Center agent is specified as follows:

```
statusOutput = “clog://localhost/../log/
agentStatus.log;lines=250; width=200;flags=rw+;mode=644”
```

The event trap sent to the Event Manager causes the Event Manager to request all the event information that it has not previously retrieved from the agent. This is accomplished by tracking the last known line number in the event file and file creation time.

The Event Manager then stores the retrieved events in the event database where it can be accessed from the Sun Management Center console.

If an event trap is lost, the Event Manager does not immediately request the event information corresponding to the trap. However, upon reception of the next event trap, it will retrieve all events not previously retrieved.

Alarm Logging

All *statusChange* traps received by the Trap Handler are logged to the trap output channel. By default, the log destination is defined in the file `base-config.x` to be a circular log file.

```
trapOutput = "clog://localhost/../log/alarms.log;lines=1000;
width=200;flags=r w+;mode=644"
```

User-Defined Alarm Actions

If any user-defined alarm actions were specified for the managed property and the detected alarm condition, the actions are performed. The execution of the alarm action is logged in the agent's circular log file; however, the user has to explicitly redirect output from the script to a file if this information is required.

User-defined alarm actions are entered through the Actions tab in the Attribute Editor, and are only applicable at present for leaf nodes. An alarm action can be specified for each of the possible alarm levels, namely critical, alert, caution, indeterminate, close, as well as for the case of any change in alarm state.

The alarm action entered is the name (without a path specified) of a user-defined Bourne shell script placed in the bin subdirectory of the directory named by environment variable `ESDIR`. The script must be owned by root and executable. Command line arguments can also be specified following the name of the script. The necessity of root ownership on the script provides an added measure of security, so that only privileged users can create scripts that run automatically.

Special command line arguments the have the following significance can be specified.

TABLE A-7 Special Command Line Arguments

Argument	Significance
`%rowname`	row name
`%state`	current alarm state

TABLE A-7 Special Command Line Arguments

Argument	Significance
`%prevstate`	previous alarm state
`%value`	current value
`%statusstringfmt`	formatted status string (similar to the message in the console tooltip)

There is also a special script for sending email to specified users. The script name to use is simply `email` followed by one or more space-separated UNIX user names. This script causes an email message to be sent to the specified user names with text:

```
SyMON alarm action notification ... statusstring: Critical yangtze
Solaris /var Space Used > 90%
```

Management Information Base (MIB)

The Management Information Base (MIB) is the realization of the managed objects and properties that comprise the management modules currently loaded by the Sun Management Center agent. The MIB is embodied by the ISO subtree described previously.

The MIB makes all the managed objects and properties accessible to other Sun Management Center components through SNMP. The MIB also contains infrastructure for loading management modules and arbitrating user interactions with managed objects and properties.

Modules

Modules are the lowest level of granularity of management models. They embody a set of managed objects and their corresponding properties, and are designed to fulfill a particular management requirement. The scope of a module is typically such that a loaded module incorporates a set of management functions broad enough to completely satisfy a particular management requirement.

Modules are defined using the module configuration file format, described previously. This specification represents a model that, when loaded, created a tree of TOE objects configured to perform the functions defined by that module. The act of loading an X file into a running agent corresponds directly to the realization of the object model, since the relationship between the information model and the underlying object technology is very close.

The management functions of modules can be enabled and disabled through an SNMP request or through the specification of the module's active time window. Disabling a module simply deactivates the autonomous data acquisition normally performed by the module's nodes.

Shadow MIB

The concept of a shadow MIB that supports SNMP access to attributes associated with the managed objects and properties in the agent MIB. These attributes can also be referred to as qualifiers.

The default shadow attributes for all managed objects and properties are specified in the file `base-shadowmap-d.x`. These shadow attribute specifications can be overridden for specific managed objects and properties by specifying the relevant parameters in the appropriate object's configuration file.

Some of the default attributes that are accessible through shadow operations include:

- Refresh attributes—refresh service, command, parameters, and interval
- Timestamp—time at which object / property was last updated
- Alarm criteria—info, warning, error level alarm limits
- Alarm actions—actions can be specified for detected events
- Data logging properties—interval, destination
- Access control configuration—users, groups, and communities
- Object and property descriptions—short, medium, full descriptions

Ad-hoc SNMP Operations

The Sun Management Center agent MIB supports the specification of MIB objects that gather data or execute actions only on demand. These MIB objects are accessible through SNMP, and their execution would normally be triggered by ad-hoc SNMP requests originating from a Sun Management Center GUI client.

Note – Note that these ad-hoc MIB objects do not gather data autonomously, and hence, are not intended for monitoring entities and determining their statuses.

These commands executed by these MIB objects must be synchronous so that the command result can be returned in the SNMP response. Examples of synchronous commands can include such things as Tcl command extensions and Tcl procedures.

Shell commands are not permitted since they are asynchronous. Note that the agent process is blocked while the synchronous command is executed; this blocking is a very important consideration when designing these synchronous commands.

Examples of ad-hoc SNMP requests include:

- Getting the current time on the agent host (for example, use Tcl *clock* command)
- Performing a system call to read the contents of a file (for example, use Tcl *file* command)

For example, the managed property related to file statistics can be associated with the ad-hoc operation to retrieve the file contents.

Other MIB objects can be associated with one or more ad-hoc operations by specifying the appropriate ad-hoc MIB objects in its ad-hoc command shadow attribute. This list of ad-hoc commands specifications are accessible through shadow SNMP operations. For example, the managed property related to process statistics can be associated with the ad-hoc operation to get the process table.

Ad-hoc Probe Operations

The Sun Management Center agent MIB also supports the specification of MIB objects that facilitate the establishment of a stream based connection between a probe client and an agent spawned probe server. These connection based operations are referred to as probe operations and are typically initiated on an ad-hoc basis by the probe client (for example, a Sun Management Center GUI client connected to the Sun Management Center server). The involvement of the Sun Management Center agent permits the use of a consistent security model (namely SNMP usec security) when executing probe requests.

Ad-hoc probe operations are used to support:

- Log file viewing
- File transfers

For example, the managed property related to scanning a logfile can be associated with the ad-hoc operation to view the logfile.

Probe operations are facilitated by the probe server that the Sun Management Center agent runs when servicing probe requests.

Probe Server

The probe server is a generic process that does the following:

1. On startup, accepts command line arguments specifying a command and a connection timeout specification.

2. Opens a listen server socket and writes the port number and a password (a string containing a framed randomly generated number) to standard output. The password string is stored internally.

3. Sets a timer for length of the connection timeout specification.

4. Waits for a connection request; if the timer expires, the process exits.

5. If a connect request is received by the listen socket, the connection is accepted, the listen socket is closed, and the timer is also cancelled.

6. Reads the password from socket and compares it with the stored password. If the passwords do not match, the process exits.

7. If the passwords match, the process redirects `stdin` and `stdout` to the socket connection and executes the command specification.

Establishing a Probe Connection

To establish the stream connection between a probe client and agent spawned probe server, the following mechanism is employed:

1. A probe client sends an SNMP set request to the MIB object in the Sun Management Center agent that supports the desired probe operation. The set request specifies any required parameters for the probe operation.

2. Sun Management Center agent receives SNMP set request and spawns a probe server process, passing in the command and timeout specification. The command specification includes arguments specified in the SNMP set request. These parameters are used to execute the actual probe process once the connection to the probe client is established and authenticated.

3. The probe server process opens a listen socket and returns the number of the opened port and a randomly generated password back to the agent. The pipe connection between the agent and the listen server process is then closed.

4. The agent forwards the port number and password back to the probe client as the SNMP set response.

5. The probe client connects to the port on the probe server process on the agent host. When the connection is accepted, the probe server process closes the listen socket so that no additional connect requests are accepted.

6. The probe client transmits the password to provide authentication. If the password is invalid, the probe server process closes the connection and exits.

7. If the passwords match, the probe server process executes the actual probe application over itself, using the command specification the agent has passed to it on startup.

8. At this point, a stream connection is established between the probe client and a process on the agent host. The communication across this connection depends on the nature of the probe request. Once the request is complete, the connection is closed.

Data Logging

The Sun Management Center agent is configurable to periodically log any managed property in the agent MIB to an internal data buffer and/or to an interface URL for persistent storage.

Each agent maintains a persistent registry of the current data logging requests to ensure that data logging continues when the agent is restarted. The agent can also load a data logging module that allows a console user to view the contents of the data logging registry.

Internal History Buffer

The logging of the value of any managed property to an internal history buffer can be enabled/disabled through shadow SNMP operations. The length and logging interval of the internal history buffer is configurable through shadow SNMP operations.

The buffered data is accessible through shadow SNMP operations. This data is not persistent and is used for things as graphing.

Logging Data to a File

The logging of the value of any managed property to a circular or regular log file can be enabled/disabled through shadow SNMP operations. The logging destination and logging interval is also configurable through shadow SNMP operations.

Data Log Format

By default, the data is logged in the following format:

```
<channel> <date> <component> <alarm code> <host> <module instance>
        <module name> <managed property> =<value>
                <units> <URL> <alarm severity> <timestamp>
```

where:

<*channel*> is the name diagnostic channel this message was logged under.

<*date*> is the timestamp in a date format (for example, Dec. 25 18:00:00.)

<*component*> is the daemon component that printed this message (for example, agent).

<*alarm label*> is the alarm code that specifies the current alarm condition of the managed property (for example, INF-0).

<*host*> is the name of the host

<*module instance*> specifies the module instance, for modules that can be instantiated multiple times.

<*module name*> specifies the module name.

<*managed property*> is the full name of the managed property being logged (for example, for the system uptime in the MIB-II module, .iso.org.dod...mib2.system.sysUptime).

<*value*> is the value of the managed property.

<*units*> specifies the units of the property value.

<*URL*> is the URL of the property (for example, `snmp://<host>:<port>/mod/<modspec>/<property>#<instance>`.

<*alarm severity*> is an integer reflecting the severity of the alarm condition; the higher the number, the more severe the alarm condition.

<*timestamp*> is the time in seconds that have elapsed since midnight January 1, 1970 (GMT).

Data Logging Destinations

Each managed property can be logged to a standard log file or to a circular log file to conserve disk space.

Logging to files can be considered as short term storage. Conversely, logging to a database can be considered as long term storage. Data logged to files can be transferred to a database in a batch fashion. This functionality is not within the scope of standard agent data logging.

If more than one managed property is logged to the same destination, the logged data is interleaved. This should not pose a problem since each logged data entry is tagged with its name and timestamp.

Logged Data Retrieval

The current design of the Sun Management Center agent does not include facilities to retrieve the data logged to a URL through shadow SNMP operations.

Data Logging Registry

This data logging registry maintains a table containing information about data currently being logged. This functionality are implemented in the form of a service and will make data logging requests persistent.

The registry contains a table to store the following information for each data element to be logged:

- *state* indicates the current status of the log destination (ok | error).
- *module name* specifies the name of the module in which the logged property resides.
- *module instance* identifies the module instance, if applicable.
- *property name* specifies the managed property being logged.
- *log interval* specifies when to log the data. This interval can be specified as a simple interval or as a complex time specification.
- *file logging status* indicates the current status of logging to a file (on | off).
- *log URL* specifies the logging destination of the data value.
- *internal buffer status* indicates the current status of logging to the internal cache (on | off).
- *buffer length* specifies the maximum length of the internal history buffer
- *log status* indicates whether the data is being logged successfully

This table is accessible by a data logging registry module to allow Sun Management Center console users to view the data currently being logged by a single agent. The module can be extended to allow console users to do such things as add, edit, and delete data logging request entries. These additional functions are not supported in Sun Management Center software.

APPENDIX **B**

Time Expression Specifications

This appendix describes time format with which complex time intervals can be specified in the agent.

It covers the following topics:

Notation

When specifying values for the time specification, white space is not usually important. However, an exception to this rule is that spaces are not allowed in numbers (that is, 4.5 not 4. 5) or in time (10:03 not 10: 03). In addition, upper or lower case is not important.

The exact specification for the time format is described in the next section. Below are some general comments about the notation used.

- Any string enclosed by “” represents a literal string that must be used.
- DIGIT represent any single numeric value from 0 to 9.
- The notation ‘N*’ before any value can be interpreted as N or more items of that value, where N is 1 or more. For example 2*DIGIT specifies 2 or more DIGIT values.
- To specify 0 or more values, precede the item with ‘*’. For example *DIGIT represents 0 or more DIGIT values.
- To specify an exact number of items, precede the item with the number of values required. For example 1DIGIT represents exactly one DIGIT value.
- The ‘|’ denotes a logical “or” condition. For example 1DIGIT | 2DIGIT represents a value that can have one or two DIGIT values.

- Any item enclosed in [] is optional. For example [`time`] indicates that the `time` value may or may not be specified.

Time Expression Specification

When specifying an interval using the time specification, the following syntax is used (the `refreshInterval` command is used as an example):

```
"refreshInterval" = timex_spec
timex_spec = empty_string
             | simple_exp *( ("&&" | "||") simple_exp )


empty_string = ""
simple_exp = absolute
             | cyclic
             | comparison
             | cron
             | variable_subsitution
```

The time specification can be set to an `empty_string` (““) or can be composed of 1 or more `simple_exp` items connected together using logical “and” and “or” operators. Using the `empty_string` specification is equivalent to specifying `second = *` (see below). The logical operators “and” and “or” are represented by the C && and || operators, respectively. The precedence of the logical operators is the same as in C. In addition, bracketing is also allowed.

The different types of `simple_exp` are describe in the following sections:

Absolute Time Expression Specification

The `absolute` format allows the specification of a single instance in time. The syntax is:

CODE EXAMPLE B-1 Absolute Time Expression Specification

```
absolute = “epoch(“ 1*DIGIT “)”
           | month_of_year “/” day_of_month “/” ( yr | year ) [ time ]
           | month day_of_month [“,”] year [ time ]
month_of_year = <1*DIGIT value from 1 to 12>
```

CODE EXAMPLE B-1 Absolute Time Expression Specification *(Continued)*

```
day_of_month = <1*DIGIT value from 1 to 31>

month = “jan” | “feb” | “mar” | “apr” | “may” | “jun” | “jul” | “aug” |
        “sep” | “oct” | “nov” | “dec” | “january” | “february” | “march”
        | “april” | “june” | “july” | “august” | “september” | “october”
        | “november” | “december”

yr = 2DIGIT
year = <4DIGIT value from 1970 to 2038>

time = (hour12 “:” minute [“:” second] time_suffix )
       | (hour24 “:” minute [“:” second])

hour12 = <1*DIGIT value from 1 to 12>
hour24 = <1*DIGIT value from 0 to 23>
minute = <2DIGIT value from 00 to 59>
second = <2DIGIT value from 00 to 59>
time_suffix = “am” | “pm”
```

The first form specifies the number of seconds from Jan 1, 1970 12:00am GMT. This is provided for UNIX programmers. The second and third forms, allow an optional `time` specification. If this is not specified, then the time used will be midnight of date specified. Examples of the `absolute` specification are:

```
refreshInterval = Jan 3, 1996 10:03:23 pm
refreshInterval = Jan 3 1996 10:03:23 pm
refreshInterval = 01/03/96 10pm
refreshInterval = epoch(234324324)
```

Cyclic Time Specification

The `cyclic` specification allows the specifications of periodic events. The period can be seconds, minutes, hours, days, or weeks in length. The syntax is:

CODE EXAMPLE B-2 Syntax for Cyclic Specification

```
cyclic = “cycle(“ rel_time “)”

rel_time = FLOAT [ rel_units ]
rel_units =  unit_week | unit_day | unit_hour | unit_min | unit_sec
```

CODE EXAMPLE B-2 Syntax for Cyclic Specification

```
unit_week = “w” | “wk” | “wks” | “week” | “weeks”
unit_day = “d” | “day” | “days”
unit_hour = “h” | “hr” | “hrs” | “hour” | “hours”
unit_min = “m” | “min” | “minute” | “minutes”
unit_sec = “s” | “sec” | “secs” | “second” | “seconds”

FLOAT = (1*DIGIT “.” *DIGIT)
        | (*DIGIT “.” 1*DIGIT)
        | *DIGIT
```

If the `rel_uints` are not supplied, then seconds are assumed. If the entire time specification is a in `cyclic` format, the `cycle()`can be dropped. For example, both of the following examples are valid:

```
refreshInterval = cycle( 5 h )
refreshInterval = 5 h
```

Comparison Time Specification

The `comparison` specification can be used to specific conditions on variables that must be true. The syntax is:

CODE EXAMPLE B-3 Syntax for Comparison Specification

```
comparison = [“!”] variable op value

variable = “day_of_week”
           | “week_of_year”
           | “day_of_year”
           | “second”
           | “minute”
           | “hour”
           | “day_of_month”
           | “week_of_month”
           | “month”
           | “year”
           | “date”
           | “time”

op = “<“
     | “<=”
```

CODE EXAMPLE B-3 Syntax for Comparison Specification

```
      | “=”
      | “==”
      | “>”
      | “>=”
      | “!=”

value = “*”
        | value_item *[ “,” value_item ]
value_item = value_basic
             | value-basic “-” value_basic
value_basic = *DIGIT
              | day_of_week
              | day
              | week_of_year
              | day_of_year
              | second
              | minute
              | hour
              | day_of_month
              | year
              | absolute
              | time
```

The optional (!) in front of the variable can be interpreted as a “not” logical operator. When (*) is specified for `value` any value is valid for that variable and that all values will be considered when finding the next beat. For example `minute=*` says that the object will be true all the time and will beat once at the start of every minute.

