VERITAS™

VERITAS Cluster Server 4.1

Agent Developer’s Guide

HP-UX

N12189G

June 2005

Disclaimer

The information contained in this publication is subject to change without notice. VERITAS Software Corporation makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. VERITAS Software Corporation shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this manual.

VERITAS Legal Notice

Copyright © 1998-2004 VERITAS Software Corporation. All rights reserved. VERITAS, the VERITAS logo, VERITAS Cluster Server, and all other VERITAS product names and slogans are trademarks or registered trademarks of VERITAS Software Corporation. VERITAS and the VERITAS logo, Reg. U.S. Pat. & Tm. Off. Other product names and/or slogans mentioned herein may be trademarks or registered trademarks of their respective companies.

VERITAS Software Corporation
350 Ellis Street
Mountain View, CA 94043
USA
Phone 650–527–8000 Fax 650–527–2908
www.veritas.com

Third-Party Copyrights

Apache Software

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

The Apache Software License, Version 1.1

Copyright (c) 1999 The Apache Software Foundation. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. The end-user documentation included with the redistribution, if any, must include the following acknowledgement:

This product includes software developed by the Apache Software Foundation (http://www.apache.org/).

Alternately, this acknowledgement may appear in the software itself, if and wherever such third-party acknowledgements normally appear.

4. The names "The Jakarta Project", "Tomcat", and "Apache Software Foundation" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact apache@apache.org.

5. Products derived from this software may not be called "Apache" nor may "Apache" appear in their names without prior written permission of the Apache Group.

THIS SOFTWARE IS PROVIDED "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE APACHE SOFTWARE FOUNDATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This software consists of voluntary contributions made by many individuals on behalf of the Apache Software Foundation. For more information on the Apache Software Foundation, please see http://www.apache.org/.

Data Encryption Standard (DES)

Support for data encryption in VCS is based on the MIT Data Encryption Standard (DES) under the following copyright:

Copyright © 1990 Dennis Ferguson. All rights reserved.

Commercial use is permitted only if products that are derived from or include this software are made available for purchase and/or use in Canada. Otherwise, redistribution and use in source and binary forms are permitted.

Copyright 1985, 1986, 1987, 1988, 1990 by the Massachusetts Institute of Technology. All rights reserved.

Export of this software from the United States of America may require a specific license from the United States Government. It is the responsibility of any person or organization contemplating export to obtain such a license before exporting.

WITHIN THAT CONSTRAINT, permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of M.I.T. not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. M.I.T. makes no representations about the suitability of this software for any purpose. It is provided as is without express or implied warranty.

SNMP Software

SNMP support in VCS is based on CMU SNMP v2 under the following copyright:

Copyright 1989, 1991, 1992 by Carnegie Mellon University

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of CMU not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

CMU DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL CMU BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Contents

Preface xiii

What's In This Guide? xiii

Conventions xiv

Getting Help xv

The *VERITAS Enabled*™ Program xv

Documentation Feedback xv

Chapter 1. Introduction 1

VCS Agents: An Overview 1

How Agents Work 2

The Agent Framework 2

Resource Type Definitions 2

Entry Points 2

Developing an Agent: Overview 3

Applications Considerations 3

Creating an Agent: Highlights 4

Create the Resource Type Definition 4

Decide to Use C++ or Scripts to Implement the Agent 4

Create the Entry Points 4

Test the Agent 4

Resource Type Definitions 5

Example Resource Type Definition: FileOnOff Resource 5

The FileOnOff Resource: an Example in the main.cf File 6

How the FileOnOff Agent Uses Configuration Information 6

Example Script Entry Points for the FileOnOff Resource 7
Online Entry Point for FileOnOff 7
Monitor Entry Point for FileOnOff 8
Offline Entry Point for FileOnOff 9
Types of Resources and Agent Entry Points They Require 10
The Attributes of Resources and Resource Types 10
Categories of Attributes 11
Attribute Data Types and Dimensions 12
Attribute Data Types 12
Attribute Dimensions 13

Chapter 2. Agent Entry Points 15
Using C++ or Script Entry Points 15
C++ Agents 16
Script Agents 16
VCSAgStartup 16
Implementing All or Some of the Entry Points in C++ 16
Implementing Entry Points Using Scripts 16
Sample Structure 17
Example: Modifying VCSAgStartup for C++ and Script Entry Points 18
Agent Entry Points 19
monitor 20
info 21
Return Values 21
online 22
offline 22
clean 23
action 25
Return Values 25
ActionTimeout 25

attr_changed 26
open 26
close 26
shutdown 26
Summary of Return Values for Entry Points 27
Agent Information File 28
Example Agent Information File 28
Agent Information 29
Attribute Argument Details 29

Chapter 3. Implementing Entry Points Using C++ 31
Data Structures 31
ArgList Attribute 33
C++ Entry Point Syntax 35
VCSAgStartup 35
monitor 36
info 37
resinfo_op 37
info_output 37
opt_update_args 37
opt_add_args 38
Example, Info Entry Point Implementation in C++ 39
online 41
offline 42
clean 43
action 44
attr_changed 46
open 48
close 49
shutdown 50

VCS Primitives . . . 51
VCSAgRegisterEPStruct . . . 51
VCSAgSetCookie . . . 52
VCSAgRegister . . . 54
VCSAgUnregister . . . 55
VCSAgGetCookie . . . 56
VCSAgEncodeString . . . 58
Example: VCSAgEncodeString . . . 59
VCSAgStrlcpy . . . 60
VCSAgStrlcat . . . 60
VCSAgSnprintf . . . 60

Chapter 4. Implementing Entry Points Using Scripts . . . 61
ArgList Attributes . . . 62
Script Entry Point Syntax . . . 63
monitor . . . 63
online . . . 63
offline . . . 63
clean . . . 64
action . . . 64
attr_changed . . . 65
info . . . 65
Example Script Implementation of the Info Entry Point . . . 65
open . . . 67
close . . . 68
shutdown . . . 68

Chapter 5. Logging Agent Messages . . . 69
Logging in C++ and Script-based Entry Points . . . 69
VCS Agent Messages: Format . . . 69
Timestamp . . . 70

Mnemonic 70
Severity 70
UMI 70
Message Text 71
C++ Agent Logging APIs 71
Agent Application Logging Macros for C++ Entry Points 72
Agent Debug Logging Macros for C++ Entry Points 73
Severity Arguments for C++ Macros 74
Initializing function_name Using VCSAG_LOG_INIT 75
Log Category 76
Examples of Logging APIs Used in a C++ Agent 77
Script Entry Point Logging Functions 80
VCSAG_SET_ENVS 81
VCSAG_SET_ENVS Examples, Shell Script Entry Points 82
VCSAG_SET_ENVS Examples, Perl Script Entry Points 82
VCSAG_LOG_MSG 83
VCSAG_LOG_MSG Examples, Shell Script Entry Points 83
VCSAG_LOG_MSG Examples, Perl Script Entry Points 84
VCSAG_LOGDBG_MSG 84
VCSAG_LOGDBG_MSG Examples, Shell Script Entry Points 85
VCSAG_LOGDBG_MSG Examples, Perl Script Entry Points 85
Using the Functions in Scripts 85
Example of Logging Functions Used in Script Agent 86

Chapter 6. Building a Custom VCS Agent 87
Building a VCS Agent for FileOnOff Resources 89
Using Script Entry Points 89
Using VCSAgStartup() and Script Entry Points 90
Using C++ and Script Entry Points 92
Using C++ Entry Points 94

Chapter 7. Setting Agent Attributes 97

Overriding Static Attributes 97

Agent Attribute Definitions 98

ActionTimeout 98

AgentFile 98

AgentReplyTimeout 98

AgentStartTimeout 98

ArgList 99

ArgList Reference Attributes 99

AttrChangedTimeout 99

CleanTimeout 99

CloseTimeout 99

ComputeStats 99

ConfInterval 100

FaultOnMonitorTimeouts 101

FireDrill 101

InfoInterval 101

InfoTimeout 101

LogDbg 102

LogFileSize 103

ManageFaults 103

MonitorInterval 104

MonitorStatsParam 104

MonitorTimeout 104

NumThreads 105

OfflineMonitorInterval 105

OfflineTimeout 105

OnlineRetryLimit 105

OnlineTimeout 105

OnlineWaitLimit 106

OpenTimeout 106
Operations 106
ResourceInfo 107
RestartLimit 107
RegList 108
SupportedActions 109
ToleranceLimit 109
Scheduling Class and Priority Configuration Support 110
Priority Ranges 110
Default Scheduling Classes and Priorities 111
Attributes for Scheduling Class and Priorities 112
AgentClass 112
AgentPriority 112
ScriptClass 112
ScriptPriority 112
Initializing Attributes in the Configuration File 113
Setting Attributes Dynamically from the Command Line 113

Chapter 8. Testing VCS Agents 115
Using Debug Messages 115
Using the VCS Engine Process 116
Test Commands 116
Using the AgentServer Utility 117

Chapter 9. State Transition Diagrams 121

Chapter 10. VCS Internationalized Message Catalogs 137
Creating SMC Files 138
SMC Format 138
Example SMC File 138
Formatting SMC Files 139

Naming SMC Files, BMC Files 139
Message Examples 140
Using Format Specifiers 140
Converting SMC Files to BMC Files 141
Storing BMC Files 141
VCS Languages 141
Displaying the Contents of BMC Files 141
Using BMC Map Files 142
Location of BMC Map Files 142
Creating BMC Map Files 142
Example BMC Map File 142
Updating BMC Files 143

Appendix A. How to Use Pre-VCS 4.0 Agents 145
Guidelines for Using Pre-VCS 4.0 Agents 145
Log Messages in Pre-VCS 4.0 Agents 146
Mapping of Log Tags (Pre-VCS 4.0) to Log Severities (VCS 4.0) 146
How Pre-VCS 4.0 Messages are Displayed by VCS 4.0 and Later 147
Comparing Pre-VCS 4.0 APIs and VCS 4.0 Logging Macros 147
Pre-VCS 4.0 Message APIs 148
VCSAgLogConsoleMsg 148
VCSAgLogI18NMsg 149
VCSAgLogI18NMsgEx 150
VCSAgLogI18NConsoleMsg 151
VCSAgLogI18NConsoleMsgEx 152

Index 153

Preface

This guide describes the API provided by the VERITAS Cluster Server™ (VCS) agent framework. It explains how to build and test an agent on UNIX platforms.

Each VCS agent manages resources of a particular type within a highly available cluster environment. An agent typically brings resources online, takes resources offline, and monitors resources to determine their state.

For information on VCS4.0, and a brief overview of the features, see *VERITAS Cluster Server Release Notes*.

What's In This Guide?

Chapter 1. "Introduction" on page 1 introduces agent development.

Chapter 2. "Agent Entry Points" on page 15 provides a list of VCS entry points and explains how to implement them.

Chapter 3. "Implementing Entry Points Using C++" on page 31 describes how to implement the VCS entry points using C++. This chapter also describes the VCS agent primitives.

Chapter 4. "Implementing Entry Points Using Scripts" on page 61 describes how to implement the VCS entry points using scripts.

Chapter 5. "Logging Agent Messages" on page 69 describes how to include logging macros and functions within agents.

Chapter 6. "Building a Custom VCS Agent" on page 87 provides step-by-step instructions for building a custom agent using C++ and scripts.

Chapter 7. "Setting Agent Attributes" on page 97 describes each agent parameter and their default values.

Chapter 8. "Testing VCS Agents" on page 115 describes testing VCS agents using the AgentServer utility.

Chapter 9. "State Transition Diagrams" on page 121 presents a graphical representation of the dynamic state changes within the agent framework.

Chapter 10. "VCS Internationalized Message Catalogs" on page 137 describes the guidelines and tools for managing internationalized messages for custom agents.

Appendix A. "How to Use Pre-VCS 4.0 Agents" on page 145 describes the guidelines for using legacy agents with VCS 4.0.

Conventions

The following conventions apply throughout the documentation set.

Typographical Conventions

Convention	Description
`monospace`	computer output, files, attribute names, device names, and directories
`monospace`(bold)	user input and commands, keywords in grammar syntax
italic	new terms, titles, emphasis, variables
`italic`	variables within a command
%	C shell prompt
$	Bourne/Korn shell prompt
#	Superuser prompt (for all shells)

You should use the appropriate conventions for your platform. For example, when specifying a path, use backslashes on Microsoft Windows and slashes on UNIX. Significant differences between the platforms are noted in the text.

Getting Help

For technical assistance, visit http://support.veritas.com and select phone or email support. This site also provides access to resources such as TechNotes, product alerts, software downloads, hardware compatibility lists, and our customer email notification service. Use the Knowledge Base Search feature to access additional product information, including current and past releases of VERITAS documentation.

Additional Resources

For license information, software updates and sales contacts, visit https://my.veritas.com/productcenter/ContactVeritas.jsp. For information on purchasing product documentation, visit http://webstore.veritas.com.