Only certain types of `value_basic` can be used in with certain `variables` (see the example below). In addition only certain types of `value_basic` can be specified in a range in `value_item`. The `value_basic` types that cannot be specified in a range are: `hour`, `hour24`, `minute`, `second`, and `time`. In addition, ranges can only be used when the “=” `op` is used. For example, `day_of_month < 2-15,25` is invalid and `day_of_month = 2-15,25` is valid.

The ranges for `value_basic` types written maximum to minimum will get translated to minimum to maximum. For example, `day_of_month=25-5`, will be treated the same as `day_of_month = 25-31,1-5`.

Internally, a precision is kept for each `value` so that seconds will be not lost when specifying ranges. For example, `time<8:05am && time>=8:05am` will cover the entire day. Without the precision, there would be a 59-second period that would not be covered (8:04:01am-8:04:59am).

Each of the variable and associated value items is described below.

```
“day_of_week”
value = day_of_week | day
day_of_week = <1DIGIT value from 0 to 7>
day = “mon” | “tue” | “wed” | “thu” | “fri” | “sat” | “sun”
      | “monday” | “tuesday” | “wednesday” | “thursday” | “friday”
      | “saturday”  “sunday”
```

- Monday corresponds to a numeric value of 1.
- Sunday can have a numeric value of 0 or 7.
- Saturday has a numeric value of 6.
- The words sun or sunday correspond to the numeric value 0, not 7.

Do not use <, <=, >=, or > with `day_of_week`, since Sunday can be 0 or 7. For example, `day_of_week <= 7` means the entire week (0-7) whereas `day_of_week<=0 or day_of_week<=Sun` means Sunday only.

Use two end points. For example, use `day_of_week=wed-sun` or `day_of_week>=wed && day_of_week<=Sun` so that the start and end days are clear.

```
“week_of_year”
value = week_of_year
week_of_year = <1*DIGIT value from 1 to 53>
```

- A value of 1 is the first week in the year where there are four days of the week in that year (monday is considered the beginning of a week).
- A value of 0 is defined to be the 1, 2 or 3 days that may lie in that year but come before week 1.

- A value of 53 may occur under certain conditions. For example, if the first day of the year falls on a thursday and the year is a leap year.

```
“day_of_year”
value = day_of_year
day_of_year = <1*DIGIT value from 1 to 366>
“second”
value = second
“minute”
value = minute
“hour”
value = hour
hour = ( hour12 time_suffix ) | hour24
midnight is 0 or 12am
noon is 12 or 12pm
“day_of_month”
value = day_of_month
day_of_month = <1*DIGIT value from 1 to 31>
“week_of_month”
value = week_of_month
week_of_month = <1DIGIT value from 1 to 5>
```

- `week_of_month` represents the number of times that day of the week has occurred in the month.
- Used to represent things like the first sunday in July `week_of_month==1 && day_of_week==sun && month==july`

```
“month”
value = month_of_year | month
“year”
value = year
“date”
value = absolute
“time”
value = time
```

Cron Time Specification

The `cron` specification allows UNIX cron style inputs. The syntax is:

```
cron = “cron(“ minute hour day_of_month month day_of_week “)”
```

The five entries correspond to the usual minute, hour, day of month, month, day of week values respectively. These values can include ranges as in the comparison specification. If the entire time specification is a `cron` specification, then the “cron()” can be dropped. For example both of the following specifications are valid:

```
refreshInterval = cron(5 * * * *)
refreshInterval = 5 * * * *
```

Variable Substitution Specification

The time specification also allows variable substitution. All variables will be de-referenced from the `base-timex-d.x` file. The notation for referencing variables is:

```
variable_subsitution = “$” <character string>
```

For example, the variable `xmas` can be specified in the `base-timex-d.x` file:

```
xmas = “day_of_month = 25 && month = dec”
```

This variable can then be used in specifying an interval:

```
refreshInterval = “!($xmas) && cycle(5s) || ($xmas) && cycle(1hr)”
```

This example indicates that on Christmas, specify a cycle interval with a period of one hour. Otherwise, cycle with a period of 5 seconds.

APPENDIX **C**

Module Building Tutorial

This appendix provides a tutorial describing how to build a module.

It covers the following topics:

Module Example

An example module that monitors the size of a file is described here to illustrate the module construction process. The functionality of the module is simplified to demonstrate the creation of a simple module prototype. Enhanced versions of the module example are then described. The process of the module example is:

1. Monitor the size of the `/var/adm/wtmp` file using UNIX `ls` command.
2. Monitor file size using Tcl file command.
3. Parameterize filename so that any file can be monitored.
4. Add SNMP table management capabilities to monitor more than one file.

Steps to Create a Module

Creating a module consists of creating the definition files that describe the module and writing whatever code is required to perform the data acquisition and alarm checking.

This process comprises:

- Naming the Module
- Creating a Data Model
- Realizing the Model
- Specifying Alarm Management Information

`filesize` Module Version 1—Simple Prototype

To simplify the implementation of the module, it is assumed that the module will always monitor the size of the system file `/var/adm/wtmp`.

Naming the Module

The first step is to name the module and create the parameter file for the module. By convention, the names of the definition files are based on the module name.

The example module to monitor the size of the file `/var/adm/wtmp` can be called *filesize*. The associated parameter file then must be named `filesize-m.x` and contain the following entries:

CODE EXAMPLE C-1 Example Parameter File (`filesize-m.x`)

```
[ load default-m.x ]

consoleHint:moduleParams(param) = module i18nModuleName \
i18nModuleDesc version enterprise i18nModuleType

param:module        = filesize
param:moduleName    = File Size Monitoring
param:version       = 1.0
param:console       = filesize
param:moduleType    = localApplication
param:enterprise    = halcyon
param:location = .iso.org.dod.internet.private.enterprises \
.halcyon.primealert.modules.filesize
param:oid           = 1.3.6.1.4.1.1242.1.2.91
param:desc          = An example module that monitors the size \
of /var/adm/wtmp.
```

CODE EXAMPLE C-1 Example Parameter File (`filesize-m.x`) *(Continued)*

```
param:i18nModuleName  = base.modules.filesize:moduleName
param:i18nModuleType  = base.modules.filesize:moduleType
param:i18nModuleDesc  = base.modules.filesize:moduleDesc

?param:i18nModuleName?i18n = yes
?param:i18nModuleType?i18n = yes
?param:i18nModuleDesc?i18n = yes
```

Creating a Data Model

Creating the data model is the most important step in the module construction process. This step involves identifying the components and properties of the managed entity that are to be included in the data model. These components and properties then must be organized in a tree hierarchy. The data model is specified in a model file.

Note – The data model does not need to contain every component and property of the managed entity. It only needs to contain the information that is pertinent to the determination of the status of the entity. Additional information about the entity can be included at the discretion of the module developer.

In the data model of the `wtmp` file, the managed object is simply the file. Managed properties of the file can include such items as its name, inode, size, last modification date, contents, and so forth. To simplify this example, the data model includes the file size as its only managed property. The size is represented by an `INTHI` primitive data type, which implies that its value is an integer and is capable of performing alarm checks against high limits.

The relationship between the managed object and property is specified in the model file:

CODE EXAMPLE C-2 Example Model File (`filesize-models-d.x`)

```
type = reference

file = { [ use MANAGED-OBJECT ]
     mediumDesc             = File
     consoleHint:mediumDesc = base.modules.filesize:file

     size = { [ use MANAGED-PROPERTY INTHI ]
          shortDesc  = size
```

CODE EXAMPLE C-2 Example Model File (`filesize-models-d.x`) *(Continued)*

```
            mediumDesc = file size
            fullDesc   = Size of file
            units      = bytes

          consoleHint:mediumDesc = base.modules.filesize:file.size
        consoleHint:i18nunits  = base.modules.filesize:units.bytes
      }
}
```

The contents of the corresponding Properties File is shown below:

CODE EXAMPLE C-3 Example Properties File (`filesize.properties`)

```
moduleName=filesize
moduleType=localApplication
moduleDesc=An example module that monitors the size of \
/var/adm/wtmp.

file=File
file.size=file size

units.bytes=bytes
```

Realizing the Model

After the data model has been defined, it is realized by instantiating it in the context of a module and adding data acquisition mechanisms.

For the example, the size of the `wtmp` file is computed by running the UNIX commands `ls -l /var/adm/wtmp | awk '{print $5}'` in a shell context. The execution of these commands is facilitated by the shell service (`_services.sh`), which provides a mechanism to run commands in a shell context.

The commands `refreshService`, `refreshCommand`, and `refreshInterval`, are specified in the `wtmp` object to define the means and the frequency at which the datais acquired. The file object is set to an active node type to enable it, periodically, to acquire data.

CODE EXAMPLE C-4 Example Agent File (`filesize-d.x`)

```
[ use MANAGED-MODULE ]
[ load filesize-m.x ]
[ requires template filesize-models-d ]
_services = { [ use SERVICE ]
```

CODE EXAMPLE C-4 Example Agent File (`filesize-d.x`) *(Continued)*

```
    #
    # Standard Bourne Shell
    #
    sh = {
        command = “pipe://localhost//bin/sh;transport=shell”
        max             = 2
    }
}
initInterval = 0
file = { [ use templates.filesize-models-d.file ]
     type = active
     refreshService = _services.sh
     refreshCommand = ls -l /var/adm/wtmp | awk ’{print $5}’
     refreshInterval = 60
}
[ load filesize-d.def ]
```

Specifying Alarm Management Information

This step involves the specification of default alarm criteria and actions for managed properties. Alarm checks are performed every time the property value is computed. The alarm criteria that can be specified is dependent on the primitive data types used to represent the property.

In the `filesize` example, the size property is an `INTHI` data type; consequently, high alarm limits can be specified.

CODE EXAMPLE C-5 Example Alarm File (`filesize-d.def`)

```
file = {
     size = {
         alarmlimit:error-gt = 2000000
         alarmlimit:warning-gt = 1500000
     }
}
```

`filesize` Module Version 2—Improving DAQ Mechanism

Alternatively, the size can be computed more efficiently in C and integrated with the agent in the form of a Tcl package. The migration of functionality from scripts to C is described in the chapter of this document entitled *"Binary Extensions and Packages"*.

For the `filesize` module example, a Tcl command extension for obtaining file statistics already exists. The Tcl `file` command can be used to get the size of a file in bytes. The agent file is modified to use the Tcl `file` command in place of the UNIX `ls` command.

Note – The `refreshService` must also be set to `_internal` to facilitate the execution of the Tcl `file` command.

Since the module MIB is modified in new version of the module, it is safe to release the new version of the module with the same module name and MIB location.

CODE EXAMPLE C-6 Example Parameter File (`filesize-m.x`)

```
[ load default-m.x ]

consoleHint:moduleParams(param) = module i18nModuleName \
i18nModuleDesc version enterprise i18nModuleType

param:module        = filesize
param:moduleName    = File Size Monitoring
param:version       = 2.0
param:console       = filesize
param:moduleType    = localApplication
param:enterprise    = halcyon
param:location      = .iso.org.dod.internet.private.enterprises
\ .halcyon.primealert.modules.filesize
param:oid           = 1.3.6.1.4.1.1242.1.2.91
param:desc          = An example module that monitors the size \
of /var/adm/wtmp.

param:i18nModuleName  = base.modules.filesize:moduleName
param:i18nModuleType  = base.modules.filesize:moduleType
param:i18nModuleDesc  = base.modules.filesize:moduleDesc
```

CODE EXAMPLE C-6 Example Parameter File (`filesize-m.x`) *(Continued)*

```
?param:i18nModuleName?i18n = yes
?param:i18nModuleType?i18n = yes
?param:i18nModuleDesc?i18n = yes
```

CODE EXAMPLE C-7 Example Agent File (`filesize-d.x`)

```
[ use MANAGED-MODULE ]
[ load filesize-m.x ]
[ requires template filesize-models-d ]
initInterval = 0
file = { [ use templates.filesize-models-d.file ]
    type = active
    refreshService = _internal
    refreshCommand = file size /var/adm/wtmp
    refreshInterval = 60
}

[ load filesize-d.def ]
```

`filesize` Module Version 3—Adding Parameters to File Name Specification

This example illustrates how to allow any file to be monitored by the module. This enhancement will allow the module to have multiple instances in order to monitor multiple files.

As in version 2 of the module, the new version does not modify module MIB, so the new version can be released with the same module name and location.

Instance parameters must be added to parameter file to support the multiple instantiation of the module. In addition, entries for the file name parameter must be added so that the Sun Management Center console user is queried for this information when the module is loaded.

CODE EXAMPLE C-8 Example Parameter File (`filesize-m.x`)

```
[ load default-m.x ]

consoleHint:moduleParams(param) = module i18nModuleName
i18nModuleDesc version enterprise i18nModuleType instance
instanceName i18nFilename

param:module        = filesize
param:moduleName    = File Size Monitoring
param:version       = 3.0
param:console       = filesize
param:moduleType    = localApplication
param:enterprise    = halcyon
param:location      =
.iso.org.dod.internet.private.enterprises.halcyon.primealert.mod
ules.filesize
param:oid           = 1.3.6.1.4.1.1242.1.2.91
param:desc          = An example module that monitors filesize

param:i18nModuleName  = base.modules.filesize:moduleName
param:i18nModuleType  = base.modules.filesize:moduleType
param:i18nModuleDesc  = base.modules.filesize:moduleDesc
?param:i18nModuleName?i18n = yes
?param:i18nModuleType?i18n = yes
?param:i18nModuleDesc?i18n = yes
```

CODE EXAMPLE C-8 Example Parameter File (`filesize-m.x`) *(Continued)*

```
param:i18nFilename =
?param:i18nFilename?description = base.modules.filesize:filename
?param:i18nFilename?access      = rw

param:instance     =
param:instanceName =
?param:instance?description = base.modules.default:instance
?param:instance?reqd        = yes
?param:instance?format      = instance
?param:instanceName?description =
base.modules.default:description
?param:instanceName?reqd        = yes
```

The refresh command must be modified to reference the filename parameter instead of simply monitoring `/var/adm/wtmp`.