The *VERITAS Enabled*™ Program

VERITAS has recently launched the VERITAS Enabled Program to help our partners maintain close integration with VERITAS products. As part of this program, VERITAS is working with a wide range of partners to create High Availability Agents for VERITAS Cluster Server. Some of these High Availability Agents are available from VERITAS and others are available from your application vendor. Visit www.VERITASEnabled.com for more information.

Documentation Feedback

Your feedback on product documentation is important to us. Send suggestions for improvements and reports on errors or omissions to `clusteringdocs@veritas.com`. Include the title and part number of the document (located in the lower left corner of the title page), and chapter and section titles of the text on which you are reporting. Our goal is to ensure customer satisfaction by providing effective, quality documentation. For assistance with topics other than documentation, visit http://support.veritas.com.

Introduction 1

This guide describes the API provided by the VERITAS Cluster Server™ (VCS) agent framework.

Note Custom agents, that is, agents developed outside of VERITAS, are not supported by VERITAS Technical Support.

VCS Agents: An Overview

Each VCS agent is a program that manages resources of a particular type, such as a disk group or an IP address, within a cluster environment. Each resource type requires an agent. The agent acts as an intermediary between VCS and the resource it manages, typically by bringing it online, monitoring its state, or taking it offline.

Agents packaged with VCS are referred to as *bundled agents*. Examples of bundled agents include Share, IP (Internet Protocol), and NIC (network interface card) agents. For more information on VCS bundled agents, including their attributes and modes of operation, see the

Agents packaged separately for use with VCS are referred to as *enterprise agents*. They include agents for Informix, Sybase, Oracle, and others. Contact your VERITAS sales representative for information on how to purchase these agents for your configuration.

For information on installing and configuring VCS, see the *VERITAS Cluster Server Installation Guide*.

How Agents Work

A single VCS agent can manage multiple resources of the same type on one host. For example, the NIC agent manages all NIC resources.

When the VCS engine process, had, comes up on a system, it automatically starts the agents required for the types of resources that are to be managed and provides the agents the specific configuration information for those resources. The agent carries out the commands from VCS to bring resources online and take them offline. The agents also periodically monitor the resources, updating VCS with their status. When an agent crashes or hangs, VCS detects the fact and restarts the agent.

Note The VCS engine process is known as "`had`." The acronym stands for "high-availability daemon."

The Agent Framework

The VCS agent framework is a set of predefined functions compiled into the agent for each resource type. These functions include the ability to connect to the VCS engine and to understand the common VCS configuration attributes, such as RestartLimit and MonitorInterval. When an agent's code is built in C++, the agent framework is compiled in with an include statement. When an agent is built using script languages, such as shell or Perl, the ScriptAgent provides the agent framework functions. The agent framework handles much of the complexity that should not concern the agent developer.

Resource Type Definitions

Each resource type requires a resource type definition. A resource type definition describes the information an agent needs to control resources of that type. The type definition can be considered similar to a header file in a C program. The type definition defines the data types of variables (attributes) to provide error checking and provides the order that the variables (attributes) are passed to the entry points.

Entry Points

An entry point is a section of code or a script used by the agent to carry out a specific function on a resource. The agent framework supports a specific set of entry points, each of which has a basic structure and a set of return values. Descriptions of each of the supported entry points begin with "Agent Entry Points" on page 19.

The agent developer implements entry points for a resource by providing the specific definitions required to manage the resource. For example, when implementing an agent's `online` entry point, the developer includes the command to start a resource; when implementing the `monitor` entry point, the developer includes the commands to check if the resource is online or not.

Developing an Agent: Overview

Before creating the agent, some considerations and planning are required, especially regarding the application for which the agent is created.

Applications Considerations

The application for which a VCS agent is developed must be capable of being controlled by the agent and be able to operate in a cluster environment. The following criteria describe an application that can successfully operate in a cluster:

- ✔ The application must be capable of being started by a specific command or set of commands. Specific commands must be available to start the application's external resources such as file systems and IP addresses.
- ✔ Each instance of an application must be capable of being stopped by a defined procedure. Other instances of the application must not be affected.
- ✔ The application must be capable of being stopped cleanly, by forcible means if necessary.
- ✔ Each instance of an application must be capable of being monitored. Monitoring can be simple or in-depth. Monitoring an application becomes more effective when the monitoring test resembles the actual activity of the application's user.
- ✔ The application must be capable of storing data on shared disks rather than locally or in memory, and each cluster system must be capable of accessing the data and all information required to run the application.
- ✔ The application must be crash-tolerant, that is, it must be capable of being run on a system that crashes and of being started on a failover node in a known state. This typically means that data is regularly written to shared storage rather than stored in memory.
- ✔ The application must be host-independent within a cluster; that is, there are no licensing requirements or host name dependencies that prevent successful failover.
- ✔ The application must run properly with other applications in the cluster.

Creating an Agent: Highlights

The steps to create and implement an agent are described by example in "Building a Custom VCS Agent" on page 87. Highlights of those steps are described here.

Create the Resource Type Definition

Create a file containing the resource type definition. Name the file `*ResourceType*Types.cf`. This file is referenced as an "include" statement in the VCS configuration file, `main.cf`. See "Resource Type Definitions" on page 5.

Decide to Use C++ or Scripts to Implement the Agent

Decide whether to implement the agent entry points using C++ code, scripts, or a combination of the two. There are advantages and disadvantages implementing entry points in either method. Refer to "Agent Entry Points" on page 15 and review the description of "Using C++ or Script Entry Points."

Create the Entry Points

The procedures and guidelines for creating the entry points are described in "Implementing Entry Points Using C++" on page 31 and "Implementing Entry Points Using Scripts" on page 61.

- Use the sample files in `$VCS_HOME/src/agent/Sample` to create a C++ agent or agent using entry points written in C++ and scripts. Build the agent binary. Place it in the directory `$VCS_HOME/bin/*resource_type*`.
- Use the ScriptAgent to build an agent using only script entry points.
- Install script entry point files in the directory `$VCS_HOME/bin/*resource_type*`.

Test the Agent

Test the agent by defining the resource type in a VCS configuration. See "Testing VCS Agents" on page 115.

Resource Type Definitions

The `types.cf` file contains definitions of standard VCS resource types. The example shown in the following paragraph is for a standard VCS resource type, FileOnOff. A custom resource type definition should be placed in a file called *`Resourcetype`*`Types.cf`.

Example Resource Type Definition: FileOnOff Resource

The FileOnOff agent is designed to manage simple files. Each FileOnOff resource manages one file. For example, when VCS wants to online the FileOnOff resource, the FileOnOff agent calls the `online` entry point to create a file of a specific name in a specific location. To monitor the FileOnOff resource, the agent calls the `monitor` entry point to verify the existence of the file. When VCS wants the FileOnOff resource taken offline, the agent calls the `offline` entry point to remove the file.

The following shows the definition for the FileOnOff resource type. It applies to all resources of the FileOnOff type:

```
type FileOnOff (
static str ArgList[] = { PathName }
str PathName
)
```

This definition is included in the VCS `types.cf` file. Note the following points about the FileOnOff type definition:

- The keyword “`type`” is followed by the name of the resource type, in this case, `FileOnOff`.
- The `ArgList` attribute includes the names of the attributes, listing them in the order they are sent to the entry points. In the case of this example, the FileOnOff resource type contains only one attribute, `PathName`, the pathname for the file.
- The `PathName` attribute is defined as a “`str`,” or string variable. The data type for each attribute must be defined.

The FileOnOff Resource: an Example in the main.cf File

In the VCS configuration file, `main.cf`, a specific resource of the FileOnOff resource type may resemble:

```
include types.cf
.
.
.
FileOnOff temp_file01 (
PathName = "/tmp/test"
)
```

The include statement at the beginning of the `main.cf` file names the `types.cf` file, which includes the FileOnOff resource type definition. The resource defined in the `main.cf` file specifies:

- The resource type: `FileOnOff`
- The unique name of the resource, `temp_file01`
- The value for the PathName attribute: "`/tmp/test`"

The agent creates a file "`test`" in the directory "`/tmp`."

How the FileOnOff Agent Uses Configuration Information

This information in the VCS configuration is passed by the engine to the FileOnOff agent when the agent starts up. The information passed to the agent includes: resource names, the corresponding resource attributes, and the values of the attributes for all of the resources of that type.

Thereafter, to bring the resource online, for example, VCS can provide the agent with the name of the entry point (`online`) and the name of the resource (`temp_file01`). The agent then calls the entry point and provides the resource name and values for the attributes in the `ArgList`. The entry point performs its tasks.

Example Script Entry Points for the FileOnOff Resource

The following example shows entry points written in a shell script.

Note The actual VCS FileOnOff entry points are written in C++, but for this example, shell script is used.

Online Entry Point for FileOnOff

The FileOnOff example entry point is simple. When the agent's `online` entry point is called by the agent, the entry point expects the name of the resource as the first argument and the value of PathName attribute as the second argument. It then creates the file in the specified path. For our specific FileOnOff example, the file `/tmp/test` would be created.

```
#!/bin/sh
# FileOnOff Online script
# Expects ResourceName and PathName
#
VCSHOME="${VCS_HOME:-/opt/VRTSvcs}"
. $VCSHOME/bin/ag_i18n_inc.sh
RESNAME=$1
VCSAG_SET_ENVS $RESNAME
#check if second attribute provided
if [ -z "$2" ]
then
   VCSAG_LOG_MSG "W" "The value for PathName is not specified”
   1020
else
   #Create the file
   touch $2
fi
exit 0;

# No need for exit code. Shell returns 0 if successful
# and 1 if not. Monitor will be called in either case.
# exit code indicates the number of seconds VCS should wait,
# after online entry point completes, before calling the monitor
# entry point to check the resource state.
```

Monitor Entry Point for FileOnOff

The example FileOnOff `monitor` entry point accepts the name of the resource and the same `ArgList` as the first and second arguments and checks if the file exists. If the file exists it returns exit code 110 for online. If the file does not exist the monitor returns 100 for offline. If the state of the file cannot be determined, the monitor returns 99.

```
#!/bin/sh
# FileOnOff Monitor script
# Expects Resource Name and Pathname
#
VCSHOME="${VCS_HOME:-/opt/VRTSvcs}"
. $VCSHOME/bin/ag_i18n_inc.sh
RESNAME=$1
VCSAG_SET_ENVS $RESNAME
#check if second attribute provided
#Exit with unknown and log error if not provided.
if [ -z "$2" ]
then
   VCSAG_LOG_MSG "W" "The value for PathName is not specified”
   1020
   exit 99
else
   if [ -f $2 ]; then exit 110;
   # Exit online (110) if file exists
   # Exit offline (100) if file does not exist
   else exit 100;
   fi
fi
```

Offline Entry Point for FileOnOff

The example FileOnOff `offline` entry point accepts the name of the resource and the same `ArgList` attribute values and deletes the file specified by `PathName`.

```
#!/bin/sh
# FileOnOff Offline script
# Expects ResourceName and Pathname
#
VCSHOME="${VCS_HOME:-/opt/VRTSvcs}"
. $VCSHOME/bin/ag_i18n_inc.sh
RESNAME=$1
VCSAG_SET_ENVS $RESNAME
#check if second attribute provided
if [ -z "$2" ]
then
   VCSAG_LOG_MSG "W" "The value for PathName is not specified”
   1020
else
#remove the file
   /bin/rm -f $2
fi
exit 0;

# No need for exit code, as shell returns 0 if successful
# and 1 if not. Monitor will be called in either case.
# Similar to online - exit code indicates how long VCS should
# wait, after offline completes, before calling monitor for the
# resource.
```

Types of Resources and Agent Entry Points They Require

Different types of resources require different types of control, requiring implementation of different entry points. Resources can be classified as OnOff, OnOnly, or Persistent, depending on the entry points required to control them.

- OnOff resources

  Most resources are OnOff, meaning VCS starts and stops them as required. For example, VCS assigns the IP address to the specified NIC and removes the assigned IP address when the associated service group is taken offline. Another example is the DiskGroup resource. VCS imports a disk group when needed and deports it when it is no longer needed. For agents of OnOff resources, all entry points can be implemented.

- OnOnly resources

  An OnOnly resource is brought online when required by VCS, but it is not taken offline when the associated service group is taken offline. For example, in the case of the FileOnOnly resource, VCS creates the specified file if required, but does not delete the file if the associated service group is taken offline. For agents of OnOnly resources, `online` and `monitor` entry points can be implemented.

- Persistent resources

  A Persistent resource cannot be brought online or taken offline, yet VCS requires the resource to be present in the configuration. For example, a NIC resource cannot be started or stopped, but it is required to configure an IP address. VCS monitors Persistent resources to ensure their status and operation. An agent for a Persistent resource requires the monitor entry point.

For all these types of resources, `info` and `action` entry points can be implemented.

The Attributes of Resources and Resource Types

Resources are configured and controlled by defining their attributes and assigning values to these attributes. The attributes assigned to VCS resources can be categorized by the scope of their control within the cluster. Some attributes only configure a specific resource, some affect the behavior of a resource type, and some are applicable to all resource types.

Categories of Attributes

- Resource-specific attributes.

  An attribute that can be defined for a specific resource only is resource-specific. Examples include the PathName attribute for the FileOnOff resource and the MountPoint attribute for the Mount resource. Resource-specific attributes are set in the `main.cf` file.

- Type-dependent attributes:

  When attributes can be defined only for resources of a specific type, they are type-dependent. An example would be the StartVolumes and Stop Volumes attributes of the VCS DiskGroup resource type. All resources of the type DiskGroup have the default values for these attributes. For example:

  ```
  type DiskGroup (
          :
          int StartVolumes = 1
          int StopVolumes = 1
          :
          )
  ```

  Type-dependent attributes can be static and non-static. Static attributes are typically defined in the `types.cf` file, identified as static. For example:

  ```
  type FileOnOff (
  static str ArgList[] = { PathName }
  .
  .
  )
  ```

  Static resource type attributes can be overridden. See “Overriding Static Attributes” on page 97.

- Type-independent attributes:

  An attribute that can be defined for resources regardless of their type is a type-independent type. An example might be the MonitorInterval attribute. These attributes are defined in the agent framework when the agent is developed.

  The value of a type-independent attribute can be set for a give type. For example, the default value of MonitorInterval is 60 seconds, but it can be set for a specific resource type in the `types.cf` file.

  ```
  type FileOnOff (
  static str ArgList[] = { PathName }
   str PathName
  static int MonitorInterval=30
  )
  ```

- Global and local attributes:

  An attribute whose value applies for the resource on all systems is global. The values of a resource's attribute can be set to have a local scope, that is, apply to specific systems. In the following example of the MultiNICA resource type, attributes applying locally are indicated by "@system" following the attribute name:

```
MultiNICA mnic (
Device@sysa = { le0 = "166.98.16.103", qfe3 = "166.98.16.103" }
Device@sysb = { le0 = "166.98.16.104", qfe3 = "166.98.16.104" }
NetMask = "255.255.255.0"
ArpDelay = 5
Options = "trailers"
RouteOptions@sysa = "default 166.98.16.103 0"
RouteOptions@sysb = "default 166.98.16.104 0"
)
```

- Temp attributes

  Temp attributes are maintained by VCS only at run time. Their values are not dumped to disk and hence, are lost when VCS is stopped and restarted. Refer to the *VERITAS Cluster Server User's Guide* for information about temp attributes.

Attribute Data Types and Dimensions

Attributes contain data regarding the cluster, systems, service groups, resources, resource types, agents, and heartbeats if the Global Cluster option is used (refer to the *VERITAS Cluster Server User's Guide* for information about the Global Cluster option).

Attribute Data Types

- String

  A string is a sequence of characters enclosed by double quotes. A string may also contain double quotes, but the quotes must be immediately preceded by a backslash character. A backslash is represented in a string as `\\`.

  Quotes are not required if a string begins with a letter, and contains only letters, numbers, dashes (-), and underscores (_). For example, a string defining a network interface such as `hme0` does not require quotes as it contains only letters and numbers. However a string defining an IP address requires quotes, such as: "192.168.100.1" because the IP contains periods.

```
str Address
```

- Integer

  Signed integer constants are a sequence of digits from 0 to 9. They may be preceded by a dash, and are interpreted in base 10. Integers cannot exceed the value of a 32-bit signed integer, 21471183247. For example:

  ```
  int StartVolumes = 1
  ```

- Boolean

  A boolean is an integer, the possible values of which are 0 (false) and 1 (true). For example, the `Critical` attribute has two possible values:

  ```
  bool Critical = 1
  ```

Attribute Dimensions

- Scalar

  A scalar has only one value. This is the default dimension. The definition of an attribute resembles:

  ```
  str scalar_attribute
  ```

  For example:

  ```
  str MountPoint
  ```

  When values are assigned to a scalar, the attribute in the `main.cf` file might resemble:

  ```
  MountPoint = "/Backup"
  ```

- Vector

  A vector is an ordered list of values. Each value is indexed using a positive integer beginning with zero. A set of brackets ([]) denotes that the dimension is a vector. Brackets are specified after the attribute name in the attribute definition. To define an attribute with a vector dimension, add a line in the resource type definition that resembles:

  ```
  str vector_attribute[]
  ```

  For example:

  ```
  str BackupSys[]
  ```

  When values are assigned to a vector, the attribute in the `main.cf` file might resemble:

  ```
  BackupSys[] = { sysA, sysB, sysC }
  ```

◆ Keylist

A keylist is an unordered list of strings, with each string being unique within the list. To define an attribute with a keylist dimension, add a line in the resource type definition that resembles:

```
keylist keylist_attribute = { value1, value2 }
```

For example:

```
keylist BackupVols = {}
```

When values are assigned to a keylist, the attribute in the `main.cf` file might resemble:

```
BackupVols = { vol1, vol2 }
```

◆ Association

An association is an unordered list of name-value pairs. Each pair is separated by an equal sign. A set of braces ({}) denotes that an attribute is an association. Braces are specified after the attribute name in the attribute definition. To define an attribute with an association dimension, add a line in the resource type definition that resembles:

```
int assoc_attr{} = { attr1 = val1, attr2 = val2 }
```

For example:

```
int BackupSysList {}
```

When values are assigned to an association, the attribute in the `main.cf` file might resemble:

```
BackupSysList{} = { sysa=1, sysb=2, sysc=3 }
```

Agent Entry Points 2

Developing a VCS agent requires using the agent framework and implementing *entry points*. An entry point is a *plug-in*, defined by the user, that is called when an event, such as onlining, offlining, or monitoring a resource, occurs within the VCS agent. Definitions of each of the supported entry points begin with "Agent Entry Points" on page 19.

The VCS agent framework ensures that a resource has only one entry point running at a time. If multiple requests or events are received for the same resource, they are queued, then processed one at a time. An exception to this behavior is an optimization such that the agent framework discards internally generated *periodic monitoring* requests for a resource that is being monitored or that has a pending monitor request. However, because the agent framework is multithreaded, a single agent process can run entry points of several resources simultaneously. For example, if a resource receives requests to offline first, then close, the `offline` entry point is called first. The `close` entry point is called only after the offline request returns or times out. However, if the offline request is received for one resource, and the close request is received for another, both are called simultaneously.

Using C++ or Script Entry Points

An entry point can be implemented as a C++ function or a script. The advantage to using C++ is that entry points are compiled and linked with the agent framework library. They run as part of the agent process, so there is no system overhead when they are called. The advantage to using scripts is that you can modify the entry points dynamically; however, a new process is created each time the entry points are called.

Note that you may use C++, Perl, and shell in any combination to implement multiple entry points for a single agent. This allows you to implement each entry point in the most advantageous manner. For example, you may use scripts to implement most entry points while using C++ to implement the `monitor` entry point, which is called often. If the `monitor` entry point is written in script, a new process must be created each time it is called.

C++ Agents

If you create an agent with all the agent's entry points in C++, or some entry points in C++ and some in script, you must create a C++ agent that contains the `VCSAgStartup` routine, the necessary C++ primitives, and the C++ entry points. A sample file containing templates for creating an agent using C++ entry points is located in $VCS_HOME/src/agent/Sample. Refer to "Building a Custom VCS Agent" on page 87 for information about how to build an agent using C++ entry points or a combination of C++ and script entry points. See also "Implementing Entry Points Using C++" on page 31 or "Implementing Entry Points Using Scripts" on page 61.

Script Agents

If you create an agent using only script entry points, that is, no C++ code, you can base the agent on the ScriptAgent, $VCS_HOME/bin/ScriptAgent. Refer to "Building a Custom VCS Agent" on page 87 for information about creating an agent using only script entry points. See also, "Implementing Entry Points Using Scripts" on page 61.

VCSAgStartup

When an agent starts, it uses the routine named `VCSAgStartup` to initialize the agent's data structures and connect the agent to the VCS engine. After the agent downloads the necessary information it needs for the configured resources from the engine, it can control the resources based on the entry points.

Implementing All or Some of the Entry Points in C++

If you implement all or some of the entry points in C++, you can use `VCSAgStartup` routine within the agent to assign the entry points to be used. You can do this by defining a variable of type `VCSAGV40EntryPointStruct` and setting its fields appropriately for each entry point. (See "VCS Primitives" on page 51.) If you implement some entry points using scripts, assign a `NULL` value to their fields in `VCSAgV40EntryPointStruct`, in which case, the agent looks for and executes the scripts.

Implementing Entry Points Using Scripts

If you implement all of the agent's entry points using scripts, you can base the agent on the ScriptAgent, which includes a built-in implementation of `VCSAgStartup` that looks for and executes the scripts.

Sample Structure

`VCSAgStartup` registers the agent entry points with the agent framework by calling the primitive `VCSAgRegisterEPStruct`, which includes the structure `VCSAgV40EntryPointStruct`.

`VCSAgV40EntryPointStruct` has the following definition:

```
// Structure used to register the entry points.

typedef struct {
   void (*open)(const char *res_name,void **attr_val);
   void (*close)(const char *res_name,void **attr_val);
   VCSAgResState (*monitor)(const char
      *res_name, void **attr_val,
      int *conf_level);
   unsigned int (*online)(const char *res_name,
      void **attr_val);
   unsigned int (*offline) (const char *res_name,
      void **attr_val);
   unsigned int (*action) (const char *res_name, const char
      *action_token, void **attr_val, char **action_args,
      char *action_output);
   unsigned int (*info) (const char *res_name,
        VCSAgResInfoOp resinfo_op, void **attr_val, char
        **info_output, char ***opt_update_args, char
        ***opt_add_args);
   void (*attr_changed) (const char *res_name,
      const char *changed_res_name, const char
      *changed_attr_name, void **new_val);
   unsigned int (*clean) (const char *res_name,
      VCSAgWhyClean reason, void **attr_val);
   void (*shutdown) ();
} VCSAgV40EntryPointStruct;
```

Example: Modifying VCSAgStartup for C++ and Script Entry Points

When using C++ to implement an entry point, assign the entry point's function to the corresponding field of `VCSAgV40EntryPointStruct`. In the following example, the function `my_shutdown` is assigned to the field `shutdown`.

Note that the `monitor` entry point, which is mandatory, is assigned a `NULL` value, indicating it is implemented using scripts. If you are using a script entry point, or if you are not implementing an optional entry point, set the corresponding field to `NULL`. For an entry point whose field is set to `NULL`, the agent automatically looks for the correct script to execute: `$VCS_HOME/bin/resource_type/`*`entry_point`*.

```
#include "VCSAgApi.h"
void my_shutdown() {
    ...
}

void VCSAgStartup() {
   VCSAgV40EntryPointStruct ep;
   ep.open = NULL;
   ep.online = NULL;
   ep.offline = NULL;
   ep.monitor = NULL;
   ep.attr_changed = NULL;
   ep.clean = NULL;
   ep.close = NULL;
   ep.info = NULL;
   ep.action = NULL;
   ep.shutdown = my_shutdown;
   }

   VCSAgRegisterEPStruct(V40, &ep);
```

Agent Entry Points

The VCS agent framework supports the entry points listed below. With the exception of `monitor`, all entry points are optional. Each may be implemented in C++ or scripts.

- monitor
- info
- online
- offline
- clean
- action
- attr_changed
- open
- close
- shutdown

Beginning with the entry point `monitor` below, each VCS agent entry point is listed and defined in the following sections.

monitor

The `monitor` entry point typically contains the code to determine status of the resource. For example, the `monitor` entry point of the IP agent checks whether or not an IP address is configured, and returns the state online, offline, or unknown.

Note This entry point is mandatory.

The framework calls the `monitor` entry point after completing the `online` and `offline` entry points to determine if bringing the resource online or taking it offline was effective. The agent framework also calls this entry point periodically to detect if the resource was brought online or taken offline unexpectedly. Under normal circumstances, the monitor runs every sixty seconds when a resource is online, and every 300 seconds when a resource of expected to be offline.

The `monitor` entry point receives a resource name and `ArgList` attribute values as input (see “ArgList” on page 99).

It returns the resource status (online, offline, or unknown), and the confidence level 0–100. The confidence level is informative only and is not used by VCS. It is returned only when the resource status is online.

A C++ entry point can return a confidence level of 0–100. A script entry point combines the status and the confidence level in a single number. For example:

- 100 indicates offline.
- 101 indicates online and confidence level 10.
- 102 indicates online and confidence level 20.
- 103–109 indicates online and confidence levels 30–90.
- 110 indicates online and confidence level 100.

If the exit value of the monitor script entry point falls outside the range 100–110, the status is considered unknown.

info

The info entry point enables agents to obtain information about an online resource. For example, the Mount agent's info entry point could be used to report on space available in the file system.

All information collected by the info entry point is stored in the temp attribute ResourceInfo. The ResourceInfo attribute is a string association that stores name-value pairs. By default, there are three such name-value pairs: State, Msg, and TS. State indicates the status of the information contained in the ResourceInfo attribute; Msg indicates the output of the info entry point, in any; TS indicates the timestamp of when the ResourceInfo attribute was last modified.