CODE EXAMPLE C-9 Example Agent File (`filesize-d.x`)

```
[ use MANAGED-MODULE ]
[ load filesize-m.x ]
[ requires template filesize-models-d ]

consoleHint:mediumDesc = base.modules.filesize:moduleDetail

initInterval = 0

file = { [ use templates.filesize-models-d.file ]
    type = active
    refreshService = _internal
    refreshCommand = file size %i18nFilename
    refreshInterval = 60
}

[ load filesize-d.def ]
```

The instance values must be internationalized.

The corresponding changes to the properties file are:

CODE EXAMPLE C-10 Example Properties File (`filesize.properties`)

```
moduleName=filesize
moduleType=localApplication
moduleDesc=An example module that monitors filesize

filename=File Name
moduleDetail=filesize [{0}]
file=File
file.size=file size
units.bytes=bytes
```

`filesize` Module Version 4—Adding SNMP Table Management Capabilities

The previous versions of this module were limited monitoring the size of a single file. To monitor the size of more than one file, the module needed to be loaded multiple times. Version four of the module can monitor one or more files in a single module instance. To do this, SNMP table management capabilities are added.

In this version of the module, the module MIB must be changed. The most significant change is the introduction of a SNMP table to support the monitoring of multiple files. As a result, the new version of this module must be released with a new module name and MIB location.

Module Name

To distinguish this version of the module from previous versions, the module name has been modified. The subspec *table* is added to indicate that multiple files can be monitored, and the module name becomes *filesize-table.* This version of the module

does not support multiple instantiation. However, this feature can be added in a similar manner as before. The associated parameter file is shown below. Differences between this version and previous versions are in bold.

CODE EXAMPLE C-11 Example Parameter File (`filesize-table-m.x`)

```
[ load default-m.x ]

consoleHint:moduleParams(param) = module i18nModuleName \
i18nModuleDesc version enterprise i18nModuleType

param:module        = filesize-table
param:moduleName    = File Size Monitoring (Table)
param:version       = 1.0
param:console       = filesize-table
param:moduleType    = localApplication
param:enterprise    = halcyon
param:location     = .iso.org.dod.internet.private.enterprises\
.halcyon.primealert.modules.filesizetable
param:oid           = 1.3.6.1.4.1.1242.1.2.92
param:desc         = An example module that monitors the size \
of multiple files.

param:i18nModuleName  = base.modules.filesize-table:moduleName
param:i18nModuleType  = base.modules.filesize-table:moduleType
param:i18nModuleDesc  = base.modules.filesize-table:moduleDesc

?param:i18nModuleName?i18n = yes
?param:i18nModuleType?i18n = yes
?param:i18nModuleDesc?i18n = yes
```

Modifying the Model

To support the monitoring of multiple files, the single managed property `size` must be made part of a SNMP table with other managed properties. The additional managed properties are:

- `rowstatus` - this node is required for SNMP management of tables
- `instance` - this node is used as the index for each row of the table
- `name` - the node is used to store the names of the files being monitored

The complete model file is shown below.

CODE EXAMPLE C-12 Example Model File (`filesize-table-models-d.x`)

```
type = reference
initInterval = 0
file = { [ use MANAGED-OBJECT ]
     mediumDesc             = File
     consoleHint:mediumDesc = base.modules.filesize-table:file
     fileTable = { [ use MANAGED-OBJECT-TABLE ]
         mediumDesc             = File Table
         consoleHint:mediumDesc = base.modules.filesize-table:file.fileTable

         fileEntry = { [ use MANAGED-OBJECT-TABLE-ENTRY ]
             mediumDesc             = File Entry
             consoleHint:mediumDesc = base.modules.filesize-
table.file.fileTable.fileEntry

             index = instance
             rowstatus = { [ use ROWSTATUS MANAGED-PROPERTY ]
                 mediumDesc             = Row Status
                 consoleHint:mediumDesc = base.modules.filesize\
-table:file.fileTable.fileEntry.rowstatus
                 consoleHint:hidden     = true
             }
             instance = { [ use STRING MANAGED-PROPERTY ]
                 mediumDesc             = File Instance
                 consoleHint:mediumDesc = base.modules.filesize\
-table:file.fileTable.fileEntry.instance

name = { [ use STRING MANAGED-PROPERTY ]
                 mediumDesc             = File Name
                 consoleHint:mediumDesc = base.modules.filesize\
-table:file.fileTable.fileEntry.name
                 required               = true
             }
             size = { [ use INTHI MANAGED-PROPERTY ]
                  shortDesc  = size
                  mediumDesc = file size
                  fullDesc   = Size of file
                  units      = bytes

                 consoleHint:mediumDesc = base.modules.filesize-\
table:file.fileTable.fileEntry.size
```

CODE EXAMPLE C-12 Example Model File (`filesize-table-models-d.x`)

```
                consoleHint:i18nunits  = base.modules.filesize-\
table:units.bytes
            }
        }
    }
}
```

Realize the Modified Model

The agent file for this version of the module contains a number of differences from the previous versions. These changes are:

- Adhoc commands are added to support the addition and removal of rows using the Sun Management Center console.
- A Procedure File which defines the refresh command as well as other procedures.
- `setrowActions` are defined for `createAndGo`, `createAndWait`, and `destroy` states. Both the `createAndGo` and `createAndWait` actions simply call a Tcl procedure that triggers a refresh and issues a SNMP trap when it is done. This trap allows the Sun Management Center console to refresh the data immediately. The `destroy` action calls a Tcl procedure `removeEntry` which is defined in the `Procedure` File.
- The `instance` node is given the operational type of derived. This is done so that data is not cascaded into it from the refresh command.

CODE EXAMPLE C-13 Example Agent File (`filesize-table-d.x`)

```
[ use MANAGED-MODULE ]
[ load filesize-table-m.x ]
[ requires template filesize-table-models-d ]

_procedures = { [ use PROC ]
    [ source filesize-table-d.prc ]
}
initInterval = 0
file = { [ use templates.filesize-table-models-d.file _procedures
]
     type            = active
     refreshService  = _internal
     refreshCommand  = getFileSizes
     refreshInterval = 300

     fileTable = {
```

CODE EXAMPLE C-13 Example Agent File (`filesize-table-d.x`) *(Continued)*

```
          fileEntry = {

    consoleHint:tableHeaderCommands = addrow
              consoleHint:tableCommands = addrow unload

              consoleHint:commandLabel(addrow) = \
base.console.ConsoleGeneric:tableRow.addPopup
           consoleHint:commandSpec(addrow) = launchUniqueDialog
%windowID .templates.tools.rowadder objectUrl=snmp://
%targetHost:%targetPort/mod/filesize-table/file/fileTable/
fileEntry#%targetFragment

              consoleHint:commandLabel(unload) =
base.console.ConsoleGeneric:tableRow.deletePopup
              consoleHint:commandSpec(unload) =
requestTableRowOperation %windowID snmp://
%targetHost:%targetPort/mod/filesize-table/file/fileTable/
fileEntry/rowstatus#%targetFragment unload

              rowstatus = {
                  setrowActions(createAndGo)   = refresh
                  setrowActions(createAndWait) = refresh
                  setrowActions(destroy)       = remove
       setrowService() = file

                  setrowCommand(refresh) = refreshValueAndTrap
       setrowCommand(remove)  = removeEntry %rowname
        }
 instance = {
                  type = derived
              }
    name = {
                  access = rw
              }

              size = {
                  defaultvalue = 0
              }
         }
    }
}
[ load filesize-table-d.def ]
```

CODE EXAMPLE C-14 Example: Procedure File (`filesize-table-d.prc`)

```
#
# Tcl proc for refreshCommand
#
# This procedure gets the list of filenames in the table and
determines the size of each file (in bytes) using the Tcl file
command. A list of filename and file size is returned.
#
proc getFileSizes {} {
    #
    # initialize result
    #
    set result ""
    #
    # get list of all filenames
    #
    set files [ toe_send [ locate fileTable*name ] getValues ]

    #
    # loop through each file and determine file size
    # append filename and filesize to result
    #
    foreach file $files {
        set filesize [ file size $file ]
        set result "$result $file $filesize"
    }

    return $result
}
#
# Tcl proc for removing a row
#
proc removeEntry { name } {
    #
    # clear any alarm status and editable parameters (limits,
    # status command, and acks) associated with this row
    #
    set tableObject [ locate fileTable.fileEntry ]
    if { $tableObject != "" } {
toe_send $tableObject cleanupRow $name CLEAR_PARMS
    }
```

CODE EXAMPLE C-14 Example: Procedure File (`filesize-table-d.prc`) *(Continued)*

```
    refreshValueAndTrap
    return [ list "$name Entry Removed" ]
}
```

CODE EXAMPLE C-15 Properties File (`filesize-table.properties`)

```
moduleName=File Size Monitoring (Table)
moduleType=localApplication
moduleDesc=An example module that monitors the size multiple
files.

file=File
file.fileTable=File Table
file.fileTable.fileEntry=File Entry

file.fileTable.fileEntry.rowstatus=Row Status
file.fileTable.fileEntry.instance=File Instance
file.fileTable.fileEntry.name=File Name
file.fileTable.fileEntry.size=File Size

units.bytes=bytes
```

Alarm Management

The alarm file is modified to take into account the new model that is used.

CODE EXAMPLE C-16 Example Alarm File (`filesize-table-d.def`)

```
file = {
     fileTable = {
         fileEntry = {
             size = {
                 alarmlimit:error-gt() = 2000000
                 alarmlimit:warning-gt() = 1500000
             }
         }
     }
}
```

APPENDIX **D**

SNMP Proxy Monitoring Modules

This appendix covers the following topics:

Proxy Monitoring

This appendix describes how to build modules that enable Sun Management Center agents to proxy monitor other legacy SNMP agents running on the same host or other hosts or devices on the network.

These modules allow Sun Management Center agents to manage and determine the status of objects being monitored by legacy agents in the same manner as objects being monitored by typical Sun Management Center modules. This is accomplished by having the Sun Management Center agent query the legacy agent and storing any retrieved data locally. This data can then be processed by the Sun Management Center agent for typical module actions such as alarm limit checking. In addition, traps issued from the legacy agent can be correlated with jobs in the Sun Management Center agent and used to trigger refresh actions that can reacquire data from legacy agents to determine the status in a timely manner.

The following sections describe the differences in the various module definition files for SNMP proxy monitoring modules. Two new module definition files are also described.

Module Parameter File

In addition to the standard module parameters required, SNMP proxy monitoring modules require additional information to be specified in the module parameter file.

These parameters are:

- `targetHost`—the name of the remote host on which the legacy agent is running.
- `targetPort`—the SNMP port of legacy agent.
- `targetEnterprise`—the symbolic OID corresponding to the branch of interest in the MIB of the legacy agent. This is typically the enterprise specific branch. A symbolic OID is an OID that specifies the node names instead of the numerical values. For example, iso.org.dod is a symbolic OID.
- `context`—the MIB context of the object if any.
- `snmpVersion`—the version of the SNMP protocol used by the legacy agent. This can have one of the following values: `SNMPv1`, `SNMPv2c`, or `SNMPv2u`.
- `securityLevel`—this defines the level of security used in the SNMP communication. The valid values are: `priv`, `auth`, or `noauth`. `priv` indicates that the SNMP communication will be encrypted (this is not currently implemented). `auth` indicates that the SNMP communication will be authenticated. This is only valid for SNMPv2u. Finally, `noauth` indicates that no authentication will be done. This is required for SNMPv2c and SNMPv1 protocols.
- `securityName`—this is the security name (or community for SNMPv1 and SNMPv2c) used to perform SNMP gets from the legacy agent.

These additional parameters are used to automatically construct the `refreshCommand` for nodes that inherit from the `TARGET-SNMP` primitive (see below). As such, only SNMP gets from legacy agents are permitted using these module parameters. Additional parameters may be specified for SNMP sets (see below).

These parameters are specified in the module parameter file in the same manner as other parameters and can have optional parameters, such as description, associated with them.

The following example shows the module parameter file for the MIB2 Proxy Module. This example includes the standard module parameters as well as the four additional parameters and associated optional parameters (`description`, `reqd` and `access`) required for SNMP proxy monitoring modules.

CODE EXAMPLE D-1 Example: `mib2-proxy-v2-m.x`

```
[load default-m.x]
# Tabulation and ordering specifications
consoleHint:moduleParams(param) = module i18nModuleName i18nModuleDesc version
console enterprise i18nModuleType instance instanceName targetHost targetPort
targetEnterprise snmpVersion securityLevel securityName context
param:module = mib2-proxy
param:moduleName = MIB-II Proxy Monitoring
param:version = 2.0
param:console = mib2-proxy
```

CODE EXAMPLE D-1 Example: `mib2-proxy-v2-m.x` *(Continued)*

```
param:moduleType    = remoteSystem
param:i18nModuleName  = base.modules.mib2Proxy:moduleName
param:i18nModuleType  = base.modules.mib2Proxy:moduleType
param:i18nModuleDesc  = base.modules.mib2Proxy:moduleDesc
?param:i18nModuleName?i18n = yes
?param:i18nModuleType?i18n = yes
?param:i18nModuleDesc?i18n = yes
param:enterprise        = sun
param:targetEnterprise  = iso.org.dod.internet.mgmt.mib-2
param:snmpVersion       = SNMPv1
param:instance          =
param:instanceName      =
param:targetHost        =
param:targetPort        =
param:context           =
param:securityName      = public
param:securityLevel     = noauth
param:location = .iso.org.dod.internet.private.enterprises.sun.prod.sunsymon.agent.modules.mib2Proxy
?param:targetHost?description       = base.modules.default:targetHost
?param:targetHost?reqd              = yes
?param:targetPort?description       = base.modules.default:targetPort
?param:targetPort?reqd              = yes
?param:targetEnterprise?description = base.modules.default:targetMibEnt
?param:targetEnterprise?reqd        = yes
?param:securityName?description     = base.modules.default:securityName
?param:securityName?reqd            = yes
?param:snmpVersion?description      = base.modules.default:snmpVersion
?param:snmpVersion?reqd             = yes
?param:securityLevel?description    = base.modules.default:securityLevel
?param:securityLevel?reqd           = yes
?param:context?description          = base.modules.default:context
?param:instance?description         = base.modules.default:instanceId
?param:instance?reqd                = yes
?param:instance?format              = instance
?param:instanceName?description     = base.modules.default:instanceName
?param:instanceName?reqd            = yes
```

Module Models File

For SNMP proxy monitoring modules, the module Model file must be used to map out the portion of the MIB in the legacy agent (under the branch specified by the `targetEnterprise`) that is of interest to the module.

For example, a fragment of the MIB2 proxy module model file for the `system` and `udp` branches of the MIB2 tree is shown in the next section.