The entry point can optionally modify a resource's ResourceInfo attribute by adding or updating other name-value pairs using the following commands:

```
hares -modify res ResourceInfo -add attribute value
```

or

```
hares -modify res ResourceInfo -update attribute value
```

For a description of the ResourceInfo attribute, see "ResourceInfo" on page 107. Refer also to the manual page for the hares command.

For input, the info entry point receives as arguments the resource name, the value of resinfo_op, and the ArgList attribute values. In the case of C++ implementation, the output of the entry point is returned in info_output. Any optional name-value pairs are returned in either opt_add_args or opt_update_args two-dimensional character arrays. See the C++ example, "Example, Info Entry Point Implementation in C++" on page 39. For the script example, see "info" on page 65.

Return Values

- If the info entry point exits with 0 (success), the output captured on stdout for the script entry point, or the contents of the info_output argument for C++ entry point, is dumped to the Msg key of the ResourceInfo attribute. The Msg key is updated only when the info entry point is successful. The State key is set to the value: Valid.
- If the entry point exits with a non-zero value, ResourceInfo is updated to indicate the error; output of the script's stdout or the C++ entry point's info_output is ignored. The State key is set to the value: Invalid. The error message is written to the agent's log file.
- If the info entry point times out, output from the entry point is ignored. The State key is set to the value: Invalid. The error message is written to the agent's log file.
- If the info entry point is killed by a user (for example, kill -15 *pid*), the State key is set to the value: Invalid. The error message is written to the agent's log file.

- If the resource for which the entry point is invoked goes offline or faults, the State key is set to the value: `Stale`.
- If the `info` entry point is not implemented, the State key is set to the value: `Stale`. The error message is written to the agent's log file.

The `info` entry point can be invoked from the command line for a given online resource using the `hares -refreshinfo` command. See the `hares` manual page.

online

The `online` entry point typically contains the code to bring a resource online. For example, the `online` entry point for an IP agent configures an IP address. When the online procedure completes, the `monitor` entry point is automatically called by the framework to verify that the resource is online.

The `online` entry point receives a resource name and `ArgList` attribute values as input. It returns an integer indicating the number of seconds to wait for the online to take effect. The typical return value is 0. If the return value is not zero, the agent framework schedules the monitor to begin only after this time has elapsed.

offline

The `offline` entry point is called to take a resource offline. For example, the `offline` entry point for an IP agent removes an IP address from the system. When the offline procedure completes, the `monitor` entry point is automatically called by the framework to verify that the resource is offline.

The `offline` entry point receives a resource name and `ArgList` attribute values as input. It returns an integer indicating the number of seconds to wait for the offline to take effect. The typical return value is 0. If the return value is not zero, the agent framework schedules the monitor to begin only after this time has elapsed.

clean

The `clean` entry point is called automatically by the agent framework when all ongoing tasks associated with a resource must be terminated and the resource must be taken offline, perhaps forcibly. The entry point receives as input the resource name, an encoded reason describing why the entry point is being called, and the `ArgList` attribute values. It must return 0 if the operation is successful, and 1 if unsuccessful.

The reason for calling the entry point is encoded according to the following `enum` type:

```
enum VCSAgWhyClean {
     VCSAgCleanOfflineHung,
     VCSAgCleanOfflineIneffective,
     VCSAgCleanOnlineHung,
     VCSAgCleanOnlineIneffective,
     VCSAgCleanUnexpectedOffline,
     VCSAgCleanMonitorHung
};
```

- VCSAgCleanOfflineHung

The `offline` entry point did not complete within the expected time.

(See “OfflineTimeout” on page 105.)

- VCSAgCleanOfflineIneffective

The `offline` entry point was ineffective.

- VCSAgCleanOnlineHung

The `online` entry point did not complete within the expected time.

(See “OnlineTimeout” on page 105.)

- VCSAgCleanOnlineIneffective

The `online` entry point was ineffective.

- VCSAgCleanUnexpectedOffline

The online resource faulted because it was taken offline unexpectedly.

- VCSAgCleanMonitorHung

The online resource faulted because the `monitor` entry point consistently failed to complete within the expected time.

(See “FaultOnMonitorTimeouts” on page 101.)

The agent supports the following tasks when the `clean` entry point is implemented:

- ✔ Automatically restarts a resource on the local system when the resource faults. (See the RestartLimit attribute for the resource type.)
- ✔ Automatically retries the `online` entry point when the attempt to bring a resource online fails. (See the OnlineRetryLimit attribute for the resource type.)
- ✔ Enables the VCS engine to bring a resource online on another system when the `online` entry point for the resource fails on the local system.

For the above actions to occur, the `clean` entry point must return 0.

action

The command hares with the -action option invokes the action entry point. Administrators can issue the command to bring about a specific action with respect to a specified resource on a given system within a given cluster. Actions are designated by a the *action_token* argument. Typically, such actions are those that can be completed in a short time and do not involve onlining or offlining the resource. The following shows the syntax for the -action option used with the hares command:

```
hares -action res_name action_token [-actionargs arg1 arg2 ... ]
  [-sys sys_name] [-clus cluster]
```

The actions specified by the action token correspond to actions defined in the static attribute SupportedActions in the resource type definition file (see "SupportedActions" on page 109). Such actions may include getting the name of a database instance (in the case of a database-related agent, for example), putting a database in the restricted mode, taking a database out of restricted mode, backing up a database, and so on.

For a script-based implementation of the action entry point, a directory named actions within /opt/VRTSvcs/bin/*agent* must contain scripts named for each action indicated by the action token. For example, the RVG agent could have the scripts named: demote, split_dg, and promote in directory /opt/VRTSvcs/bin/RVG/actions. The agent framework invokes the script directly. (See "action" on page 64.)

For C++-based implementation of the action entry point, case statements correspond to actions, one for each action_token. See "action" on page 44.

Return Values

The action entry point exits with a 0 if it is successful, or 1 if not successful. The command hares -action exits with 0 if the action entry point exits with a 0 and 1 if the action entry point is not successful.

ActionTimeout

The default value of the static resource type attribute, ActionTimeout, is 20 seconds. This attribute can be overridden for specific resources. See "ActionTimeout" on page 98.

attr_changed

The `attr_changed` entry point is called when a resource attribute is modified, and only if that resource is registered with the agent framework for notification. See the primitives "VCSAgRegister" on page 54 and "VCSAgUnregister" on page 55 for details. To register automatically, see "RegList" on page 108. This entry point receives as input the resource name registered with the agent framework for notification, the name of the changed resource, the name of the changed attribute, and the new attribute value. It does not return a value. This entry point provides a way to respond to resource changes. Most agents do not require this functionality and will not implement this entry point.

open

The `open` entry point is called when the VCS agent starts managing a resource; for example, when the agent starts, or when the value of the `Enabled` attribute is changed from 0 to 1. It receives a resource name and `ArgList` attribute values as input and returns no value. This entry point typically initializes the resource.

Note A resource can be brought online, taken offline, and monitored only if it is managed by a VCS agent. The value of the resource's `Enabled` attribute must be set to 1.

When a VCS agent is started, the `open` entry point of each resource is guaranteed to be called before its `online`, `offline`, or `monitor` entry points are called. This allows you to include initializations for specific resources. Most agents do not require this functionality and will not implement this entry point.

close

The `close` entry point is called when the VCS agent stops managing a resource. For example, it is called when the value of the `Enabled` attribute is changed from 1 to 0. It receives a resource name and `ArgList` attribute values as input and returns no value. This entry point typically deinitializes the resource if implemented. Most agents do not require this functionality and will not implement this entry point.

Note A resource is monitored only if it is managed by a VCS agent. The value of the resource's `Enabled` attribute must be set to 1.

shutdown

The `shutdown` entry point is called before the VCS agent shuts down. It receives no input and returns no value. Most agents do not require this functionality and will not implement this entry point.

Summary of Return Values for Entry Points

The following table summarizes the return values for each entry point.

Entry Point	Return Values
Monitor	C++ Based Returns ResStateValues: ◆ VCSAgResOnline ◆ VCSAgResOffline ◆ VCSAgResUnknown Script-Based Exit values: ◆ 100 - Offline ◆ 101-110 - Online ◆ 99 - Unknown
Info	0 if successful; non-zero value if not successful
Online	Integer specifying number of seconds to wait before monitor can check the state of the resource; typically 0, i.e., check resource immediately.
Offline	Integer specifying number of seconds to wait before monitor can check the state of the resource; typically 0, i.e., check resource immediately.
Clean	0 if successful; non-zero value if not successful If clean fails, the resource remains in a transition state awaiting the next periodic monitor. After the periodic monitor, clean is attempted again. The sequence of clean attempt followed by monitoring continues until clean succeeds. Refer to "State Transition Diagrams" on page 121 for descriptions of internal transition states.
Action	0 if successful; non-zero value if not successful
Attr_changed	None
Open	None
Close	None
Shutdown	None

Agent Information File

The Java-based graphical user interface (GUI), Cluster Manager, can display information about the attributes of a given resource type. For each custom agent, developers can create an XML file that contains the attribute information for use by the GUI. The XML file also contains information to be used by the GUI to allow or disallow certain operations.

Example Agent Information File

The agent's information file is an XML file, named `agent_name.xml`, located in the agent directory. The file contains information about the agent, such as its name and version, and the description of the arguments for the resource type attributes. For example, the following file contains information for the FileOnOff agent:

```
<?xml version="1.0">
<agent name="FileOnOff" version="4.0">
    <agent_description>Creates, removes, and monitors files.
    </agent_description>
    <!--Platform the agent runs on-->
    <platform>Solaris</platform>
    <!--Type of agent : script-Binary-Mixed-->
    <agenttype>Binary</agenttype>
    <!--info entry point implemented or not-->
    <info_implemented>No</info_implemented>
    <!--The minimum VCS version needed for this agent-->
    <minvcsversion>4.0</minvcsversion>
    <!--The agent vendor name-->
    <vendor>VERITAS</vendor>
    <!--Attributes list for this agent-->
    <attributes>
        <PathName type="str" dimension="Scalar" editable="True"
        important="True" mustconfigure="True" unique="True"
        persistent="True" range="" default=""
        displayname="PathName">
        <attr_description>Specifies the absolute pathname.
        </attr_description>
        </PathName>
    </attributes>
    <!--List of files installed by this agent-->
    <agentfiles>
        <file name="$VCS_HOME/bin/FileOnOff/FileOnOffAgent" />
    </agentfiles>
</agent>
```

Agent Information

The information describing the agent is contained in the first section of the XML file. The following table describes this information, which is also contained in the previous file example:

Agent Information	Example
Agent name	`name="FileOnOff"`
Version	`version="4.0"`
Agent description	`<agent_description>Creates, removes, and monitors files.</agent_description>`
Platform. For example, Windows 2000 i386, or Solaris Sparc, or HP-UX 11.11.	`<platform>HP-UX</platform>`
Agent vendor	`<vendor>VERITAS<\vendor>`
`info` entry point implemented or not; Yes, or No; if not indicated, `info` entry point is assumed not implemented	`<info_implemented>No</info_implemented>`
Agent type, for example, Binary, Script or Mixed	`<agenttype>Binary</agenttype>`
VCS compatibility; the minimum version required to support the agent	`<minvcsversion>4.0</minvcsversion>`

Attribute Argument Details

The agent's attribute information is described by several arguments which are listed in the following table. Refer also to the previous XML file example for the FileOnOff agent and see how the `PathName` attribute information is included in the file.

Argument	Description
type	Possible values for attribute type, such as "str" for strings; see "Attribute Data Types" on page 12

Argument	Description
dimension	Values for the attribute dimension, such as "Scalar;" see "Attribute Dimensions" on page 13 for more information on dimensions
editable	Possible Values = "True" or "False" Indicates if the attribute is editable or not. In most cases, the resource attributes are editable.
important	Possible Values = "True" or "False" Indicates whether or not the attribute is important enough to display. In most cases, the value is True.
mustconfigure	Possible Values = "True" or "False" Indicates if the attribute must be configured to make the resource online. GUI displays such attributes with special indication. If the value of the `mustconfigure` attribute is not specified, the resource state should become "UNKNOWN" in the first monitor cycle. Example of such attributes are `Address` for the IP agent, `Device` for the NIC agent, and `FsckOpt` for the Mount agent).
unique	Possible Values = "True" or "False" Indicates if the attribute value must be unique in the VCS configuration, that is, whether or not two resources of same resource type may have the same value for this attribute. Example of such an attribute is `Address` for the IP agent. Not used in the GUI.
persistent	Possible Values = "True". This argument should always be set to "True"; it is reserved for future use.
range	Defines the acceptable range of the attribute value. GUI or any other client can use this value for attribute value validation. Value Format: The range is specified in the form {a,b} or [a,b]. Square brackets indicate that the adjacent value is included in the range. The curly brackets indicate that the adjacent value is not included in the range. For example, {a,b] indicates that the range is from a to b, contains b, and excludes a. In cases where the range is greater than "a" and does not have an upper limit, it can be represented as {a,] and, similarly, as {,b] when there is no minimum value.
default	It indicates the default value of attribute
displayname	It is used by GUI or clients to show the attribute in user friendly manner. For example, for FsckOpt its value could be "fsck option".