CODE EXAMPLE D-2 Example: mib2-proxy-models-d.x

```
type  = reference
consoleHint:mediumDesc = base.modules.mib2Proxy:moduleDetail
#
# system Managed Object
# implements MIB-II system Group
#
system = { [use MANAGED-OBJECT]
        mediumDesc = MIB-II System Group
        consoleHint:mediumDesc = base.modules.mib2Proxy:system
        sysDescr = { [use STRING MANAGED-PROPERTY]
                shortDesc      = sysDescr
                mediumDesc     = System Descr
                fullDesc       = MIB-II System Description

               consoleHint:mediumDesc = base.modules.mib2Proxy:system.sysDescr
        }
        sysObjectID = { [use OID MANAGED-PROPERTY]
                shortDesc       = sysOID
                mediumDesc      = System OID
                fullDesc        = Object Identifier of the software system
             consoleHint:mediumDesc = base.modules.mib2Proxy:system.sysObjectID
        }
        sysUpTime = { [use STRING MANAGED-PROPERTY]
                shortDesc    = Time since up
                mediumDesc   = Time since System is up
              fullDesc         = The time in microseconds since the system is up
               consoleHint:mediumDesc = base.modules.mib2Proxy:system.sysUpTime
        }
        sysContact = { [use STRING MANAGED-PROPERTY]
                shortDesc       = Contact
                mediumDesc      = System Contact
                fullDesc        = Contact name for this system
              consoleHint:mediumDesc = base.modules.mib2Proxy:system.sysContact
        }
```

CODE EXAMPLE D-2 Example: mib2-proxy-models-d.x *(Continued)*

```
        sysName = { [use STRING MANAGED-PROPERTY]
                mediumDesc        = system name
                consoleHint:mediumDesc = base.modules.mib2Proxy:system.sysName
        }
        sysLocation = { [use STRING MANAGED-PROPERTY]
                shortDesc         = Time since up
                mediumDesc        = system location
             consoleHint:mediumDesc = base.modules.mib2Proxy:system.sysLocation
        }
        sysServices = { [use INT MANAGED-PROPERTY]
                mediumDesc        = system services
             consoleHint:mediumDesc = base.modules.mib2Proxy:system.sysServices
        }
}
...
...
...
udp = { [use MANAGED-OBJECT]
        mediumDesc        = MIB-II UDP Group
        consoleHint:mediumDesc = base.modules.mib2Proxy:udp
        udpInDatagrams = { [use COUNTER MANAGED-PROPERTY]
                mediumDesc         = udpInDatagrams
             consoleHint:mediumDesc = base.modules.mib2Proxy:udp.udpInDatagrams
        }
        udpNoPorts = { [use COUNTER MANAGED-PROPERTY]
                mediumDesc           = udpNoPorts
                consoleHint:mediumDesc = base.modules.mib2Proxy:udp.udpNoPorts
        }
        udpInErrors = { [use COUNTER MANAGED-PROPERTY]
                mediumDesc          = udpInErrors
               consoleHint:mediumDesc = base.modules.mib2Proxy:udp.udpInErrors
        }
        udpOutDatagrams = { [use COUNTER MANAGED-PROPERTY]
                mediumDesc        = udpOutDatagrams
             consoleHint:mediumDesc = base.modules.mib2Proxy:udp.udpOutDatagrams
        }
        udpTable = { [ use MANAGED-OBJECT-TABLE ]
                shortDesc         = UDP Table
                mediumDesc        = UDP Table
                fullDesc          = UDP Table in MIB-II
                consoleHint:mediumDesc = base.modules.mib2Proxy:udp.udpTable
                udpEntry = {[ use MANAGED-OBJECT-TABLE-ENTRY ]
                        shortDesc         = UDP Entry
```

CODE EXAMPLE D-2 Example: mib2-proxy-models-d.x *(Continued)*

```
                        mediumDesc        = UDP Entry
                        fullDesc          = UDP Entry in MIB-II
                  consoleHint:mediumDesc = base.modules.mib2Proxy:udp.udpTable.udpEntry
                        index = udpLocalAddress udpLocalPort
                        udpLocalAddress = { [ use STRING MANAGED-PROPERTY ]
                                mediumDesc= udpLocalAddress
                                consoleHint:mediumDesc = \
base.modules.mib2Proxy:udp.udpTable.udpEntry.udpLocalAddress
                        }
                        udpLocalPort = { [ use INT MANAGED-PROPERTY ]
                                mediumDesc = udpLocalPort
                                consoleHint:mediumDesc = \
base.modules.mib2Proxy:udp.udpTable.udpEntry.udpLocalPort
                        }
                }
        }
}
```

Legacy MIB OIDs Mapping File

To allow the Sun Management Center agent to reference the legacy agent using URLs, an additional module definition file must be created and loaded into the agent (see the next section). This file is used to map symbolic object identification names to their numeric OID values. The naming convention for this file is <*module*><-*subspec*>-oids-d.dat and can be generated using the script mib2x. This script is located in /opt/SUNWsymon/util/bin/<arch>. The general usage for this script is:

```
mib2x -f <ASN.1 MIB text file> > <module><-subspec>-oids-d.dat
```

This script reads an ASN.1 MIB text file and generates, on standard output, text with the following format:

```
<sym1> = <oid1> [<instance>]
<sym1>/<sym2> = <oid1>.<oid2> [<instance>]
<sym1>/<sym2>/<sym3> = <oid1>.<oid2>.<oid3> [<instance>]
...
```

This file maps symbolic names of nodes in the legacy MIB to OIDs. The text created by the mib2x script may need to be edited manually. The first few lines from the MIB2 Proxy module legacy MIB OIDs file (`mib2-proxy-oids-d.dat`) are:

```
iso = 1
iso/org = 1.3
iso/org/dod = 1.3.6
iso/org/dod/internet = 1.3.6.1
iso/org/dod/internet/directory = 1.3.6.1.1
iso/org/dod/internet/mgmt = 1.3.6.1.2
iso/org/dod/internet/mgmt/mib-2 = 1.3.6.1.2.1
iso/org/dod/internet/mgmt/mib-2/system = 1.3.6.1.2.1.1
iso/org/dod/internet/mgmt/mib-2/system/sysDescr = 1.3.6.1.2.1.1.1
...
```

Module Realization File

To enable the Sun Management Center agent to reference the legacy agent using URLs, OIDs file of the legacy agent must be loaded into the agent. The loading of this file and the qualifiers required to collect data from the legacy agent are described in the following section.

Loading the Legacy MIB OIDs Mapping File

The legacy agent's OIDs file must be loaded into the Sun Management Center agent SNMP OID cache. This is done by specifying the following qualifiers in the module realization file:

```
activateActions(post) = <key>
activateService(<key>) = .services.snmp
activateCommand(<key>) = cache load <module><-subspec>-oids
```

Where *<key>* can be any unique identifier and the *<module><-subspec>*`-oids` corresponds to legacy MIB OIDs file. The `activateActions(post)` qualifier is a space separated list of keys that corresponds to actions that will be performed after the current MIB tree has been instantiated. Each *<key>* must have a `activateService` and `activateCommand` that specifies the command as well as the context in which the command is to be executed.

In this case, there is only one key and it corresponds to the action of loading the legacy MIB OIDs file into the context of the SNMP service object. For example, for the MIB2 Proxy module the qualifiers used to load the OIDs file are:

```
activateActions(post) = loadcache
activateService(loadcache) = .services.snmp
activateCommand(loadcache) = cache load pmib2-oids
```

Data Acquisition

For SNMP proxy monitoring modules, data acquisition is accomplished through proxy SNMP operations such as SNMP get, instead of typical module data acquisition mechanisms such as shell scripts or TCL/TOE code. To facilitate data acquisition for proxy SNMP operations, one of the following primitives should be used:

- `TARGET-SNMP`—to do an snmp get for typical data
- `TARGET-SNMP-BINARY`–`to` do an snmp get for binary data

These primitives automatically set the node type to `active` and constructs the `refreshCommand`. Nodes that inherit from this primitive are typically the nodes that realize the objects from the models file. For example, shown below are the two objects from the MIB2 Proxy module realization file that instantiate the objects from the models file. These objects also inherit from the `TARGET-SNMP` primitive.

CODE EXAMPLE D-3 Example: `mib2-proxy-d.x`

```
[ requires templates mib2-proxy-models-d ]

...

system = { [ use templates.mib2-proxy-models-d.system TARGET-SNMP
]
...
}

...

udp = { [ use templates.mib2-proxy-models-d.udp TARGET-SNMP ]
    udpTable = { [ use templates.mib2-proxy-models-d.udp.udpTable
TARGET-SNMP ]
     }
}
```

Nodes that inherit from the TARGET-SNMP primitive must also specify two additional qualifiers. These two qualifiers are used to automatically construct the refreshCommand.

They are:

- refreshOidPrefix
- refreshOids

The refreshOidPrefix specifies the symbolic OID in the legacy agent from the enterprise branch (as specified by the targetEnterprise parameter) to the node containing the managed properties of interest. The refreshOids qualifier is a comma separated list of managed properties. Together, the targetEnterprise, refreshOidPrefix, and refreshOids specify the full symbolic OIDs used to access the MIB of the legacy agent. For example, suppose data from the symbolic OID shown below is required.

```
iso.org.dod.internet.mgmt.mib-2.system.sysDescr
```

The targetEnterprise is set to iso.org.dod.internet.mgmt.mib-2, the refreshOidPrefix is set to system, and the refreshOids is set to sysDescr#0. The #0 is added to indicate a scalar value. If the managed property was a vector value, then no #0 is required.

Note – The number of items in the refreshOids list is the number of data values that will be acquired from the legacy agent and cascaded into passive managed properties below the active node. As a result, the number of items in the refreshOids list must match the number of passive managed properties below the active node.

Nodes that inherit from the TARGET-SNMP-BINARY primitive must specify a refreshHint qualifier in addition to the refreshOidPrefix and refreshOids qualifiers. refreshHint must specify the conversion from binary to ascii. See *Mapping of the DISPLAY-HINT clause* in *RFC 1903* for more information on the valid contents of refreshHint (Note: octal and binary conversions are not supported for INT nodes and octal conversions are not supported for STRING nodes). The refreshHint specification is typically 1x:, indicating that the value returned from the snmp get is a : delimited string, where each delimited value is a hexadecimal representing a single byte

Shown below is a section from the MIB2 proxy module realization file illustrating the specification of the `refreshOidPrefix` and `refreshOids` qualifiers.

```
system = { [ use templates.mib2-proxy-models-d.sysget TARGET-SNMP
]
    type = active
    refreshInterval = 3600
    refreshOidPrefix = system
    refreshOids =
sysDescr#0,sysObjectID#0,sysUpTime#0,sysContact#0,
                  sysName#0,sysLocation#0,sysServices#0
}
udp = { [ use templates.mib2-proxy-models-d.udp TARGET-SNMP ]
     type = active
     refreshInterval = 3600
     refreshOidPrefix = udp
     refreshOids = udpInDatagrams#0,udpNoPorts#0,udpInErrors#0,
udpOutDatagrams#0
    udpTable = { [ use templates.mib2-proxy-models-d.udp.udpTable
TARGET-SNMP ]
          type = active
          refreshOp = walk
          refreshInterval = 0
          refreshOidPrefix = udp.udpTable.udpEntry
          refreshOids = udpLocalAddress,udpLocalPort
     }
}
```

In the `iftableget` node, `refreshOp` is set to `walk` and the items in the `refreshOids` list do not have #0 appended to them, as the data values to be retrieved are vectors.

By default, the `TARGET-SNMP` primitive sets `refreshOp = get` to perform a single SNMP get operation for scalar values. However, by setting the `refreshOp` to `walk`, the `TARGET-SNMP` primitive will traverse the MIB tree from the point specified and return all values. As a result, all values from the vector are returned.

Shown below is the complete module realization file for the MIB2 proxy module.

CODE EXAMPLE D-4 Module Realization: MIB2 Proxy Module

```
[ use MANAGED-MODULE ]
[ load mib2-proxy-m.x ]
[ requires template mib2-proxy-models-d ]
```

CODE EXAMPLE D-4 Module Realization: MIB2 Proxy Module

```
consoleHint:mediumDesc = base.modules.mib2Proxy:moduleDetail

refreshService = .services.snmp

activateActions(post) = loadcache
activateService(loadcache) = .services.snmp
activateCommand(loadcache) = cache load mib2-proxy-oids

system = { [ USE TEmplates.mib2-proxy-models-d.system TARGET-SNMP
]
    type = active
    initInterval = 1
    refreshInterval = 3600
    refreshOidPrefix = system
    refreshOids = \
sysDescr#0,sysObjectID#0,sysUpTime#0,sysContact#0,sysName#0,\
sysLocation#0,sysServices#0
}
...
...
...
udp = { [ use templates.mib2-proxy-models-d.udp TARGET-SNMP ]
    type = active
    initInterval = 1
    refreshInterval = 3600
    refreshOidPrefix = udp
    refreshOids =
udpInDatagrams#0,udpNoPorts#0,udpInErrors#0,udpOutDatagrams#0
   udpTable = { [ use templates.mib2-proxy-models-d.udp.udpTable
TARGET-SNMP ]
        type = active
        initInterval = 1
        refreshOp = walk
        refreshInterval = 0
        refreshOidPrefix = udp.udpTable.udpEntry
        refreshOids = udpLocalAddress,udpLocalPort
    }
}
```

SNMP Sets

SNMP sets to legacy agents can be made as part of the `setActions` infrastructure (see Chapter 6). The `setService` must be set to `.services.snmp` and the `setCommand` must be set to:

```
set <ip address> <port> -1 {{<varbind>} [{<varbind>} ... ]} \
[-version <snmpVersion>] [-securityLevel <securityLevel>] \
[-securityName <securityName>] [-context <context>] [-timeout <timeout>] \
[-retries <retries>]
```

where:

<*ip address*> is the IP address of the host where the legacy agent is running. If this is the same as the `targetHost` module parameter, then `%targetAddress` can be used for this value.

<*port*> is the port used by the legacy agent. If this value is the same as the `targetPort` module parameter, then `%targetPort` can be used for this value.

<*varbind*> consists of <*url*> <*asn1 type*> <*value*> [<*display hint*>].

<*url*> can be either one of `sym/<symbolic oid>`, `oid/<numeric oid>`, or `mod/<module oid>`.

<*asn1 type*> specifies the type of data being set. This can be `OCTET STRING`, `Integer32`, `NULL`, `OBJECT IDENTIFIER`, `IpAddress`, `Counter32`, `Unsigned32`, `TimeTicks`, `Counter64`, `INTEGER`, or `Gauge32`.

<*value*> is the value to be set.

<*display hint*> is optional and used to convert <*value*> to the appropriate format for setting. See *Mapping of the DISPLAY-HINT clause* in *RFC 1903* for more information (Note that octal and binary conversions are not supported for `INTEGER` types and octal conversions are not supported for `OCTET STRING` types. The <*display hint*> specification is typically used to set binary data in the legacy agents. In such a case, <*display hint*> is typically `1x:`, indicating that the <*value*> is a : delimited string, where each delimited value is a hexadecimal representing a single byte.

<*snmpVersion*> is optional (default is `SNMPv2u`) and specifies snmp version used by the legacy agent. If this is the same as the `snmpVersion` module parameter, `%snmpVersion` can be used for this value.

<*securityLevel*> is optional (default is `auth`) and specifies the SNMP security level supported by the legacy agent. If this value is the same as the `securityLevel` module parameter, `%securityLevel` can be used.

<*securityName*> is optional (default is `espublic`) and specifies the name (or community) with which to perform the SNMP set. If this value is the same as the `securityName` module parameter, `%securityName` can be used.

<*context*> is optional (default is no context) and specifies the MIB context for the object to be set. If this value is the same as the `context` module parameter, then `%context` can be used.