Implementing Entry Points Using C++ 3

This chapter describes how to use C++ to implement agent entry points. This chapter also describes agent primitives, the C++ functions provided by the VCS agent framework.

Because the agent framework is multithreaded, all C++ code written by the agent developer must be MT-safe. For best results, avoid using global variables. If you do use them, access must be serialized (for example, by using mutex locks).

The following guidelines also apply:

- Do not use C library functions that are unsafe in multithreaded applications. Instead, use the equivalent reentrant versions, such as `readdir_r()` instead of `readdir()`. Access manual pages for either of these commands by entering: `man` *`command`*.
- When acquiring resources (dynamically allocating memory or opening a file, for example), use thread-cancellation handlers to ensure that resources are freed properly. See the manual pages for `pthread_cleanup_push` and `pthread_cleanup_pop` for details. Access manual pages for either of these commands by entering: `man` *`command`*.

Data Structures

```
// Values for the state of a resource - returned by the
// monitor entry point.
enum VCSAgResState {
 VCSAgResOffline,// Resource is offline.
 VCSAgResOnline,// Resource is online.
 VCSAgResUnknown// Resource is neither online nor offline.
};
```

```
// Values for the reason why the clean entry point
// is called.

enum VCSAgWhyClean {
  VCSAgCleanOfflineHung,  // offline entry point did
                    // not complete within the
                        // expected time.
  VCSAgCleanOfflineIneffective,     // offline entry point
                             // was ineffective.
  VCSAgCleanOnlineHung,          // online entry point did
                        // not complete within the
                       // expected time.
  VCSAgCleanOnlineIneffective,   // online entry point
                                   // was ineffective.
  VCSAgCleanUnexpectedOffline,   // the resource became
                           // offline unexpectedly.
  VCSAgCleanMonitorHung   // monitor entry point did
                     // not complete within the
                      // expected time.
};
// Structure used to register the entry points.

typedef struct {
  void (*open)(const char *res_name, void **attr_val);
  void (*close)(const char *res_name, void **attr_val);
  VCSAgResState (*monitor)(const char *res_name,
         void **attr_val, int, *conf_level);
  unsigned int (*online)(const char *res_name,
                 void **attr_val);
  unsigned int (*offline)(const char *res_name,
                 void **attr_val);
  unsigned int (*action) (const char *res_name, const char
       *action_token, void **attr_val, char **action_args,
       char *action_output);
  unsigned int (*info) (const char *res_name, VCSAgResInfoOp
         resinfo_op, void **attr_val, char **info_output, char
         ***opt_update_args, char ***opt_add_args);
  void (*attr_changed)(const char *res_name,
     const char *changed_res_name, const char
     *changed_attr_name, void **new_val);
  unsigned int (*clean)(const char *res_name,
     VCSAgWhyClean reason, void **attr_val);
  void (*shutdown) ();
} VCSAgV40EntryPointStruct;
```

The structure `VCSAgV40EntryPointStruct` consists of function pointers, one for each VCS entry point except `VCSAgStartup`. The `VCSAgStartup` entry point is called by name, and therefore must be implemented using C++ and named `VCSAgStartup`.

ArgList Attribute

The `ArgList` attribute is a predefined static attribute that specifies the list of attributes whose values are passed to the `open`, `close`, `online`, `offline`, `action`, `info`, and `monitor` entry points. The values of the `ArgList` attributes are passed through a parameter of type `void **`. For example, the signature of the `online` entry point is:

```
unsigned int
 res_online(const char *res_name, void **attr_val);
```

The parameter attr_val is an array of character pointers that contains the `ArgList` attribute values. The last element of the array is a NULL pointer. Attribute values in attr_val are listed in the same order as attributes in `ArgList`.

The values of scalar attributes (integer and string) are each contained in a single element of attr_val. The values of non-scalar attributes (vector, keylist, and association) are contained in one or more elements of attr_val. If a non-scalar attribute contains *N* components, it will have *N*+1 elements in `attr_val`. The first element is *N*, and the remaining *N* elements correspond to the *N* components. See“ArgList” on page 99 for more information. See the chapter describing the VCS configuration language in the *VERITAS Cluster Server User's Guide* for attribute definitions.

For example, if Type "Foo" is defined in the file `types.cf` as:

```
Type Foo (
   str Name
   int IntAttr
   str StringAttr
   str VectorAttr[]
   str AssocAttr{}
   static str ArgList[] = { IntAttr, StringAttr,
      VectorAttr, AssocAttr }
)
```

And if a resource "Bar" is defined in the file `main.cf` as:

```
Foo Bar (
   IntAttr = 100
   StringAttr = "Oracle"
   VectorAttr = { "vol1", "vol2", "vol3" }
   AssocAttr = { "disk1" = "1024", "disk2" = "512" }
)
```

The parameter `attr_val` will be:

```
attr_val[0] ===>  "100"     // Value of IntAttr, the first
                 // ArgList attribute.
attr_val[1] ===>  "Oracle" // Value of StringAttr.
attr_val[2] ===>  "3"    // Number of components in VectorAttr.
attr_val[3] ===>  "vol1"
attr_val[4] ===>  "vol2"
attr_val[5] ===>  "vol3"
attr_val[6] ===>  "4"    // Number of components in AssocAttr.
attr_val[7] ===>  "disk1"
attr_val[8] ===>  "1024"
attr_val[9] ===>  "disk2"
attr_val[10]===>  "512"
attr_val[11]===>  NULL  // Last element.
```

C++ Entry Point Syntax

The following paragraphs describes the syntax for C++ entry points.

VCSAgStartup

```
void VCSAgStartup();
```

The entry point `VCSAgStartup()` must use the primitive `VCSAgRegisterEPStruct()` to register the other entry points with the VCS agent framework. (See ““VCS Primitives” on page 51.) Note that the name of the C++ function must be `VCSAgStartup()`.

For example:

```
// This example shows the VCSAgStartup() entry point
// implementation,assuming that the monitor, online, and
// offline entry points are implemented in C++ and the respective
// function names are res_monitor, res_online, and res_offline.

#include "VCSAgApi.h"
void VCSAgStartup() {
   VCSAgV40EntryPointStruct ep;
     ep.open = NULL;
     ep.close = NULL;
     ep.monitor = res_monitor;
     ep.online = res_online;
     ep.offline = res_offline;
     ep.action = NULL;
     ep.info = NULL;
     ep.attr_changed = NULL;
     ep.clean = NULL;
     ep.shutdown = NULL;
     VCSAgRegisterEPStruct(V40, &ep);
}
VCSAgResState res_monitor(const char *res_name, void
               **attr_val, int *conf_level) {
...
}
unsigned int res_online(const char *res_name,
               void **attr_val) {
...
}
unsigned int res_offline(const char *res_name,
                void **attr_val) {
...
}
```

monitor

```
VCSAgResState
res_monitor(const char *res_name, void **attr_val,int
*conf_level);
```

You may select any name for the function.

The parameter `conf_level` is an output parameter. The return value, which indicates the resource status, must be a defined `VCSAgResState` value. See "Summary of Return Values for Entry Points" on page 27.

The `monitor` field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

VCSAgResState
 res_monitor(const char *res_name, void **attr_val, int
*conf_level)
{

  // Code to determine the state of a resource.
  VCSAgResState res_state = ...
  if (res_state == VCSAgResOnline) {
    // Determine the confidence level (0 to 100).
    *conf_level = ...
  }
  else {
      *conf_level = 0;
  }
  return res_state;
}
void VCSAgStartup() {
  VCSAgV40EntryPointStruct ep;
  ...
  ep.monitor =  res_monitor;
  ...
  VCSAgRegisterEPStruct(V40, &ep);
}
```

info

```
unsigned int (*info) (const char *res_name,
        VCSAgResInfoOp resinfo_op, void **attr_val, char
        **info_output, char ***opt_update_args, char
        ***opt_add_args);
```

You may select any name for the function.

resinfo_op

The `resinfo_op` parameter indicates whether to initialize or update the data in the `ResourceInfo` attribute. The values of this field and their significance are described in the following table:

Value of `resinfo_op`	Significance
1	Add and initialize static and dynamic name-value data pairs in the `ResourceInfo` attribute.
2	Update just the dynamic data in the `ResourceInfo` attribute.

info_output

The parameter `info_output` is a character string that stores the output of the `info` entry point. The output value could be any summarized data for the resource. The `Msg` key in the `ResourceInfo` attribute is updated with `info_output`. If the `info` entry point exits with success (0), the output stored in `info_output` is dumped into the `Msg` key of the `ResourceInfo` attribute.

The `info` entry point is responsible for allocating memory for `info_output`. The agent framework handles the deletion of any memory allocated to this argument. Since memory is allocated in the entry point and deleted in the agent framework, the entry point needs to pass the address of the allocated memory to the agent framework.

opt_update_args

The `opt_update_args` parameter is an array of character strings that represents the various name-value pairs in the `ResourceInfo` attribute. This argument is allocated memory in the `info` entry point, but the memory allocated for it will be freed in the agent framework. The `ResourceInfo` attribute is updated with these name-value pairs. The names in this array must already be present in the `ResourceInfo` attribute.

For example:

```
ResourceInfo = { State = Valid, Msg = "Info entry point output",
        TS = "Wed May 28 10:34:11 2003", FileOwner = root,
        FileGroup = root, FileSize = 100 }
```

A valid `opt_update_args` array for this `ResourceInfo` attribute would be:

```
opt_update_args = { "FileSize", "102" }
```

This array of name-value pairs updates the dynamic data stored in the `ResourceInfo` attribute.

An invalid `opt_update_args` array would be one that specifies a key not already present in the `ResourceInfo` attribute or one that specifies any of the keys: `State`, `Msg`, or `TS`. These three keys can only be updated by the agent framework and not by the entry point.

opt_add_args

`opt_add_args` is an array of character strings that represent the various name-value pairs to be added to the `ResourceInfo` attribute. The names in this array represent keys that are *not* already present in the `ResourceInfo` association list and have to be added to the attribute. This argument is allocated memory in the `info` entry point, but this memory is freed in the agent framework. The `ResourceInfo` attribute is populated with these name-value pairs.

For example:

```
ResourceInfo = { State = Valid, Msg = "Info entry point output",
        TS = "Wed May 28 10:34:11 2003" }
```

A valid `opt_add_args` array for this would be:

```
        opt_add_args = { "FileOwner", "root", "FileGroup", "root",
                        "FileSize", "100" }
```

This array of name-value pairs adds to and initializes the static and dynamic data stored in the `ResourceInfo` attribute.

An invalid `opt_add_args` array would be one that specifies a key that is already present in the `ResourceInfo` attribute, or one that specifies any of the keys `State`, `Msg`, or `TS`; these are keys that can be updated only by the agent framework, not by the entry point.

Example, Info Entry Point Implementation in C++

The `info` field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

For example:

```
extern "C" unsigned int
file_info(const char *res_name, VCSAgResInfoOp resinfo_op,
        void **attr_val, char **output, char ***opt_update_args,
        char ***opt_add_args) {

        struct stat stat_buf;
        int i;
        char **args = NULL;
        char *out = new char [80];

        *output = out;

        VCSAgSnprintf(out, 80,
"Output of info entry point...updates the ”Msg” key in ResourceInfo
attribute");

// Use the stat system call on the file to get its information

        if ((attr_val) && (*attr_val)) {
           if ((stat((CHAR *)(*attr_val), &stat_buf) == 0) &&
                        (strlen((CHAR *)(*attr_val)) != 0)) {

           if (resinfo_op == VCSAgResInfoAdd) {
// Add and initialize all the static and
// dynamic keys in the ResourceInfo attribute

           args = new char * [7];
           for (i = 0; i < 6; i++) {
                       args[i] = new char [15];
           }
// All the static information - file owner and group
           VCSAgSnprintf(args[0], 15, "%s", "Owner");
           VCSAgSnprintf(args[1], 15, "%d", stat_buf.st_uid);
           VCSAgSnprintf(args[2], 15, "%s", "Group");
           VCSAgSnprintf(args[3], 15, "%d", stat_buf.st_gid);
```

```
// Initialize the dynamic information for the file
           VCSAgSnprintf(args[4], 15, "%s", "FileSize");
           VCSAgSnprintf(args[5], 15, "%d", stat_buf.st_size);
           args[6] = NULL;
               *opt_add_args = args;
          }
          else {
// Simply update the dynamic keys in the ResourceInfo
// attribute. In this case, the dynamic info on the file

          args = new char * [3];
          for (i = 0; i < 2; i++) {
                 args[i] = new char [15];
          }
          VCSAgSnprintf(args[0], 15, "%s", "FileSize");
          VCSAgSnprintf(args[1], 15, "%d", stat_buf.st_size);
          args[2] = NULL;
          *opt_update_args = args;
          }
    }
    else {
// Set the output to indicate the error
           VCSAgSnprintf(out, 80, "Stat on the file %s failed",
           *attr_val);
           return 1;
    }
    else {
// Set the output to indicate the error
           VCSAgSnprintf(out, 80,
           "Error in arglist values passed to the info entry
           point");
           return 1;
    }

// Successful completion of the info entry point
    return 0;
}
```

online

```
unsigned int
res_online(const char *res_name, void **attr_val);
```

You may select any name for the function.

The `online` field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

unsigned int
 res_online(const char *res_name, void **attr_val) {
  // Implement the code to online a resource here.
  ...
  // If monitor can check the state of the resource
  // immediately, return 0. Otherwise, return the
  // appropriate number of seconds to wait before
  // calling monitor.
  return 0;
}

void VCSAgStartup() {
   VCSAgV40EntryPointStruct ep;
   ...
   ep.online =  res_online;
   ...
   VCSAgRegisterEPStruct(V40, &ep);
}
```

offline

```
unsigned int
res_offline(const char *res_name, void **attr_val);
```

You may select any name for the function.

The `offline` field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

unsigned int
 res_offline(const char *res_name, void **attr_val) {
  // Implement the code to offline a resource here.
  ...
  // If monitor can check the state of the resource
  // immediately, return 0. Otherwise, return the
  // appropriate number of seconds to wait before
  // calling monitor.
  return 0;
}

void VCSAgStartup() {
   VCSAgV40EntryPointStruct ep;
   ...
   ep.offline =  res_offline;
   ...
   VCSAgRegisterEPStruct(V40, &ep);
}
```

clean

```
unsigned int
res_clean(const char *res_name, VCSAgWhyClean reason, void
**attr_val);
```

You may select any name for the function.

The clean field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

unsigned int
res_clean(const char *res_name, VCSAgWhyClean reason,
  void **attr_val) {
// Code to forcibly offline a resource.
...
// If the procedure is successful, return 0; else
// return 1.
return 0;


void VCSAgStartup() {
  VCSAgV40EntryPointStruct ep;
  ...
    ep.clean =  res_clean;
    ...
    VCSAgRegisterEPStruct(V40, &ep);
}
```

action

```
unsigned int
action(const char *res_name, const char *action_token,
   void **attr_val, char **args, char *output);
```

You may select any name for the function.

The `action` field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

For example:

```
extern "C"
unsigned int file_action (const char *res_name, const char *token,
                       void **attr_val, char **args, char *output)
{

        //
        // checks on the attr_val entry point arg list
        //

        //
        // perform an action based on the action token passed in
        //

        if (!strcmp(token, "token1")) {
                //
                // Perform action corresponding to token1
                //
        } else if (!strcmp(token, "token2") {
                //
                // Perform action corresponding to token2
                //
        }
        :
        :
        :
        } else {
                //
                // a token that doesnt have an impl yet
                //
                VCSAgSnprintf(ep_output, MAXBUFFER,
                       "No implementation provided for token(%s)",
                        token);
        }

        //
```

```
        // Any other checks to be done
        //

        //
        // return value should indicate whether the ep succeeded or
        // not:
        //  return 0 on success
        //  any other value on failure
        //
        if (success)
           return 0;
        else
           return 1;
}
```

attr_changed

```
void
res_attr_changed(const char *res_name, const char
            *changed_res_name,
            const char *changed_attr_name,
            void **new_val);
```

The parameter `new_val` contains the attribute's new value. The encoding of `new_val` is similar to the encoding of the "ArgList Attribute" on page 33.

You may select any name for the function.

The `attr_changed` field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

Note This entry point is called only if you register for change notification using the primitive "VCSAgRegister" on page 54, or the agent parameter RegList (see "RegList" on page 108).

For example:

```
#include "VCSAgApi.h"

void
 res_attr_changed(const char *res_name,
          const char *changed_res_name,
          const char *changed_attr_name,
          void **new_val) {
  // When the value of attribute Foo changes, take some action.
  if ((strcmp(res_name, changed_res_name) == 0) &&
     (strcmp(changed_attr_name, "Foo") == 0)) {
    // Extract the new value of Foo. Here, it is assumed
    // to be a string.
    const char *foo_val = (char *)new_val[0];
    // Implement the action.
     ...
  }
```

```
  // Resource Ora1 managed by this agent needs to
  // take some action when the Size attribute of
  // the resource Disk1 is changed.
  if ((strcmp(res_name, "Ora1") == 0) &&
      (strcmp(changed_attr_name, "Size") == 0) &&
      (strcmp(changed_res_name, "Disk1") == 0)) {

    // Extract the new value of Size. Here, it is
    // assumed to be an integer.
    int sizeval = atoi((char *)new_val[0]);
    // Implement the action.
      ...
  }
}

void VCSAgStartup() {
    VCSAgV40EntryPointStruct ep;
    ...
    ep.attr_changed =  res_attr_changed;
    ...
    VCSAgRegisterEPStruct(V40, &ep);
}
```

open

```
void res_open(const char *res_name, void **attr_val);
```

You may select any name for the function.

The `open` field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

void  res_open(const char *res_name, void **attr_val) {
  // Perform resource initialization, if any.
  // Register for attribute change notification, if needed.
}

void VCSAgStartup() {
  VCSAgV40EntryPointStruct ep;
  ...
  ep.open =  res_open;
  ...
  VCSAgRegisterEPStruct(V40, &ep);
}
```

close

```
void res_close(const char *res_name, void **attr_val);
```

You may select any name for the function.

The close field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

void  res_close(const char *res_name,void **attr_val) {
  // Resource-specific de-initialization, if needed.
  // Unregister for attribute change notification, if any.
}

void VCSAgStartup() {
    VCSAgV40EntryPointStruct ep;
    ...
    ep.close =  res_close;
    ...
    VCSAgRegisterEPStruct(V40, &ep);
}
```

shutdown

```
void res_shutdown();
```

You may select any name for the function.

The `shutdown` field of `VCSAgV40EntryPointStruct` passed to `VCSAgRegisterEPStruct()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

void  res_shutdown(const char *res_name) {
   // Agent-specific de-initialization, if any.
}

void VCSAgStartup() {
   VCSAgV40EntryPointStruct ep;
   ...
   ep.shutdown =  res_shutdown;
   ...
   VCSAgRegisterEPStruct(V40, &ep);
}
```

VCS Primitives

Primitives are C++ methods implemented by the VCS agent framework. Beginning with the primitive `VCSAgRegisterEPStruct()` below, each VCS primitive is listed and defined in the following sections.

VCSAgRegisterEPStruct

```
void VCSAgRegisterEPStruct (VCSAgAgentVersion version, void *
entry_points);
```

This primitive requests that the VCS agent framework use the entry point implementations designated in `entry_points`. It must be called only from the `VCSAgStartup` entry point.

For example:

```
// This example shows how to use VCSAgRegisterEPStruct()
// Primitive within the VCSAgStartup() entry point. It
// is assumed here that the monitor, online, and offline
// entry points are implemented in C++, and that the
// respective function names are res_monitor,
// res_online, and res_offline.

#include "VCSAgApi.h"

void VCSAgStartup() {
   VCSAgV40EntryPointStruct ep;

   ep.open = NULL;
   ep.close = NULL;
   ep.monitor = res_monitor;
   ep.online = res_online;
   ep.offline = res_offline;
   ep.action = NULL;
   ep.info = NULL;
   ep.attr_changed = NULL;
   ep.clean = NULL;
   ep.shutdown = NULL;
   VCSAgRegisterEPStruct(V40, &ep);
}
```

VCSAgSetCookie

```
void VCSAgSetCookie(const char *name, void *cookie);
```

This primitive requests that the VCS agent framework store a cookie. This value is transparent to the VCS agent framework, and can be obtained later by calling the primitive `VCSAgGetCookie()`. Note that a cookie is not stored permanently; it is lost when the VCS agent process exits. This primitive can be called from any entry point.

For example:

```
#include "VCSAgApi.h"
...
//
// Assume that the online, offline, and monitor
// operations on resource require a certain key. Also
// assume that obtaining this key is time consuming, but
// that it can be reused until this process is
// terminated.
//
// In this example, the open entry point obtains the key
// and stores it as a cookie. Subsequent online,
// offline, and monitor entry points get the cookie and
// use the key.
//
// Note that the cookie name can be any unique string.
// This example uses the resource name as the cookie
// name.
//

void *get_key() {
      ...
}
void  res_open(const char *res_name, void **attr_val) {
      if (VCSAgGetCookie(res_name) == NULL) {
           void *key = get_key();
           VCSAgSetCookie(res_name, key);
      }
}
```

```
VCSAgResState  res_monitor(const char *res_name, void
         **attr_val, int *conf_level_ptr) {
      VCSAgResState state = VCSAgResUnknown;
      *conf_level_ptr = 0;
      void *key = VCSAgGetCookie(res_name);
      if (key == NULL) {
           // Take care of the rare cases when
           // the open entry point failed to
           // obtain the key and set the the cookie.
           key = get_key();
           VCSAgSetCookie(res_name, key);
      }

      // Use the key for testing if the resource is
      // online, and set the state accordingly.
      ...

      return state;
}
```

VCSAgRegister

```
void
VCSAgRegister(const char *notify_res_name,
          const char *res_name,
          const char *attr_name);
```

This primitive requests that the VCS agent framework notify the resource `notify_res_name` when the value of the attribute `attr_name` of the resource `res_name` is modified. The notification is made by calling the `attr_changed` entry point for `notify_res_name`. Note that `notify_res_name` can be the same as `res_name`. This primitive can be called from any entry point, but it is useful only when the `attr_changed` entry point is implemented.

For example:

```
#include "VCSAgApi.h"
...
void  res_open(const char *res_name, void **attr_val) {

     // Register to get notified when the
     // "CriticalAttr" of this resource is modified.
     VCSAgRegister(res_name, res_name, "CriticalAttr");

     // Register to get notified when the
     // "CriticalAttr" of "CentralRes" is modified.
     VCSAgRegister(res_name, "CentralRes”, "CriticalAttr");

     // Register to get notified when the
     // "CriticalAttr" of another resource is modified.
     // It is assumed that the name of the other resource
     // is given as the first ArgList attribute.
     VCSAgRegister(res_name, (const char *)attr_val[0],
              "CriticalAttr");
}
```

VCSAgUnregister

```
void
VCSAgUnregister(const char *notify_res_name, const char *res_name,
          const char *attr_name);
```

This primitive requests that the VCS agent framework stop notifying the resource `notify_res_name` when the value of the attribute `attr_name` of the resource `res_name` is modified. This primitive can be called from any entry point.

For example:

```
#include "VCSAgApi.h"
...
void  res_close(const char *res_name, void **attr_val) {

     // Unregister for the "CriticalAttr" of this resource.
     VCSAgUnregister(res_name, res_name, "CriticalAttr");

     // Unregister for the "CriticalAttr" of "CentralRes".
     VCSAgUnregister(res_name, "CentralRes”, "CriticalAttr");

     // Unregister for the "CriticalAttr" of another resource.
     // It is assumed that the name of the other resource is
     // given as the first ArgList attribute.
     VCSAgUnregister(res_name, (const char *)
               attr_val[0], "CriticalAttr");
}
```

VCSAgGetCookie

```
void *VCSAgGetCookie(const char *name);
```

This primitive requests that the VCS agent framework get the cookie set by an earlier call to `VCSAgSetCookie()`. It returns `NULL` if cookie was not previously set. This primitive can be called from any entry point.

For example:

```
#include "VCSAgApi.h"
...
//
// Assume that the online, offline, and monitor
// operations on resource require a certain key. Also
// assume that obtaining this key is time consuming, but
// that it can be reused until this process is terminated.
//
// In this example, the open entry point obtains the key
// and stores it as a cookie. Subsequent online,
// offline, and monitor entry points get the cookie and
// use the key.
//
// Note that the cookie name can be any unique string.
// This example uses the resource name as the cookie name.
//

void *get_key() {
      ...
}
```

P

```
void  res_open(const char *res_name, void **attr_val) {
      if (VCSAgGetCookie(res_name) == NULL) {
           void *key = get_key();
           VCSAgSetCookie(res_name, key);
      }
}

VCSAgResState  res_monitor(const char *res_name, void
         **attr_val, int *conf_level_ptr) {
      VCSAgResState state = VCSAgResUnknown;
      *conf_level_ptr = 0;
      void *key = VCSAgGetCookie(res_name);
      if (key == NULL) {
           // Take care of the rare cases when the open
           // entry point failed to obtain the key and
           // set the the cookie.
           key = get_key();
           VCSAgSetCookie(res_name, key);
      }

      // Use the key for testing if the resource is
      // online, and set the state accordingly.
      ...

      return state;
}
```

VCSAgEncodeString

```
int VCSAgEncodeString(VCSAgEncodingType from_encoding,
                void *input_string, int input_string_length,
                VCSAgEncodingType to_encoding,
                void **output_string, int *output_string_length)
```

where `VCSAgEncodingType` is an enumerated data type defined as:

```
enum VCSAgEncodingType {
  VCSAgUTF8, /* UTF-8 encoding */
  VCSAgUCS2, /* UCS-2 encoding */
  VCSAgStr   /* OS encoding */
  };
```

This primitive is used to encode a string from one encoding format to another. The encoding formats are limited to UTF-8 and the default OS encoding formats for UNIX agents.

This API allocates the required amount of memory for the output encoded string. The caller is responsible for freeing this memory. The function stores a pointer to the encoded string in the variable “output_string” and sets “ouput_string_length” to the number of characters (ASCII or UCS-2) in this newly encoded string.

The return values of this primitive are 0 on success, 1 on failure.

This API is needed as part of the localization changes in the agent framework. The agent entry points written in C++ receive the arguments in UTF-8 encoding format. If the entry points need the localized values for these arguments, they must convert the arguments, using the `VCSAgEncodeString` API, into the OS_encoding or the UCS-2 encoding format.

Currently, the VCS engine does not accept localized values as input for arguments. Also, any output captured from the script entry points run in a locale other than “C” are converted into the UTF-8 encoding format by the agent framework and logged in the VCS engine log, also in the UTF-8 encoding format. To view the output in the appropriate locale, use the `hamsg` utility. Please refer to the *VCS User’s Guide* for more on the `hasmg` utility.

Example: VCSAgEncodeString

```
//
// The example shows how the primitive VCSAgEncodeString can
// be used in the agent.
//

//
// A function used in the agent to convert a given string in UTF-8
// encoding to the OS-encoding. Its a wrapper around the primitive
// VCSAgEncodeString.
//
char* ConvertToOSEncoding(const char *utf8_string)
{
        int len = 0, out_string_len;
        char *out_string = NULL;

        len = strlen(utf8_string);
        VCSAgEncodeString(VCSAgUTF8, /* convert from UTF8 */
                      (void *)utf8_string,
                      len,
                      VCSAgStr, /* to the OS encoding format */
                      (void **)&out_string,
                      &out_string_len);
        return out_string;
}

VCSAgResState res_monitor(const char *res_name, void **attr_val,
                int *conf_level_ptr)
{       ...
        char *os_encoded_string = NULL;
        ...
        //
        // Memory for os_encoded_string is allocated in the agent
        // framework primitive VCSAgEncodeString
        //
        os_encoded_string = ConvertToOSEncoding((const char
*)attr_val[0]);
        ...
        // Free the memory allocated to os_encoded_string
        if (os_encoded_string) {
                delete[] os_encoded_string;
                os_encoded_string = NULL;
        }
        ...
}
```

VCSAgStrlcpy

```
void VCSAgStrlcpy(CHAR *dst, const CHAR *src, int size)
```

This primitive copies the contents from the input buffer “src” to the output buffer “dst” up to a maximum of “size” number of characters. Here, “size” refers to the size of the output buffer “dst.” This helps prevent any buffer overflow errors. The output contained in the buffer “dst” may be truncated if the buffer is not big enough.

VCSAgStrlcat

```
void VCSAgStrlcat(CHAR *dst, const CHAR *src, int size)
```

This primitive concatenates the contents of the input buffer “src” to the contents of the output buffer “dst” up to a maximum such that the total number of characters in the buffer “dst” do not exceed the value of “size.” Here, “size” refers to the size of the output buffer “dst.”

This helps prevent any buffer overflow errors. The output contained in the buffer “dst” may be truncated if the buffer is not big enough.

VCSAgSnprintf

```
int VCSAgSnprintf(CHAR *dst, int size, const char *format, ...)
```

This primitive accepts a variable number of arguments and works just like the C library function “sprintf.” The difference is that this primitive takes in, as an argument, the size of the output buffer “dst.” The primitive stores only a maximum of “size” number of characters in the output buffer “dst.” This helps prevent any buffer overflow errors. The output contained in the buffer “dst” may be truncated if the buffer is not big enough.

Implementing Entry Points Using Scripts 4

As mentioned in previously, the `VCSAgStartup` entry point must be implemented using C++. Other entry points may be implemented using C++ or scripts. If no other entry points are implemented in C++, implementing `VCSAgStartup` is not required. Instead, developers may use ScriptAgent. See "Using Script Entry Points" on page 89 for an example.

Script entry points can be executables or scripts, such as shell or Perl (VCS includes a Perl distribution). Also, the PATH environment variable must include the directory where sh is installed.

Adhere to the following rules when implementing a script entry point:

✔ In the `VCSAgStartup` entry point, set the corresponding field of `VCSAgV40EntryPointStruct` to NULL prior to calling VCSAgRegisterEPStruct(). (If necessary, review "Agent Entry Points" on page 19.)

✔ Place the script file in the correct directory after verifying the name of the script file.

- Verify that the name of the script file is the same as the entry point.
- Place the file in the directory $VCS_HOME/bin/resource_type. For example, if the `online` entry point for Oracle is implemented using Perl, the online script must be:

    $VCS_HOME/bin/Oracle/online

The input parameters of script entry points are passed as command-line arguments. The first command-line argument for all the entry points is the name of the resource (except shutdown, which has no arguments).

Some entry points have an output parameter that is returned through the program exit value. Some entry points return their output in the arguments passed to them, for example, `info` and `action`.

ArgList Attributes

The `open`, `close`, `online`, `offline`, `monitor`, `action`, `info`, and `clean` scripts receive the resource name and values of the `ArgList` attributes. The values of scalar `ArgList` attributes (integer and string) are each contained in a single command-line argument. The values of complex `ArgList` attributes (vector and association) are contained in one or more command-line arguments.

If a vector or association attribute contains N components, it is represented by N+1 command-line arguments. The first command-line argument is N, and the remaining N arguments correspond to the N components. (For more information, see “ArgList” on page 99. See the chapter on the VCS configuration language in the *VERITAS Cluster Server User's Guide* for attribute definitions.)

```
If Type “Foo” is defined in types.cf as:
Type Foo (
    str Name
    int IntAttr
    str StringAttr
    str VectorAttr[]
    str AssocAttr{}
    static str ArgList[] = { IntAttr, StringAttr,
        VectorAttr, AssocAttr }
)
```

And if a resource “Bar” is defined in main.cf as:

```
Foo Bar (
    IntAttr = 100
    StringAttr = "Oracle"
    VectorAttr = { "vol1", "vol2", "vol3" }
    AssocAttr = { "disk1" = "1024", "disk2" = "512" }
)
```

The online script for Bar is invoked as:

```
online Bar 100 Oracle 3 vol1 vol2 vol3 4 disk1 1024 disk2 512
```

Script Entry Point Syntax

The following paragraphs describe the syntax for script entry points.

monitor

`monitor` *`resource_name  ArgList_attribute_values`*

A script entry point combines the status and the confidence level in the exit value. For example:

- 100 indicates offline.
- 101 indicates online and confidence level 10.
- 102–109 indicates online and confidence levels 20–90.
- 110 indicates online and confidence level 100.

If the exit value falls outside the range 100–110, the status is considered unknown. For example, if the exit value equals 99, the status of the resource is considered UNKNOWN.

online

`online` *`resource_name  ArgList_attribute_values`*

The exit value is interpreted as the expected time (in seconds) for the online procedure to be effective. The exit value is typically 0.

offline

`offline` *`resource_name  ArgList_attribute_values`*

The exit value is interpreted as the expected time (in seconds) for the offline procedure to be effective. The exit value is typically 0.

clean

`clean resource_name ArgList_attribute_values`

The variable *clean_reason* equals one of the following values:

0 - The `offline` entry point did not complete within the expected time. (See "OfflineTimeout" on page 105.)

1 - The `offline` entry point was ineffective.

2 - The `online` entry point did not complete within the expected time. (See "OnlineTimeout" on page 105.)

3 - The `online` entry point was ineffective.

4 - The resource was taken offline unexpectedly.

5 - The `monitor` entry point consistently failed to complete within the expected time. (See "FaultOnMonitorTimeouts" on page 101.)

The exit value is 0 (successful) or 1.

action

`action` *resource_name ArgList_attribute_values_AND_action_arguments*

The exit value is 0 (successful) or 1 (if unsuccessful).

For example:

```
#!/bin/ksh
ResName = $1
ArgListattr1 - $2
ArgListattr2 = $3
.
.
ArgListattrn = $n+1
NumActionArgs - $n+2
Actionarg1 = $n+3
Actionarg2 = $n+4
.
.
.
```

attr_changed

```
attr_changed resource_name changed_resource_name
    changed_attribute_name new_attribute_value
```

The exit value is ignored.

Note This entry point is called only if you register for change notification using the primitive `VCSAgRegister()` (see “VCSAgRegister” on page 54), or the agent parameter `RegList` (see “RegList” on page 108).

info

```
info resource_name resinfo_op ArgList_attribute_values
```

The attribute `resinfo_op` can be have the values 1 or 2.

Values of resinfo_op	Significance
1	Add and initialize static and dynamic name-value data pairs in the `ResourceInfo` attribute.
2	Update just the dynamic data in the `ResourceInfo` attribute.

This entry point can add and update static and dynamic name-value pairs to the `ResourceInfo` attribute. The info entry point has no specific output, but rather, it updates the `ResourceInfo` attribute (see “ResourceInfo” on page 107).

Example Script Implementation of the Info Entry Point

```
#!/bin/ksh

# Parse the args to the entry point
ResName=$1;
ResInfo_op=$2;
PathName=$3;
LS="/usr/bin/ls";
HARES="/opt/VRTSvcs/bin/hares";
HAGRP="/opt/VRTSvcs/bin/hagrp";

# output of the info entry point
echo "Output of info entry point...updates the 'Msg' key in
ResourceInfo attribute";
```

```
if [ -z "$PathName" ]
then
        exit 1;
else
        # Capture the info on the file
        file_info=`$LS -l "$PathName" 2>/dev/null`
        res = `echo $?`

        if [ $res -ne 0 ]
        then
                # ls failed on the pathname
                exit 1
        fi

        if [ $ResInfo_op -eq 1 ]
        then
               # Add and initialize static and dynamic data in the
              # ResourceInfo attribute

              size=`echo "$file_info" | awk ' BEGIN { FS = " " }
                { print $5 } '`
              res=0

              # Fire the "hares -modify commands to add these
              # name-value pairs to the ResourceInfo attribute

              $HARES -modify "$ResName" ResourceInfo -add
              "FileOwner" "$owner" "FileGroup" "$group" "FileSize"
             "$size"
              res=`echo $?`

                if [ $res -ne 0 ]
                then
                        exit 1
                fi

        elif [ $ResInfo_op -eq 2 ]
        then
            # Update the dynamic data only in the ResourceInfo
            # attribute.In this case, the file size is what will
            # keep changing dynamically - so update only that.

                size=`echo "$file_info" | awk ' BEGIN { FS = " " }
              { print $5 } '`
```

```
                # Fire the hares -modify -update command to update
                # the "FileSize" key in the ResourceInfo attribute
                # with the current file size. An enhancement here
                # is, check if the file size hasn't changed from the
                # value stored currently in the ResourceInfo
                # attribute. If so, dont update the
                # "FileSize" key. Else, update it.

                 resinfo_size=`$HARES -value "$ResName" ResourceInfo |
                 grep FileSize`

                # Get the FileSize value
                  prev_size=`echo "$file_size" | awk ' BEGIN
                  { FS = " " } { print $2 }'`

                  res=0
                  if [ $prev_size -ne $size ]
                  then
                # File size has changed =- so update the attribute
                # ResourceInfo so that it shows the latest file size

                  $HARES -modify "$ResName" ResourceInfo -update
                  "FileSize" "$size"
                             res=`echo $?`

                             if [ $res -ne 0 ]
                             then
                                     exit 1
                             fi
                  fi
            fi

            #The info entry point has completed successfully.
            exit 0
     fi
```

open

open *resource_name ArgList_attribute_values*

The exit value is ignored.

close

```
close resource_name  ArgList_attribute_values
```

The exit value is ignored.

shutdown

```
shutdown
```

The exit value is ignored.

Logging Agent Messages 5

This chapter describes the APIs and functions that developers can use within their agents to generate log file messages that conform to a standard message logging format.

- For information on creation and management of messages for internationalization, see "VCS Internationalized Message Catalogs" on page 137.
- For information on APIs used by VCS 3.5 and earlier, see "How to Use Pre-VCS 4.0 Agents" on page 145.

Logging in C++ and Script-based Entry Points

For VCS 4.0, developers creating C++ agent entry points can use a set of macros to log application or debug messages. Developers of script-based entry points can use a set of functions, or "wrappers," that call the VCS `halog` utility to generate application or debug messages.

VCS Agent Messages: Format

An agent log message consists of five fields. The format of the message is:

<Timestamp> <Mnemonic> <Severity> <UMI> <MessageText>

The following is an example message, of severity ERROR, generated by the VCS FileOnOff agent's `online` entry point when attempting to online the resource, a file named "MyFile":