<*timeout*> is optional (default is 30 seconds) and specifies the time out for the SNMP request in seconds.

<*retries*> is optional (default is 3 times) and specified the number of times the SNMP set is retried.

SNMP Set Example

In the code fragment below, `setnode` is a node that has `setActions` defined. Whenever an SNMP set is made to this node, it will execute an SNMP set to a legacy agent. The set command specifies <*ip address*>, <*port*>, <*snmpVersion*>, <*securityLevel*>, and <*securityName*> to be the same as the module parameters. Note, this would indicate that the read and write security names (or communities) are the same. The SNMP set is setting some fixed binary data to the OID 1.3.6.1.4.1.9999.1.1.0.

```
setnode = { ...
    setActions = doset
    setService(doset) = services.snmp
    setCommand(doset) = set %targetAddress %targetPort -1 {{oid/
1.3.6.1.4.1.9999.1.1.0 {OCTET STRING}
0:0:4:da:40:cc:e1:f7:99:1f:e1:0 1x:}} -version %snmpVersion -
securityLevel %securityLevel -securityName %securityName
}
```

Module Trap Action Definition File

In certain cases, the legacy agent may issue traps that the SNMP proxy monitoring module may be interested in. In such a case, the legacy agent must first be configured to send the traps to the Sun Management Center Trap handler.

Once traps are being sent to the Sun Management Center Trap handler, they will be forwarded to the Sun Management Center agent. The SNMP proxy monitoring module must then add to the agent the actions to be preformed for the traps that it is interested in. This is done by loading a new module definition file in to the Sun Management Center agent.

Naming Conventions

The naming convention for this file is <*module*>-<*subspec*>-traps-d.x and the format is as follows:

```
[ requires class trapaction ]
[ inherit classes.trapaction ]

<object1> = {
    [ inherit classes.trapaction ]

    criteria = enterprise
    enterprise = <oid corresponding to targetEnterprise>

    <object2> = {
        [ inherit classes.trapaction ]

        criteria = <criteria1> [ <criteria2> <criteria3> ... ]
        <criteria1> = <value1>
        ...

        trapActions = <key1> [ <key2> ... ]
        trapService(<key1>) = <service>
        trapMethod(<key1>) = <command>
    }

    [ <other objects> ]
}
```

<*object1*> should be a unique identifier indicating the module name.

Sample Specification

Specifying the following:

`criteria` = enterprise

`enterprise` = <*oid corresponding to targetEnterprise*>

allows traps from the specified enterprise (the OID corresponding to the value set in the `targetEnterprise` module parameter) to be passed down to *<object2>* (and other objects if specified) for further processing. This allows *<object2>* to be selective in which traps are processed. Again *<object2>* is another unique identifier.

Specifying the following:

```
criteria = <criteria1> [ <criteria2> <criteria3> ... ] \
<criteria1> = <value1>
```

allows further trap filtering capabilities. The `criteria` qualifier is a space separated list of parameters whose values are checked for comparison.

Valid Parameters

The list of valid parameters are:

- `.agentAddr`—address of the agent originating the trap (for example, 192.83.121.224)
- `enterprise`—the OID up to the enterprise branch from where the trap was issued (e.g. 1.3.6.1.4.1.1242)
- `genericTrap`—the generic type of the trap (for example, 1)
- `specificTrap`—the specific type of the trap (for example, 0)
- `trapOid` —the OID where the trap was issued (not including the enterprise) (for example, 1.1.1.1.1)
- `timeStamp`—time stamp of the trap (for example, 472d 3:17:27.72)
- `receiveTime`—time trap was received (for example, Wed Oct 18 12:12:20 EDT 1995)
- `version`—SNMP version (for example, SNMPv1)
- `community`—SNMP community (for example, public)
- `forwarder`—trap forwarder (for example, 192.83.121.224:30001)

Use `trapOid` in place of `the genericTrap` and `specificTrap` specifications. The `trapOid` specification will support both SNMPv1 and SNMPv2 traps, whereas `genericTrap` and `specificTrap` specifications only support SNMPv1 traps.

For every parameter specified in the `criteria` list, there must be a qualifier that corresponds to the parameter and the value to be checked against.

The following example will check the `agentAddr` parameter for equivalence to 192.83.121.224. :

```
criteria = agentAddr
agentAddr = 192.83.121.224
```

If the test fails, no further processing of the trap is done. The ‘equal to’ test can be changed to ‘not equal to’, by specifying `not` before the value.

The following exa,mple will test the `agentAddr` parameter and fail if it equals 1982.83.121.224.

```
criteria = agentAddr
agentAddr = not 192.83.121.224
```

If no `criteria` list is specified, all traps passed into *<object2>* will activate all trap actions. The specification of actions to be performed on traps is made by the following set of qualifiers:

```
trapActions = <key1> [ <key2> ... ]
trapService(<key1>) = <service>
trapMethod(<key1>) = <command>
```

The `trapActions` qualifier is a space separated list of actions to be performed. For each action in the list, there must be a `trapService` and `trapMethod` specified. Typically the action required on a trap is to refresh the data values.

This is done by specifying:

```
trapActions = <key>
trapService(<key>) = .services.snmp
trapMethod(<key>) = jobFireByTag %agentAddr:%enterprise
```

In this case, the `trapMethod` qualifier specifies a command to fire all jobs associated with the specific agent and enterprise.

Example: Trap Action File for HP JetDirect

Shown below is the trap action file for the HP JetDirect module:

CODE EXAMPLE D-5 Example: `hp-jetdirect-trapspd.x`

```
[ requires class trapaction ]
[ inherit classes.trapaction ]
hpjetdirect = {
    [ inherit classes.trapaction ]
    criteria                = enterprise
    enterprise              = 1.3.6.1.4.1.11.2.3.9.1
    notauth = {
       [ inherit classes.trapaction ]
       #
       # match traps that are not authentication failures
       #
       criteria             = trapOIDRegexp
      trapOIDRegexp       = not ^1\\.3\\.6\\.1\\.6\\.3\\.1\\.1\\.5\\.5$
       #
       # perform trap correlation in job module using trap enterprise
       #
       trapActions                = jobfire
       trapService(jobfire)       = .services.snmp
      trapMethod(jobfire)        = jobFireByTag %agentAddr:/%enterprise
    }
}
```

To load the trap actions file into the agent, an additional key is required in the `activateActions(post)` list (see Chapter 6).

Shown below are the qualifiers required:

`activateActions(post) = <key1> <key2>`

`activateService(<key2>) = .services.trap`

`activateCommand(<key2>) = loadActions <module><-subspec>-traps`

In this case the `activateCommand` qualifier will load the file `<module>-<subspec>-traps-d.x` file into the context of the .services.trap object in the Sun Management Center agent.

Example: Qualifiers for Loading the HP JetDirect Module Trap Actions File

Shown below are the qualifiers for loading the HP JetDirect module trap actions file.

```
activateActions(post) = loadtraps
activateService(loadtraps) = .services.trap
activateCommand(loadtraps) = loadActions hp-jetdirect-traps
```

Example: Qualifiers for Loading Both the OIDs and Trap Actions Files for the HP JetDirect Module

Shown below are the qualifiers for loading both the OIDs and trap actions files for the HP JetDirect module:

```
activateActions(post) = loadcache loadtraps

activateService(loadcache) = .services.snmp
activateCommand(loadcache) = cache load hp-jetdirect-oids

activateService(loadtraps) = .services.trap
activateCommand(loadtraps) = loadActions hp-jetdirect-traps
```

APPENDIX **E**

URL Specifications

This appendix covers the following topics:

Uniform Resource Locator (URL)

The Uniform Resource Locators (URLs) schemes employed in Sun Management Center software comply with the general URL format defined in the RFCs pertaining to URLs.

<scheme>:/ / *<net_loc>* / *<path>*;*<params>*?*<query>*#*<fragment>*

URLs are used by Sun Management Center components to access and interface with various resources. URLs are used to do the following:

- Access values and attributes of managed objects and properties in the Sun Management Center agent MIB via SNMP
- Interface with standard I/O file descriptors
- Interface with regular and circular log files
- Interface with other processes using pipes, UNIX, TCP and UDP sockets
- Interface with the UNIX system logging service (for example, syslog).

The discussion of URLs is divided into SNMP URLs and interface URLs.

SNMP URLs

The Sun Management Center components employ URLs to uniquely identify values and qualifiers of managed objects and properties in Sun Management Center agent MIBs. SNMP URLs permit a more readable representation of SNMP object identifiers (OIDs). These URLs can be resolved to the actual OID by querying the finder object residing in all Sun Management Center agents.

For example, SNMP URLs can be used to access the value of a managed property stored in a Sun Management Center agent MIB such as the MIB-II system description.

SNMP URLs can also access qualifiers associated with managed properties and managed objects. These qualifiers are also referred to as shadow attributes and the act of accessing these qualifiers are known as shadow operations. For example, the refresh attributes of a managed property like `refreshcommand` or `refreshinterval` can be accessed through shadow operations.

SNMP URL Format

The general format of the SNMP URL is shown below:

```
snmp://<host>[:<port>]/<type>/<spec>[?<query>][#<instance>]
```

where

snmp specifies the SNMP scheme

<*host*> specifies the host on which the SNMP agent resides

<*port*> specifies the port the SNMP agent is listening on

<*type*> specifies the SNMP URL type which can be one of *oid*, *sym*, or *mod*

<*spec*> will vary according to the type and specifies the data element being identified

<*query*> specifies the shadow attribute being accessed

<*instance*> specifies the managed property instance being accessed

SNMP URL Types

The following SNMP URL types are supported in Sun Management Center:

- *Numeric* type (oid) uses object identifiers (OIDs) to identify the MIB object
- *Symbolic* type (sym) uses the logical names to identify the MIB object
- *Module* type (mod) uses module, instance, and logical names to identify the MIB object

Numeric

Numeric SNMP URLs are specified using the *oid* type and are comprised of the subidentifiers (subIDs) of the MIB object they represent. Numeric SNMP URLs are easily converted to a SNMP request packet since they contain the object identifier (OID) specifications required to construct SNMP PDUs.

These URLs have the following form:

```
snmp://<host>[:<port>]/oid/<subid1>[.../<subidN>] [?<query>][#<instance>]
```

Note – The separators between subIDs can be either slashes "/" or dots ".".

For example, the value of the system description managed property in the MIB-II module can be accessed using the following numeric SNMP URL:

```
snmp://manila/oid/1/3/6/1/2/1/1/1#0
snmp://manila/oid/1.3.6.1.2.1.1.1#0
```

Symbolic

Symbolic SNMP URLs are specified using the *sym* type and are comprised of the hierarchical name of MIB object they represent. The objects named in the hierarchy can be from the root of the MIB tree (.iso) or relative to the base of the enterprises MIB object (.iso.org...private.enterprises).

They have the following form:

```
snmp://<host>[:<port>]/sym/<name1>[.../<nameN>][?<query>][#<instance>]
```

Note – The separators between names can be either slashes (/) or dots (.). Also, a single slash (/) or a double slash (//) can follow the SNMP URL type, for example:
`snmp://<host>[:<port>]/sym//<name1>[.../<nameN>][?<query>][#<instance>]`
is the same as:
`snmp://<host>[:<port>]/sym/<name1>[.../<nameN>][?<query>][#<instance>]`

The value of the system description managed property in the MIB-II module can be accessed using the following symbolic SNMP URL:

```
snmp://manila/sym/iso/org/dod/internet/mgmt/mib2/system/sysDescr#0
snmp://manila/sym/iso.org.dod.internet.mgmt.mib2.system.sysDescr#0
```

The logical names must be resolved before an SNMP request packet is constructed. The names can be resolved to OIDs by performing a lookup in an URL/OID cache or by sending a finder request to the target agent. Once the symbolic SNMP URL is mapped to a numeric SNMP URL, the SNMP request packet can be built and sent.

Module

Module SNMP URLs are specified using the *mod* type and are comprised of the module specification and the hierarchical name of MIB object relative to the root of the module. The module specification consists of the module name and an optional instance specification, separated by a '+' sign.

They have the following form:

`snmp://<host>[:<port>]/mod/<module>[+<inst>]/<name1>[.../<nameN>] [?<query>][#<instance>]`

Note – The separators between names can be either slashes "/" or dots ".".

For example, the value of the system description managed property in the MIB-II module can be accessed using the following module SNMP URL:

```
snmp://manila:161/mod/mib2-system/sysDescr#0
```

The module names, instances names, and logical names must be resolved to OIDs before an SNMP request packet can be constructed. The names can be resolved to OIDs by performing a lookup in an URL/OID cache or by sending a finder request to the target agent. Once the module SNMP URL is mapped to a numeric SNMP URL, the SNMP request packet can be built and sent.

Shadow Operations

In addition to getting and setting the value of managed properties, SNMP URLs can also be used to access additional attributes of managed properties and managed objects, known as qualifiers.

The set of qualifiers accessible through SNMP includes such things as alarm limits, refresh attributes, descriptions, and so forth. The complete list of available qualifiers for each managed object or property is listed in the file `base-shadowmap-d.x`.

The shadow map is specified in a URL in the ?<*query*> specification.

Some shadow map attributes support the specification of an index to access a specific instance of the shadow map attribute. For example, the status list shadow map allows the list of statuses associated with a managed object or property to be accessed. Specifying `?statuslist` alone as the query accesses the entire list. To access specific elements in the status list, `?statuslist.`*N* can be specified to access the *N*th status. If the specified status instance does not exist, nothing is returned.

SNMP URL Examples

Examples of SNMP URLs for values of managed property and qualifiers of managed objects and properties are provided in this section. Each example is represented using numeric, symbolic (absolute and relative), and module type SNMP URLs.

Managed Property Value (scalar)

Managed properties represent the entities being monitored by the Sun Management Center agent. Managed properties that are scalars are specified by an instance specification of “#0”.

For example, the value of the CPU idle property in the Solaris standard module can be accessed using the following URLs:

Numeric SNMP URL

```
snmp://manila:161/oid/1.3.6.1.4.1.1242.1.1.2.1.2.1.6.1#0
```

Symbolic SNMP URL (absolute)

```
snmp://manila:161/sym/
iso.org.dod.internet.private.enterprise.halcyon.openagent.v4.module
s.operatingSystem.solaris.standard.cpu.idle#0
```

Symbolic SNMP URL (relative)

```
snmp://manila:161/sym/
halcyon.openagent.v4.modules.operatingSystem.solaris.standard.cpu.i
dle#0
```

Module SNMP URL

```
snmp://manila:161/mod/solaris-standard/cpu/idle#0
```

Managed Property Value (vector)

The Sun Management Center agent MIB can also model tabular entities. SNMP URLs support access to specific row entries in such tables through the use of instance specifications (for example, *#<instance>*).

For example, the file system statistics are represented by a table in the Solaris standard module. Each file system partition constitutes a row in this table and partition's mount point name is used as the index. Thus, to access the value of the size managed property of `/usr filesystem` partition, the following URLs can be used:

Numeric SNMP URL

```
snmp://manila:161/oid/1.3.6.1.4.1.1242.1.1.2.1.2.1.8.1.1.2#/usr
```

Symbolic SNMP URL

```
snmp://manila:161/sym/iso.org.dod...local.solaris\
-standard.filesystem. fileTable.fileEntry.size#/usr
```

Module SNMP URL

```
snmp://manila:161/mod/solaris-standard/filesystem/\
fileTable/fileEntry/size#/usr
```

Managed Property Qualifier (Scalar Property, Scalar Qualifier)

Qualifiers associated with managed properties are also accessible using SNMP URLs. Managed property qualifiers are specified using the query specification (that is, *?<query>*).

For example, the refresh interval qualifier of the CPU idle property in the Solaris standard module can be accessed using the following URLs:

Numeric SNMP URL

```
snmp://manila:161/oid/
1.3.6.1.4.1.1242.1.1.2.1.2.1.6.1?refreshinterval#0
```

Symbolic SNMP URL

```
snmp://manila:161/sym/iso.org.dod...local.solaris-standard.\
cpu.idle?refreshinterval#0
```

Module SNMP URL

```
snmp://manila:161/mod/solaris-standard/cpu/idle?refreshinterval#0
```

Managed Property Qualifier (Vector Property, Scalar Qualifier)

Similarly, qualifiers associated with managed properties which are vectors are also accessible using SNMP URLs. These qualifiers are specified using the query (*?<query>*) and instance specifications (*#<instance>*).

For example, the refresh interval qualifier of the file system size managed property for the `/usr` partition in the Solaris standard module can be accessed using the following URLs:

Numeric SNMP URL

```
snmp://manila:161/oid/
1.3.6.1.4.1.1242.1.1.2.1.2.1.8.1.1.2?refreshinter-val#/usr
```

Symbolic SNMP URL

```
snmp://manila:161/sym/iso.org.dod...local.solaris\
-standard.filesystem. fileTable.fileEntry.size?refreshinterval#/usr
```

Module SNMP URL

```
snmp://manila:161/mod/solaris-standard/filesystem/fileTable\
/fileEntry/size?refreshinterval#/usr
```

Managed Property Qualifier (Vector Property, Vector Qualifier)

Qualifiers associated with managed properties can themselves be vectors. Specific elements of the qualifier's vector list are specified using *?<query>.N* to access the Nth element.

For example, the alarm limit qualifier of managed properties is a vector since multiple alarm limits can be specified (for instance. info, warning, error). To access the first alarm limit of the file system size managed property for the `/usr` partition in the solaris standard module, the following URLs can be used:

Numeric SNMP URL

```
snmp://manila:161/oid/
1.3.6.1.4.1.1242.1.1.2.1.2.1.8.1.1.2?alarmlimit.0#/usr
```

Symbolic SNMP URL

```
snmp://manila:161/sym/iso.org.dod...local.solaris-
standard.filesystem.fileTable.fileEntry.size?alarmlimit.0#/usr
```

Module SNMP URL

```
snmp://manila:161/mod/solaris-standard/filesystem/fileTable/
fileEntry/size?alarmlimit.0#/usr
```

Managed Object Qualifier (Scalar Qualifier)

Qualifiers associated with managed object are also accessible using SNMP URLs. Managed object qualifiers are also specified using the query specification (that is, *?<query>*).

For example, the refresh interval qualifier of the CPU managed object in the Solaris standard module can be accessed using the following URLs:

Numeric SNMP URL

```
snmp://manila:161/oid/
1.3.6.1.4.1.1242.1.1.2.1.2.1.6?refreshinterval#0
```

Symbolic SNMP URL

```
snmp://manila:161/sym/iso.org.dod...local.solaris-standard.
cpu?refreshinterval#0
```

Module SNMP URL

```
snmp://manila:161/mod/solaris-standard/cpu?refreshinterval#0
```

Managed Object Qualifier (Vector Qualifier)

Qualifiers associated with managed objects can themselves be vectors. Specific elements of the qualifier's vector list are specified using *?<query>.N* to access the Nth element.

For example, the status list qualifier of managed objects is a vector since multiple status messages can be generated by the managed properties associated with the managed object.

For example, to access the first status message of the file system managed object in the Solaris standard module, the following URLs can be used:

Numeric SNMP URL

```
snmp://manila:161/oid/
1.3.6.1.4.1.1242.1.1.2.1.2.1.8.1.1?statuslist.0
```

Symbolic SNMP URL

```
snmp://manila:161/sym/iso.org.dod...local.solaris-
standard.filesystem.fileTable.fileEntry?statuslist.0
```

Module SNMP URL

```
snmp://manila:161/mod/solaris-standard/filesystem/fileTable/
fileEntry/?statuslist.0
```

Condensed URL specifications

The values and qualifiers of managed properties associated with the same managed object can be specified using a condensed form of SNMP URLs.

For example, the load managed object in the Solaris standard module contains three managed properties (one, five, and fifteen load averages). This managed object also has status and timestamp qualifiers associated with it.

The following condensed SNMP URL can be used to specify the values of the one, five, and fifteen managed properties, and the status and timestamp qualifiers.

```
snmp://manila:161/mod/local.solaris\
-standard.load(one,five,fifteen,?status,?timestamp)#0
```

This condensed SNMP URL expands to the following URLs:

```
snmp://manila:161/mod/solaris-standard.load.one#0

snmp://manila:161/mod/solaris-standard.load.five#0

snmp://manila:161/mod/solaris-standard.load.fifteen#0

snmp://manila:161/mod/solaris-standard.load.?status#0

snmp://manila:161/mod/solaris-standard.load.?timestamp#0
```

Interface URLs

The interface library provides a common interface for I/O communication for agent components. These interfaces are not supported by the console and server components. Currently, the supported I/O schemes are:

- `clog`—circular log file
- `desc`—file descriptor
- `file`—text file
- `inet`—internet sockets
- `pipe`—UNIX pipe
- `syslog`—UNIX system logging facility
- `unix`—UNIX sockets

Intraface options can be incorporated into most interface URLs. These options are described later in this appendix.

clog

Circular log files are fixed sized files which can be opened for writing and/or reading. Circular log files are identified using the clog scheme and are always assumed to be on the local host.

The circular log file URL has the following format:

```
clog://localhost/<filename>[;<intraface_options>][;lines=<lines>]\
[;width=<width>][;flags=<flags>][;mode=<mode>]
```

where:

<*filename*> is the path and file name of the circular log file. Note that the path is assumed to be relative to the current working directory (for example, `$ESDIR/cfg`) unless it begins with a "`/`", in which case, it is assumed to be an absolute path.

<*intraface_options*> are intraface specifications.

<*lines*> is the maximum number of lines in the log file. The default value is 1000.

<*width*> is the number of characters per line. The default value is 80. Lines that exceed the width specification are truncated.

<*flags*> is the file permission to use when opening the file. The valid values are `r` | `ro` | `rw` | `rw+` and the default value is `rw+`

<*mode*> is the 3-digit octal mode that shall be assigned to the file. If not set, 666 and the user's umask will be used.

The following examples demonstrate how various circular log files can be specified using the clog URL (assuming that the current working directory is `$ESDIR/cfg`):

```
/tmp/circular.log  --> clog://localhost//tmp/circular.log
$ESDIR/cfg/abc.log --> clog://localhost/abc.log
$ESDIR/log/xyz.log --> clog://localhost/../log/xyz.log
```

desc

File descriptor URLs are identified using the desc scheme and are always assumed to be on the local host.

The file descriptor URL has the following format:

```
desc://localhost/<file_desc>[;<intraface_options>]
```

where

<*file_desc*> is either a numeric file descriptor corresponding to an already open file or one of the standard file descriptor names (for example, `stderr`, `stdout`, or `stdin`)

<*intraface_options*> are intraface specifications.

Examples of desc URLs are provided below:

```
Standard Output --> desc://localhost/stdout
Standard Error --> desc://localhost/stderr
```

file

Standard ASCII text files are can also be specified using file URL. Standard files are identified using the file scheme and are currently always assumed to be on the local host.

The file URL has the following format:

```
file://localhost/<filename>[;<intraface_options>][;flags=<flags>]
```

where

<*filename*> is the path and filename. Note that the path is assumed to be relative to the current working directory (that is, `$ESDIR/cfg`) unless it begins with a "`/`", in which case, it is assumed to be an absolute path.

<*intraface_options*> are intraface specifications.

<*flags*> is the file permission for the file. The valid values are: `r` | `ro` | `w` | `wo` | `w+` | `wo+` | `rw` | `rw+`. The default value is `rw`.

The following examples demonstrate how various files can be specified using the file URL (assuming that the current working directory is `$ESDIR/cfg`):

```
/tmp/regular.log  --> file://localhost//tmp/regular.log

$ESDIR/cfg/abc.log --> file://localhost/abc.log

$ESDIR/log/xyz.log --> file://localhost/../log/xyz.log
```

inet

Internet socket interfaces can be specified using inet URLs. These URLs can be used to specify client or server TCP and UDP sockets.

These URLs have the following format:

```
inet://[<host>]:<port>/<protocol>[;<intraface_options>]
```

where

<*host*> is optional. If <*host*> is specified, the socket is opened as a client to connect to the port on the specified host. The <*host*> can set to 'localhost', a host name, or an IP address. If <*host*> is not specified, the socket is opened on the local host as a server listen socket.

<*port*> is the port to connect to or open, depending on whether the socket is a client or a server. The port number can be specified explicitly (for example, 161). Alternatively, the service name associated with the port can specified (for example, SNMP)

<*protocol*> is the type of socket to use. Valid values are: `tcp` | `udp`

<*intraface_options*> are intraface specifications.

Various inet URL examples are listed below:

```
SNMP UDP Socket Port 161 --> inet://:161/udp

TCP Listen Socket Port 20000 --> inet://:20000/tcp

TCP Connect Socket to Host bob on Port 30000 --> inet://bob:30000/tcp
```

pipe

UNIX pipe interfaces can be used to specify a pipe connection to another process. These URLs are denoted by the pipe scheme and support pipes on the local host only.

Pipe URLs have the following format:

```
pipe://localhost/<command>[;<intraface_options>][;flags=<flags>]
```

where

<*command*> is any valid UNIX command with an optional path. Note that the command path is assumed to be relative to the current working directory unless it begins with a "`/`", in which case, an absolute path is implied.

<*intraface_options*> are intraface specifications.

<*flags*> is the file permission for the pipe. The valid values are `r` | `ro` | `w` | `wo` | `rw` and the default value is `rw`.

An example of a pipe URL that establishes a pipe connection to a bourne shell using the shell transport intraface is specified below:

```
pipe://localhost//bin/sh;transport=shell
```

syslog

An interface to the UNIX system logging facility (that is, syslog) can be specified using syslog URLs. These URLs are denoted by the syslog scheme and support interfacing with the syslog daemon on the local host only. It should be noted that the syslog facility itself supports remote logging.

UNIX syslog URLs have the following format:

```
syslog://localhost/<priority>[;<intraface_options>][;app=<appname>]
[;facility=<facility>] [;logopt=<logopt1>|<logopt2>|...]
```

where

<*priority*> is the priority level of the message and can be one of: LOG_EMERG | \ LOG_ALERT | LOG_CRIT | LOG_ERR | LOG_WARN | LOG_NOTICE | \ LOG_INFO | LOG_DEBUG

<*intraface_options*> are intraface specifications.

<*appname*> is the application name used to identify the message.

<*facility*> is the system log facility to enter the message under and can be one of: LOG_KERN | LOG_USER | LOG_MAIL | LOG_DAEMON | LOG_AUTH | LOG_SYSLOG\ | LOG_LPR | LOG_NEWS | LOG_UUCP | LOG_CRON | LOG_LOCAL0 \ | LOG_LOCAL1 | LOG_LOCAL2 | LOG_LOCAL3 | LOG_LOCAL4 | LOG_LOCAL5 | \ LOG_LOCAL6 | LOG_LOCAL7.

The default value is LOG_USER.

<*logopt*> are any logging options that are OR'ed together. The valid values are: LOG_PID, LOG_CONS, LOG_NDELAY, LOG_NOWAIT

The default syslog URL employed by the agent daemon is shown below:

```
syslog://localhost/LOG_ALERT;app=agent; facility=LOG_DAEMON;logopt=LOG_PID
```

For more information about the syslog parameters, refer to the *syslog.conf* man page.

UNIX

UNIX socket interfaces can be specified using unix URLs. These URLs can be used to specify client connection or server listen sockets on the local host only.

These URLs have the following format:

```
unix://localhost/<filename>[;<intraface_options>][;role=<role>]
```

where

<*filename*> is the UNIX filename to use for the socket connection

<*intraface_options*> are intraface specifications.

<*role*> is either ‘listener’ or ‘connected’. The default value is ‘connected’

Examples of UNIX URLs are listed below:

```
unix://localhost/file;role=listener

unix://localhost/file;role=connected
```

Intraface Options

Intraface options can be incorporated into an interface URL to specify additional layers to the communication channel.

Intraface options have the following format:

```
[;ace=<ace>][;pie=<pie>][;transport=<transport>]
```

where:

<*ace*> is the authentication, compression, and encryption option (for example, `sin`)

<*pie*> is the parameter insertion and extraction option
(for example, `tid | tsid | type`)

<*transport*> is the transport option (for example, `ctrld | eot | eotn | shell`)

Parameter Insertion and Extraction (PIE)

The PIE layer provides the ability to employ a predefined format for the messages being transmitted across the interface. The insertion function is used to insert the message header information and message data when writing a message to the interface. The extraction function is used to extract the message header information and message data when reading data from the interface.

Currently supported PIE types are:

- `tid` - insert `/extract` type and id parameters
- `tsid` - insert `/extract` type, subtype, and id parameters
- `type` - insert `/extract` type parameter

Authentication, Compression, Encryption (ACE)

The ACE layer provides authentication, compression, and encryption functions for communication across the interface. This layer allows a message to be compressed and encrypted before it is written to an interface. Conversely, it allows the same message to be authenticated, uncompressed, and decrypted when reading from an interface.

Currently supported ACE types are:

- `sin` - encrypt/decrypt using a sin-wave algorithm

Transport

This layer provides framing functionality when reading and writing across an interface.

Currently support transport types are:

- `ctrld` - uses ctrl-d (\004) to delimit packets
- `eot` - uses eot (\004) to delimit packets
- `eotn` - uses eot\n (\004\n) to delimit packets
- `shell` - uses '\necho \004\n' to delimit packets

APPENDIX **F**

Status Propagation

This appendix covers the following topics:

This appendix provides an example topology hierarchy configuration to demonstrate how status changes are propagated from the Sun Management Center agent to the Topology agent and Sun Management Center console.

Note – The Domain Alarms at the top of the Sun Management Center console should show the count of host entities in alarm throughout the topology hierarchy. The count does *not* include the cumulative alarms within each host.

Example Topology Hierarchy

The following figure illustrates an example topology consisting of the following domains, groups, and entities:

- Domain A
- Child Group B
- Host entities C and D
- Modules E and F residing in Host C and modules G and H in Host D.

In addition, there is a console viewing Domain A and there are no current alarms. Thus, the total alarm counts are zero in the console.

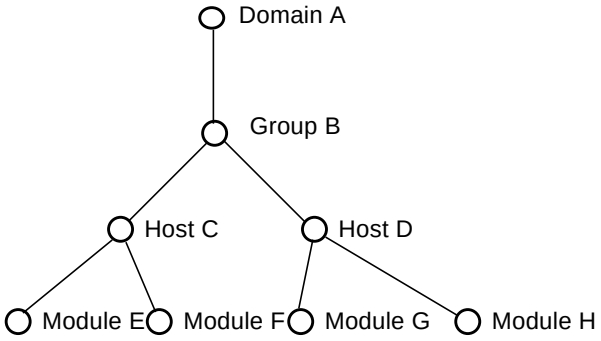

FIGURE F-1 Example Topology Hierarchy

The following sections present example events or scenarios:

- Event 1: Node in Module E on Host C Goes into Error (Red)
- Event 2: Node in Module G on Host D Goes into Warning (Amber)
- Event 3: Node in Module F on Host C Goes into Warning (Amber)
- Event 4: Another Node in Module E on Host C Goes into Warning (Amber)

Event 1: Node in Module E on Host C Goes into Error (Red)

- A status trap indicating a ‘status change for module E on host C’ is generated.
- Since the error in module E is the most severe alarm on host C, the overall status of host C changes, causing another status trap indicating a status change for host C.
- Group B repolls host C, since the status trap for host C correlates with an existing SNMP job of group B.
- Since the host C in error is the most serious in group B, the overall status of group B is now in error as well. This generates a status trap indicating a status change for group B.
- A console viewing Topology group B would display the error immediately since this last status trap from group B would trigger a repoll of the group B and a subsequent screen update. Also, the host C icon gets a red circle at this point. The total count at the top of the screen still shows 0.
- A status trap for group B also triggers a repoll of domain A since it correlates with an existing SNMP job. Domain A immediately repolls group B for its current Status and child host alarm counts.

- The group B error alarm is the most severe alarm condition in domain A. This causes domain A to generate a status trap indicating a status change for domain A.
- If the console was viewing the top level of domain A, this status trap triggers a repoll of the domain A status, and thereby causes a red circle to appear on the group B icon in the display.
- The domain counts on the Console screen are triggered by the domain A status trap, to repoll domain A for the current alarm counts. The screen updates, showing the error count as 1.

All of the above happens within a few seconds, since it is all trap based.

Event 2: Node in Module G on Host D Goes into Warning (Amber)

Assume this second event takes place after the first. This alarm is not as severe as the initial alarm currently in effect on host C.

- Status trap is generated indicating a status change for module G on host D.
- Since this is the most severe alarm condition on host D, the overall status of host D is warning. Another status trap indicating a status change for host D is generated.
- Group B detects a status change on host D and immediately repolls host D for its current status.
- The warning on host D is less severe than the error on host C, so the overall status of group B does not change (still is equal to the host C error). Nonetheless, group B generates a status trap indicating a “status change for group B”. This trap is issued to inform higher topology layers about the change in child host D in group B.
- A console viewing group B would display the host D icon to be amber (warning) since this last status trap from group B would trigger a repoll and screen update. The total warning count at top of screen still shows 1 error (from host C), but 0 warnings (host D warning count not propagated up yet).
- Domain A detects the status trap for the group B and repolls group B for its current status and counts.
- Although the status of domain A's children have changed (group B was error before, and is still in error), domain A recognizes that the total child host count has changed: before there was one error, now there is one error and one warning. Accordingly, domain A generates a status trap indicating a status change in domain A. This trap will trigger a repoll and update of the Domain Counts on the Console Screen.

Event 3: Node in Module F on Host C Goes into Warning (Amber)

Assume this third event takes place after the others. This is a second event on host C. This alarm is only a warning and is less severe than the previous alarm on host C, which was an error. This new alarm is in a “different” module from the previous host C alarm.

- Status trap indicating a status change for module F on host C is generated.
- This is not the most serious condition on host C, but host C generates a status trap anyway, to indicate to upper layers that the status of module F changed (because this new alarm is in a different module, and module F was not in alarm before).
- Group B detects the status trap from host C and repolls host C for its current status.
- Group B notes that the overall status of host C has not changed, so group B does not need to generate any new traps (no change in child counts or status for group B).

Event 4: Another Node in Module E on Host C Goes into Warning (Amber)

Assume this fourth event takes place after the others. This one is a warning in another node in module E (there is already an error on this module from one of the previous events).

- Status trap issued indicating status change for a node in module E on host C.
- This is not the most severe alarm condition on module E on host C (an earlier error alarm condition was detected on another node). The overall status of host C Is unchanged and no new status trap is issued.
- No counts or status need to be updated anywhere in the topology.

Missed SNMP Traps

The delivery of SNMP traps is not guaranteed. If a status change trap is missed, then status and counts will not be updated immediately. However, this will correct itself on the next poll (generally less than five minutes).

APPENDIX **G**

SNMP Trap Subscription

This appendix covers the following topics:

Sun Management Center Agent Components and Trap Subscription

Sun Management Center agent components (including the Trap Handler) support SNMP trap subscription. Trap subscription allows interested parties to request that selected SNMP traps be forwarded to them. Sun Management Center agent components can also subscribe for traps.

Every Sun Management Center agent component supports a MIB that contains the `.iso*base.trapForward` branch, which contains the following nodes:

- `clientRegistrar`—supports trap subscription for specific traps through the specification of the trap criteria.
- `jobAdder`—for existing trap subscriptions with the *tag* criteria set to be true. This node supports incremental additions to the list hosts of interest.
- `jobRemover`—for existing trap subscriptions with the *tag* criteria set to be true, this node supports incremental deletions from the list of hosts of interest.

The `jobAdder` and `jobRemover` nodes are designed to be used with the Sun Management Center agent SNMP job caches, which are used to support periodic SNMP jobs.

Subscribing for Traps

The clientRegistrar node is used to subscribe for traps and is located at:

```
iso.org.dod.internet.private.enterprises...base.trapForward.\
clientRegistrar
```

For the SUN enterprise MIB, the corresponding url for this node is:

```
snmp://<host>:<port>/sym//iso/org/dod/internet/private/
enterprises/sun/prod/sunsymon/agent/base/trapForward/
clientRegistrar#0
```

To subscribe for SNMP traps from a Sun Management Center agent component, perform an SNMP set of a trap subscription specification into the clientRegistrar node. If a trap client has an existing trap subscription, a subsequent subscription replaces the previous subscription. The trap subscription specification has the following format:

```
{<ipAddress>:<snmpPort> {{{<spec1>} {<spec2>} ...}}}
```

where:

ipAddress is the IP Address of the trap subscriber.

snmpPort is the SNMP port of the trap subscriber.

specN is the trap filter criteria, filter criteria expressions, or subscription expiry specification.

Trap filter criteria are specified using the following format:

```
{<criteria> <regexp>}
```

where:

criteria is the trap filter criteria. These criteria are mapped to the contents of the trap PDU.

regexp is the regular expression for the corresponding criteria

Possible criteria values are:

- trapAddress—IP address of trap originator
- trapOID—Trap object identifier

- `oid`*N* - Nth OID in trap varbind
- `value`*N* - Nth value in the trap varbind
- `tag` - flag indicating whether to use the *taglist* criteria which contains a list of host IP addresses when determining a match (that is, if true, use taglist; if false, ignore taglist)
- `taglist` - list of host IP address to match against the IP address of trap originator. The entries cannot be regular expressions, they must be actual IP address.

The tag and taglist criteria is designed to be used by agent SNMP job cache, which subscribes for traps from a specific a list of IP addresses hosts. This explicit list of IP addresses is used in place of a regular expression to support the dynamic modification of the list via the `jobAdder` and `jobRemover` nodes.

By default, if multiple trap filter criteria are specified, they are OR'd together. To modify this behaviour, filter criteria expressions can be used.

Filter criteria expressions are specified as follows:

```
{filter-exp {%<criteria1> <logical operator> %<criteria2> ...}}
```

where:

`criteria`*N* is one of the filter criteria specified above

`logical operator` can be a logical AND '&&' or logical OR '||'

Subscription expiry is specified as follows:

```
{expiry <seconds>}
```

where:

`seconds` is the number of seconds before subscription is cancelled. For a subscription that does not expire, 0 can be specified. If the expiry is not specified, it defaults to ~46 days.

Trap Subscription Examples

For example, if your process is on host:port 204.225.247.123:162 and you wish to subscribe for `linkDown` (whose trap OID is 1.3.6.1.6.3.1.1.5.3) traps from any agent, set the following trap subscription spec:

```
{204.225.247.123:162 {{{trapOID 1.3.6.1.6.3.1.1.5.3}}}}
```

Similarly, to subscribe for all linkDown (1.3.6.1.6.3.1.1.5.3) and linkUp (1.3.6.1.6.3.1.1.5.4) traps, set the following trap subscription spec:

```
{204.225.247.123:162 {{{trapOID
^1\\.3\\.6\\.1\\.6\\.3\\.1\\.1\\.5\\.[34]}}}}
```

Note – The regular expression includes double backslashes '\ \' for the dots '.' since one set is removed when the expression is processed by the Tcl procedure that processes the trap subscription request.

To subscribe for all linkDown traps originating from a specific agent on host:port 204.225.247.100:161, set the following trap subscription spec:

```
{204.225.247.123:162 {{{trapAddress 204.225.247.100} {trapOID
1.3.6.1.6.3.1.1.5.3} {filter-exp {trapAddress && trapOID}}}}}
```

For example, if your process is on host 129.146.53.216 port 2000, and you wish to subscribe for statusChange (whose OID is 1.3.6.1.4.1.42.2.12.2.0.1) traps from any agent, set the following trap subscription spec:

```
{129.146.53.216:2000 {{{trapOID 1.3.6.1.4.1.42.2.12.2.0.1}}}}
```

Similarly, to subscribe for statusChange, valueRefresh, moduleLoaded, moduleUnloaded (respective OIDs are 1.3.6.1.4.1.42.2.12.2.0.1 1.3.6.1.4.1.42.2.12.2.0.2 1.3.6.1.4.1.42.2.12.2.0.4 1.3.6.1.4.1.42.2.12.2.0.5) traps from any agent, set the following trap subscription specification:

```
{129.146.53.216:2000 {{{trapOID 1.3.6.1.4.1.42.2.12.2.0.[1245]}}}}
```

Similarly, to subscribe for statusChange, valueRefresh traps from an agent running on 129.146.53.216, set the following subscription specification:

```
{129.146.53.216:2000 {{{trapAddress 129.146.53.216 } {trapOID
1.3.6.1.4.1.42.2.12.2.0.[12]}
{filter-exp {trapAddress && trapOID}}}}}
```

SNMP SET Command

The following are the SNMP set commands for above examples:

Trap subscription in Sun Management Center Trap Handler (running on 129.146.53.216:162) for statusChange trap:

```
snmpset -h 129.146.53.216 -p 162 -c community \
1.3.6.1.4.1.42.2.12.2.1.4.1.0  "OctetString" \
"{129.146.53.216:2000 \
{{{trapOID 1.3.6.1.4.1.42.2.12.2.0.1}}}}"
```

Trap subscription in Sun Management Center Trap Handler (running on 129.146.53.216:162) for statusChange, valueRefresh, moduleLoaded, moduleUnloaded traps.

```
snmpset -h 129.146.53.216 -p 162 -c community \
1.3.6.1.4.1.42.2.12.2.1.4.1.0 OctetString "{129.146.53.216:2000 \
{{{trapOID 1.3.6.1.4.1.42.2.12.2.0.[1245]}}}}"
```

Trap subscription in Sun Management Center Trap Handler (running on 129.146.53.216:162) for statusChange and valueRefresh traps from agent running on 129.146.53.216.

```
snmpset -h 129.146.53.216 -p 162 -c community \
1.3.6.1.4.1.42.2.12.2.1.4.1.0  "OctetString" \
"{129.146.53.216:2000 \
{{{trapAddress 129.146.53.216 } {trapOID \
1.3.6.1.4.1.42.2.12.2.0.[12]} \
{filter-exp {trapAddress && trapOID}}}}}"
```

Note – All of the preceding `snmpset` commands will work if the Trap Handler is configured for no authentication. Otherwise, these sets have to be valid `SNMPV2USEC SET` Commands.

Adding Jobs

The `jobAdder` node is used to incrementally add IP addresses to the `taglist` criteria of an existing trap subscription. This node is located at:

```
iso.org.dod.internet.private.enterprises...base.trapForward.jobAdder
```

For the SUN enterprise MIB, the corresponding url for this node is:

```
snmp://<host>:<port>/sym//iso/org/dod/internet/private/enterprises/
sun/prod/sunsymon/agent/base/trapForward/jobAdder#0
```

To add jobs to an existing subscription for traps from a Sun Management Center agent component, perform an SNMP set of a trap subscription specification into this node. The job adder specification has the following format:

```
<subscriber address>  <list of IP addresses>
```

where:

`subscriber address` is the IP address:port of the subscriber

`list of IP address` is one or more IP addresses to be added to the taglist.

Removing Jobs

The `jobRemover` node is used to remove IP address entries from the taglist criteria of an existing trap subscription. This node is located at:

```
iso.org.dod.internet.private.enterprises...base.trapForward.jobRemover
```

For the SUN enterprise MIB, the corresponding URL for this node is:

```
snmp://<host>:<port>/sym//iso/org/dod/internet/private/enterprises/
sun/prod/sunsymon/agent/base/trapForward/jobRemover#0
```

To remove jobs from an existing subscription for SNMP traps from a Sun Management Center agent component, perform an SNMP set of a trap subscription specification into this node. The job remover specification has the following format:

```
<subscriber address>  <list of IP addresses>
```

where:

`subscriber address` is the IP address:port of the subscriber

`list of IP address` is one or more IP addresses to be removed from the taglist.

Sun Management Center Enterprise Specific Traps

This section provides the MIB for enterprise specific traps generated by Sun Management Center agent.

CODE EXAMPLE G-1 Sun Management Center Enterprise Specific Traps

```
base OBJECT IDENTIFIER ::= { * 1 }
traps OBJECT IDENTIFIER ::= { base 0 }
trapInfo OBJECT IDENTIFIER ::= { base 3 }

statusChange NOTIFICATION-TYPE
OBJECTS { statusOID }
STATUS current
DESCRIPTION
"A statusChange trap signifies that the status of an object has
changed."
::= { traps 1 }
valueRefresh NOTIFICATION-TYPE
OBJECTS { refreshOID }
STATUS current
DESCRIPTION
```

CODE EXAMPLE G-1 Sun Management Center Enterprise Specific Traps

```
"A valueRefresh trap signifies that the value of an object has been
manually refreshed."
::= { traps 2 }

event NOTIFICATION-TYPE
OBJECTS { eventInfo }
STATUS current
DESCRIPTION
"An event trap signifies that an event has been detected and logged
by the monitoring software."
::= { traps 3 }

moduleLoaded NOTIFICATION-TYPE
OBJECTS { version }
STATUS current
DESCRIPTION
"A moduleLoaded trap signifies that a module has been loaded."
::= { traps 4 }

moduleUnloaded NOTIFICATION-TYPE
OBJECTS { version }
STATUS current
DESCRIPTION
"A moduleUnloaded trap signifies that a module has been unloaded."
::= { traps 5 }

userConfig NOTIFICATION-TYPE
OBJECTS { userConfig }
STATUS current
DESCRIPTION
"A userConfig trap signifies that the sender of the trap is
requesting user configuration information."
::= { traps 6 }
trapClient NOTIFICATION-TYPE
STATUS current
DESCRIPTION
"A trapClient trap signifies that the sender of the trap is
requesting that traps be forwarded to it."
::= { traps 7 }

userUpdate NOTIFICATION-TYPE
STATUS current
DESCRIPTION
```

CODE EXAMPLE G-1 Sun Management Center Enterprise Specific Traps

```
"A userUpdate trap signifies that the user configuration
information has changed. It is employed by a master agent to inform
its subagents to reload its user configuration data."
::= { traps 8 }

statusOID OBJECT-TYPE
SYNTAX OBJECT IDENTIFIER
MAX-ACCESS accessible-for-notify
STATUS current
DESCRIPTION
"The identification of the object for which the status changed.
This occurs as the first trap-specific varbind in a
statusChangeTrap."
::= { trapInfo 1 }

refreshOID OBJECT-TYPE
SYNTAX OBJECT IDENTIFIER
MAX-ACCESS accessible-for-notify
STATUS current
DESCRIPTION
"The identification of the object for which the value was refreshed
changed. This occurs as the first
trap-specific varbind in a valueRefreshTrap."
::= { trapInfo 2 }

eventInfo OBJECT-TYPE
SYNTAX OCTET STRING
MAX-ACCESS accessible-for-notify
STATUS current
DESCRIPTION
"The event message of the object for which an event was detected.
This occurs as the first trap-specific
varbind in an eventTrap."
::= { trapInfo 3 }

userConfig OBJECT-TYPE
SYNTAX OCTET STRING
MAX-ACCESS accessible-for-notify
STATUS current
DESCRIPTION
"The snmp engine id and the usmUserSpinLock value of the  snmp
entity requesting user configuration information. This occurs as
the first trap-specific varbind in a userConfigTrap."
::= { trapInfo 4 }
```

SNMP Trap Subscription Support

Sun Management Center agent components (including the Trap Handler) support SNMP trap subscription. Trap subscription allows interested parties to request selected SNMP traps be forwarded to them. Sun Management Center agent components can also subscribe for traps.

Every Sun Management Center agent component supports a MIB that contains the clientRegistrar node. The clientRegistrar node is located at:

```
iso.org.dod.internet.private.enterprises...base.trapForward.clientRegistra
```

For the SUN enterprise MIB, the corresponding url for this node is:

```
snmp://<host>:<port>/sym//iso/org/dod/internet/private/enterprises\
/sun/prod/sunsymon/agent/base/trapForward/clientRegistrar#0
```

To subscribe for SNMP traps from a Sun Management Center agent component, one performs an SNMP set of a trap subscription specification into the clientRegistrar node. The trap subscription specification has the following format:

```
{<ipAddress>:<snmpPort> {{{<spec1} {<spec2>} ...}}}
```

where:

ipAddress is the IP Address of the trap subscriber

snmpPort is the SNMP port of the trap subscriber

specN is the trap filter criteria, filter criteria expressions, or subscription expiry specification

Trap filter criteria are specified using the following format:

```
{<criteria> <regexp>}
```

where:

criteria is the trap filter criteria. These criteria are mapped to the contents of the trap PDU.

regexp is the regular expression for the corresponding criteria

Possible criteria values are:

trapAddress - IP address of trap originator

trapOID - trap object identifier

oidN - Nth OID in trap varbind

valueN - Nth value in the trap varbind

By default, if multiple trap filter criteria are specified, they are OR'd together. To modify this behavior, filter criteria expressions can be used.

Filter criteria expressions are specified as follows:

```
{filter-exp {%<criteria1> <logical operator> %<criteria2> ...}}
```

where:

criteriaN is one of the filter criteria specified above

logical operator can be a logical AND '&&' or logical OR '||'

Subscription expiry is specified as follows:

```
{expiry <seconds>}
```

where:

seconds is the number of seconds before subscription is cancelled. For a subscription that does not expire, 0 can be specified. If the expiry is not specified, it defaults to ~46 days.

For example, if your process is on host:port 204.225.247.123:162 and you wish to subscribe for linkDown (whose trap OID is 1.3.6.1.6.3.1.1.5.3) traps from any agent, set the following trap subscription spec:

```
{204.225.247.123:162 {{{trapOID 1.3.6.1.6.3.1.1.5.3}}}}
```

Similarly, to subscribe for all linkDown (1.3.6.1.6.3.1.1.5.3) and linkUp (1.3.6.1.6.3.1.1.5.4) traps, set the following trap subscription spec:

```
{204.225.247.123:162 {{{trapOID ^1\\.3\\.6\\.1\\.6\\.3\\.1\\.1\\.5\\.[34]}}}}
```

Note – The regular expression includes double backslashes '\ \' for the dots '.' since one set is removed when the expression is processed by the Tcl procedure that processes the trap subscription request.

To subscribe for all linkDown traps originating from a specific agent on host:port 204.225.247.100:161, set the following trap subscription spec:

```
{204.225.247.123:162 {{{trapAddress 204.225.247.100} {trapOID\
1.3.6.1.6.3.1.1.5.3} {filter-exp {trapAddress && trapOID}}}}}
```

Glossary

3-tier architecture The Sun Management Center server stands between the Sun Management Center console on one end and the Sun Management Center agents on the other.

Sun Management Center agents provide the data required for manageability.

Sun Management Center console components provide the system monitoring, control, and configuration user interfaces.

The Sun Management Center server acts as a request broker between the agent and the console.

ACE Authentication, Compression, Encryption. The ACE layer provides authentication, compression, and encryption functions for communication across the interface.

ACLs Access Control Lists. The Sun Management Center agent MIB supports the specification of multiple levels of SNMP read or write access controls. These access control (ACL) specifications define the minimum security level required of users and/or groups to perform SNMP read or write operations on objects in the MIB.

active nodes An active node is a managed object or managed property that has refresh information associated with.

agent A software process, usually specific to a particular local managed host, that carries out manager requests and makes local system and application information available to remote users.

agent MIB The MIB that corresponds to a specific agent.

alarm An abnormal event, which may be indicative of current or impending problems, is detected by a Sun Management Center agent. The agent passes information about the abnormal event to the Sun Management Center server. The server passes this information on to the user as an alarm when the abnormal event matches a predefined alarm threshold.

alarm acknowledgment Sun Management Center users can acknowledge alarms indicating that the alarm does not represent a serious problem or that the problem is being resolved. Acknowledged alarms take a lower priority than unacknowledged alarms.

alarm file The Alarm File defines information used by alarm checks performed on managed properties. This file is loaded by the Agent File.

API Application Programming Interface. Examples are alarm API, authenticate API, data API, and so forth.

Attribute Editor A window that provides information about the selected object. In addition, the Attribute Editor in the Sun Management Center software enables you to customize various monitoring criteria for that object. The monitoring criteria are dependent on the type of object. There are Attribute Editors for domains, hosts, modules, and data properties.

Base-modules.dat file This file contains three module entries: `mib2-system`, `agent-stats`, and `fscan+syslog`.

Bourne shell service Essentially an object maintaining a pipe to one or more shell processes to which commands can be directed and the results returned asynchronously.

check operations Provides a mechanism for triggering refresh operations based on some criteria tested by the check operation.

community A string similar to a password that is used to authenticate access to an agent's monitored data.

complex alarm A complex alarm is based on a set of conditions becoming true. Unlike simple alarms, you cannot set thresholds for complex alarms. See also *simple alarm*.

console window A graphical user interface component of Sun Management Center software based on Java technology that is used to view monitored hosts (and managed objects) information and status and to interact with Sun Management Center agents.

DAQ Data aquisition.

data cascade The dissemination of a buffer of data into a tree of managed objects and managed properties is known as the *data cascade.*

derived nodes Derived nodes establish dependency relationships with the nodes on which they rely through the use of the refresh triggers specification.

digitalFilter This function provides a multiply and accumulate function to provide digital filtering capabilities.

discovery A Sun Management Center tool available from the main console window that is used to find hosts, routers, networks, and Simple Network Management Protocol (SNMP) devices that can be reached from the Sun Management Center server.

domain An arbitrary collection of hosts and networks that are monitored by the software as a single hierarchal entity. Users may choose to divide their enterprise into several domains, each to be managed by different users.

dynamic loadable modules A Sun Management Center agent module that can be loaded or unloaded at runtime, enabling monitored properties to be displayed on the main console window without having to restart the console or agent.

event An occurrence that triggers a change in the state of a managed object.

enterprise module parameter The enterprise module parameter is used to specify the OIDs file in which the location of the module is defined.

file scanning The act of scanning a file (usually a log file) for certain patterns (regular expressions) that may be indicative of problems or significant information. Sun Management Center agents use file scanning to assist in the monitoring of systems and applications when these components do not provide direct access to status information.

filter file These filters are used to extract the pertinent information from the raw results of data acquisition commands.

fileFilter The filter file defines data filters implemented with Tcl/TOE procedures. These filters are used to extract the pertinent information from the raw results of data acquisition commands.

hardware modules These modules mamage hardware for the host on which the agent is running. For example boards, SIMMs, and so on.

hierarchy view A window view that defines objects in a hierarchy or tree relationship to one another. Objects are grouped depending on the rank of the object in the hierarchy.

initHoldoff This qualifier specifies the time, in time specification, to wait before running the refresh command for the first time.

initInterval This qualifier specifies, in time specification, the time window within which the node should run the refresh command for the first time after the module initializes.

instance A single word or alpha-character string that is used internally within the Sun Management Center agent to identify uniquely a particular module or a row within a module.

internal service The internal service should be specified when the refresh command is a Tcl/TOE command or procedure to be executed in the current node's context.

localization/internationalization Sun Management Center consoles and associated GUI clients operate in a global environment. To do this, a mechanism is required to isolate the language dependent code/information from the language independent code and provide a straightforward method for graphical developers to reference the language dependent information.

managed entities The physical and logical components of a system that are being managed. For example disks, boards, hosts and networks.

managed object table primitive This branch primitive is used in conjunction with the `MANAGED-OBJECT-TABLE-ENTRY` primitive when constructing a managed object with a table of managed properties

manage In Sun Management Center software, *manage* is defined as being able to monitor, as well as manipulate an object. For example, management privileges include acknowledging and closing alarms, loading and unloading modules, changing alarm thresholds, and so on. Management privileges are similar to read, write, and execute access.

managed object classes Building blocks used to model managed entities.

mandatory parameters All modules must specify the standard set of parameters.

managed object primitive A primitive used by managed object nodes that are branch nodes in the object tree.

managed property classes These classes are used to group together related managed properties of one managed object .

managed property class This primitive is used to group related managed properties of a managed object together.

MEL pg 396

MIB Management Information Base. A MIB is a hierarchical database schema describing the data available from an agent. The MIB is used by Sun Management Center agents to store monitored data that can be accessed remotely.

MIB node service This type of service should be used when the refresh command is to be executed in the context of another MIB node.

model file Defines the building blocks used to monitor an entity to be managed.

module A software component that can be loaded dynamically to monitor data resources of systems, applications and network devices.

monitor In Sun Management Center software, monitor is defined as being able to observe an object, view alarms and properties. Monitoring privileges are similar to read-only access.

nested managed object This managed object contains other managed objects.

node A node is a workstation or server.

object A particular resource (computer host, network interface, software process, and so on), which is subject to monitoring or management by Sun Management Center software. A managed object is one that you can manipulate. For example, you can acknowledge and turn off an alarm condition for an object that you can manage. A monitored object is one that you can observe but not acknowledge or otherwise manage.

object manager primitive This primitive is used to identify the start of a subtree in the hierarchy where the contents of the subtree may change and must be discovered dynamically.

operating system modules These modules manage operating system entities for the host on which the agent is running. For example swap, cpu usage, and so on.

optional parameters These additional parameters are specified in the module's Parameter File to facilitate user input for the requisite information when the module is loaded.

pctFilter This function computes the value of a named managed property as a percentage of another managed property.

passive nodes Nodes that do not actively collect data but instead have data cascaded into them.

Parameter File Specifies the parameters which the module requires when it is loaded by the agent.

Prevalidate Actions The purpose of prevalidate actions is to ensure that the value can be set into the node

postrowActions These actions are triggered to execute after the set but before the postvalidate actions. command, 181

Postvalidate Actions Post-validate actions can be specified to validate the set value.

procedure file Objects in the MIB tree may need to perform special data acquisition functions or alarm status actions.This provides a simple mechanism to override or extend the functionality of the core MIB object primitives.

rCompareRule This rule performs numeric comparisons, regular expression checks, or string comparisons.

rateFilter This function accepts the name of a managed property and returns the rate of change per second for the managed property since the previous sample.

reference node Reference nodes are objects that are loaded for use as a template in the model file.

refreshFilter The refreshFilter qualifier specifies a Tcl command or procedure that is used to process the data acquired by the refresh command.

refreshMode Qualifier The refreshMode qualifier specifies the execution mode of the refresh command.

refresh **operation** The refresh operation consists of performing DAQ and disseminating the acquired data into the appropriate managed property nodes.

refreshParams The refreshParams qualifier can be used to specify arguments to be passed to the refresh command.

refresh service A refresh service is an object within the agent that can be used for the purposes of data acquisition.

refreshTrigger Derived nodes establish dependency relationships with one or more nodes on which they rely on through the use of the refresh triggers specification.

remote modules Capable of managing entities on remote hosts. For example Oracle, Sybase, and so on.

remote server context A remote server context refers to a collection of Sun Management Center agents and a particular server layer with which the remote agents are associated.

request caching The Sun Management Center server consolidates duplicate outstanding requests originating from multiple consoles and eliminates the execution of redundant requests.

rollbackActions The purpose of rollback actions is to restore the state of the object after the failed set.

RMI Remote Method Invocation.

rule A rule is an alarm check mechanism that allows for complex or special purpose logic in determining the status of a monitored host or node.

seed The password for the Sun Management Center user group called `esmaster`. The seed is an alpha-numeric string of up to 8 characters. (This is not necessarily a UNIX password.) You can select your own seed, or accept the default seed (`maplesyr`) provided by the Sun Management Center software. If you select your own seed, be sure to record it for later reference.

shadow MIB supports SNMP access to attributes associated with the managed objects and properties in the agent MIB

superior service The superior service should be specified when the refresh command is a Tcl/TOE command or procedure to be executed in the context of the current node's superior in the tree hierarchy.

server The collection of programs and processes (SNMP-based trap, event, topology, configuration, and Java server) that work on behalf of a Sun Management Center user to help manage a particular set of networks, hosts and devices. Usually sends requests to Sun Management Center agents, accepts collected data from them, and passes the data to the main console window for display.

server context See "remote server context."

setActions The setActions specification defines one or more actions to execute when the value of the object is set via SNMP.

simple alarm Simple alarms are based on one condition becoming true. You may set alarm thresholds for simple alarms.

SNMP Simple Network Management Protocol. A complex protocol designed to allow networked entities (hosts, routers, and so on) to exchange monitoring information.

SNMP Service The SNMP service should be specified when the refresh command is an SNMP get request for acquiring data from another SNMP agent.

SNMPv2 usec SNMP version 2, user-based security model security standards.

Sun Management Center superuser Sun Management Center superuser is a valid user on a server host. The superuser decides what the agents are in the context of the server. By default, the superuser password is used as a seed for security key generation.

Sun Management Center user Sun Management Center users are the members of the `symon` group in the `/etc/group` file.

tableRateFilter This function is similar to rateFilter function, except that it operates on a list of data instead of a scalar.

Tcl Tool Command Language.

Time A time specification format that permits the entry of complex time specifications, including time windows, specific points in time, and time intervals.

timeoutInterval If the refresh command does not complete within the specified time out interval, then the command will be aborted.

transposeFilter A useful data filter is the `transposeFilter`, which can be used to transpose a table of data

TOE Tcl Object Extension.

tooltip Proxy monitoring.

topology view The topology view displays the members of the object selected in the hierarchy view.

transposeFilter TransposeFilter can be used to transpose a table of data.

updateFilter The update filter specifies a Tcl command or procedure that is used to process the data being cascaded into the passive node.

URL Uniform Resource Locator. A URL is a textual specification describing a resource which is network-accessible.

userFilter Loops through each line to determine the console user and count the number of unique users and sessions.

.x file A Sun Management Center file used to represent TOE objects.

Index

NUMERICS

A

B

C

D

E

F

G

H

I

N

O

P

Q

R

S

T

U

V

W

X