```
Jun 26 2003 11:32:56 VCS ERROR V-16-2001-14001
FileOnOff:MyFile:online:Resource could not be brought up because,
the attempt to create the file (/tmp/MyFile) failed with error (Is
a Directory)
```

The first four fields of the message above is composed of the *timestamp*, an uppercase *mnemonic* that represents the product (VCS, in this case), the *severity*, and the *UMI* (unique message ID). The subsequent lines contain the *message text*.

Timestamp

The timestamp indicates when the message was generated. It is formatted according to the locale.

Mnemonic

The mnemonic field is used to indicate the product. The mnemonic, such as "VCS," must use all capital letters. All VCS bundled agents, enterprise agents, and custom agents use the mnemonic: "VCS."

Severity

The severity of each message is displayed in the third field of the message (Critical, Error, Warning, Notice, or Information for normal messages; 1-21 for debug messages). All C++ logging macros and script-based logging functions provide a means to define the severity of messages, both normal and debugging.

UMI

The UMI (unique message identifier) includes an originator ID, a category ID, and a message ID.

- The originator ID is a decimal number preceded by a "V-" assigned by VERITAS.
- The category ID is a number in the range of 0 to 65536 assigned by VERITAS. For each custom agent, VERITAS must be contacted so that a unique category ID can be registered for the agent.
  - For C++ messages, the category ID is defined in the `VCSAgStartup` entry point (see "Log Category" on page 76).
  - For script-based entry points, the category is set within the VCSAG_SET_ENVS function (see "VCSAG_SET_ENVS" on page 81).
  - For debug messages, the category ID, which is 50 by default, need not be defined within logging functions.
- Message IDs can range from 0 to 65536 for a category. Each normal message (that is, non-debug message) generated by an agent must be assigned a message ID. For C++ entry points, the *msgid* is set as part of the VCSAG_LOG_MSG and VCSAG_CONSOLE_LOG_MSG macros. For script-based entry points, the *msgid* is set using the VCSAG_LOG_MSG function. The *msgid* field is not used by debug functions or required in debug messages.

Message Text

The message text is a formatted message string preceded by a dynamically generated header consisting of three colon-separated fields. namely, *<name of the agent>*:*<resource>*:*<name of the entry point>*:*<message>*. For example:

```
FileOnOff:MyFile:online:Resource could not be brought up because,
the attempt to create the file (/tmp/MyFile) failed with error (Is
a Directory)
```

In the case of C++ entry points, the header information is generated

In the case of script-based entry points, the header information is set within the VCSAG_SET_ENVS function (see "VCSAG_SET_ENVS" on page 81).

C++ Agent Logging APIs

The VCS agent framework provides four logging APIs (macros) for use in agent entry points written in C++.

These APIs include two application logging macros:

```
VCSAG_CONSOLE_LOG_MSG(sev, msgid, flags, fmt, variable_args...)
VCSAG_LOG_MSG(sev, msgid, flags, fmt, variable_args...)
```

and the macros for debugging:

```
VCSAG_LOGDBG_MSG(dbgsev, flags, fmt, variable_args...)
VCSAG_RES_LOG_MSG(dbgsev, flags, fmt, variable args...)
```

The arguments of these APIs are described, briefly described in the tables that, are described in detail in the following paragraphs:

Agent Application Logging Macros for C++ Entry Points

The macro VCSAG_LOG_MSG can be used within C++ agent entry points to log all messages ranging in severity from CRITICAL to INFORMATION to the agent log file. The VCSAG_CONSOLE_LOG_MSG macro can be used to send messages to the engine log, and, in the case of messages of CRITICAL and ERROR severity, to the console.

The argument fields for the application logging macros are described in the following table:

`sev`	Severity of the message from the application. The values of `sev` are macros VCS_CRITICAL, VCS_ERROR, VCS_WARNING, VCS_NOTICE, and VCS INFORMATION; see "Severity Arguments for C++ Macros" on page 74.
`msgid`	The 16-bit integer message ID.
`flags`	Default flags (0) prints UMI, NEWLINE. A macro, VCS_DEFAULT_FLAGS, represents the default value for the flags.
`fmt`	A formatted string containing formatting specifiers symbols. For example: "Resource could not be brought down because the attempt to remove the file (%s) failed with error (%d)"
`variable_args`	Variable number (as many as 6) of type char, char *, or integer

In the following example, both macros are used to log an error message to the agent log and to the console:

```
.
.
VCSAG_LOG_MSG(VCS_ERROR, 14002, VCS_DEFAULT_FLAGS,
     "Resource could not be brought down because the
     attempt to remove the file(%s) failed with error(%d)",
     CHAR *)(*attr_val), errno);

VCSAG_CONSOLE_LOG_MSG(VCS_ERROR, 14002, VCS_DEFAULT_FLAGS,
     "Resource could not be brought down because, the
     attempt to remove the file(%s) failed with error(%d)",
     (CHAR *)(*attr_val), errno);
```

Agent Debug Logging Macros for C++ Entry Points

The macros VCSAG_RES_LOG_MSG and VCSAG_LOGDBG_MSG can be used within agent entry points to log debug messages of a specific severity level to the agent log.

The `LogDbg` attribute can be used to specify a debug message severity level. See the description of the `LogDbg` attribute ("LogDbg" on page 102). The `LogDbg` attribute can be set at the resource type level, and can be overridden to be set at the level for a specific resource.

The VCSAG_LOGDBG_MSG macro controls logging at the level of the resource type level, whereas VCSAG_RES_LOG_MSG macro can enable logging debug messages at the level of a specific resource.

The argument fields for the application logging macros are described in the following table:

`dbgsev`	Debug severity of the message. The values of `dbgsev` are macros ranging from VCS_DBG1 to VCS_DBG21; see "Severity Arguments for C++ Macros" on page 74.
`flags`	Describes the logging options. Default flags (0) prints UMI, NEWLINE. A macro, VCS_DEFAULT_FLAGS, represents the default value for the flags
`fmt`	A formatted string containing symbols. For example: "PathName is (%s)"
`variable_args`	Variable number (as many as 6) of type char, char * or integer

For example:

```
VCSAG_RES_LOG_MSG(VCS_DBG4, VCS_DEFAULT_FLAGS, "PathName is (%s)",
        (CHAR *)(*attr_val));
```

For the example shown, the specified message is logged to the agent log if the specific resource has been enabled (that is, the LogDbg attribute is set) for logging of debug messages at the severity level DBG4.

Severity Arguments for C++ Macros

A severity argument for a logging macro, for example, VCS_ERROR or VCS_DBG1, is in fact a macro itself that expands to include the following information:

- actual message severity
- function name
- name of the file that includes the function
- line number where the logging macro is expanded

For example, the application severity argument VCS_ERROR within the `monitor` entry point for the FileOnOff agent would expand to include the following information:

```
ERROR, file_monitor, FileOnOff.C, 28
```

Application severity macros map to application severities defined by the enum `VCSAgAppSev` and the debug severity macros map to severities defined by the enum `VCSAgDbgSev`. For example, in the `VCSAgApiDefs.h` function, these enumerated types are defined as:

```
enum VCSAgAppSev {
   AG_CRITICAL,
   AG_ERROR,
   AG_WARNING,
   AG_NOTICE,
   AG_INFORMATION
};

enum VCSAgDbgSev {
   DBG1,
   DBG2,
   DBG3,
   .
   .
   DBG21,
   AG_DBG_SEV_End
};
```

With the severity macros, agent developers need not specify the name of the function, the file name, and the line number in each log call. The name of the function, however, must be initialized by using the macro VCSAG_LOG_INIT. See "Initializing function_name Using VCSAG_LOG_INIT" on page 75.

Initializing function_name Using VCSAG_LOG_INIT

One requirement for logging of messages included in C++ functions is to initialize the *function_name* variable within *each* function. The macro, VCSAG_LOG_INIT, defines a local constant character string to store the function name:

```
VCSAG_LOG_INIT(func_name) const char *_function_name_ = func_name
```

For example, the function named “file_offline” would contain:

```
void file_offline (int, a, char *b)
{
    VCSAG_LOG_INIT(“file_offline”);
    .
    .
}
```

Note If the function name is not initialized with the VCSAG_LOG_INIT macro, when the agent is compiled, errors indicate that the name of the function is not defined.

See the “Examples of Logging APIs Used in a C++ Agent” on page 77 for more examples of the VCSAG_LOG_INIT macro.

Log Category

The log category for the agent is defined using the primitive `VCSAgSetLogCategory (cat_ID)` within the `VCSAgStartup` entry point. The log category is set to 2001 in the following example:

```
VCSEXPORT void VCSDECL VCSAgStartup()
{
        VCSAG_LOG_INIT("VCSAgStartup");
        VCSAgV40EntryPointStruct ep;

        ep.open                 = NULL;
        ep.close                = NULL;
        ep.monitor              = file_monitor;
        ep.online               = file_online;
        ep.offline              = file_offline;
        ep.clean                = file_clean;
        ep.attr_changed         = NULL;
        ep.shutdown             = NULL;
        ep.action               = NULL;
        ep.info                 = NULL;

        VCSAgSetLogCategory(2001);

        char *s = setlocale(LC_ALL, NULL);
        VCSAG_LOGDBG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS, "Locale is
          %s", s);

        VCSAgRegisterEPStruct(V40, &ep);
}
```

The log category for debug messages is by default 50, and does not need to be set.

Examples of Logging APIs Used in a C++ Agent

```
#include <stdio.h>
#include <locale.h>
#include "VCSAgApi.h"

void file_attr_changed(const char *res_name, const char
        *changed_res_name,const char *changed_attr_name, void
        **new_val)
{
       /*
        * NOT REQUIRED if the function is empty or is not logging
        * any messages to the agent log file
        */
        VCSAG_LOG_INIT("file_attr_changed");
}
extern "C" unsigned int
file_clean(const char *res_name, VCSAgWhyClean wc, void
        **attr_val)
{
        VCSAG_LOG_INIT("file_clean");
        if ((attr_val) && (*attr_val)) {
                if ((remove((CHAR *)(*attr_val)) == 0) || (errno ==
               ENOENT)) { return 0;       // Success
               }
        }
        return 1;       // Failure
}

void file_close(const char *res_name, void **attr_val)
{
        VCSAG_LOG_INIT("file_close");
}

//
// Determine if the given file is online (file exists) or
// offline (file does not exist).
//
extern "C" VCSAgResState
file_monitor(const char *res_name, void **attr_val, int
*conf_level)
{
        VCSAG_LOG_INIT("file_monitor");

        VCSAgResState state = VCSAgResUnknown;
        *conf_level = 0;
```

```
        /*
         * This msg will be printed for all resources if VCS_DBG4
         * is enabled for the resource type. Else it will be
         * logged only for that resource that has the dbg level
         * VCS_DBG4 enabled
         */

         VCSAG_RES_LOG_MSG(VCS_DBG4, VCS_DEFAULT_FLAGS, "PathName is
             (%s)", (CHAR *)(*attr_val));

         if ((attr_val) && (*attr_val)) {
               struct stat stat_buf;
               if ( (stat((CHAR *)(* attr_val), &stat_buf) == 0)
                      && (strlen((CHAR *)(* attr_val)) != 0) ) {
                      state = VCSAgResOnline; *conf_level = 100;

               }
               else {

                          state = VCSAgResOffline;
                          *conf_level = 0;
               }

          }
          VCSAG_RES_LOG_MSG(VCS_DBG7, VCS_DEFAULT_FLAGS, "State is
              (%d)", (int)state);
          return state;
}
extern "C" unsigned int
file_online(const char *res_name, void **attr_val) {
   int fd = -1;
   VCSAG_LOG_INIT("file_online");
   if ((attr_val) && (*attr_val)) {
      if (strlen((CHAR *)(* attr_val)) == 0) {
           VCSAG_LOG_MSG(VCS_WARNING, 3001, VCS_DEFAULT_FLAGS,
              "The value for PathName attribute is not specified");

           VCSAG_CONSOLE_LOG_MSG(VCS_WARNING, 3001,
             VCS_DEFAULT_FLAGS,
             "The value for PathName attribute is not specified");

           return 0;
         }
```

```
        if (fd = creat((CHAR *)(*attr_val), S_IRUSR|S_IWUSR) < 0) {

            VCSAG_LOG_MSG(VCS_ERROR, 3002, VCS_DEFAULT_FLAGS,
                "Resource could not be brought up because, "
                "the attempt to create the file(%s) failed  "
                "with error(%d)", (CHAR *)(*attr_val), errno);

            VCSAG_CONSOLE_LOG_MSG(VCS_ERROR, 3002,
                VCS_DEFAULT_FLAGS,
                "Resource could not be brought up because, "
                "the attempt to create the file(%s) failed "
                "with error(%d)", (CHAR *)(*attr_val), errno);
            return 0;
        }

        close(fd);
    }
    return 0;
}

extern "C" unsigned int
file_offline(const char *res_name, void **attr_val)
{
        VCSAG_LOG_INIT("file_offline");
        if ((attr_val) && (*attr_val) && (remove((CHAR*)
        (*attr_val)) != 0) && (errno != ENOENT)) {

         VCSAG_LOG_MSG(VCS_ERROR, 14002, VCS_DEFAULT_FLAGS,
             "Resource could not be brought down because, the
             attempt to remove the file(%s) failed with
             error(%d)", (CHAR *)(*attr_val), errno);

         VCSAG_CONSOLE_LOG_MSG(VCS_ERROR, 14002,
             VCS_DEFAULT_FLAGS, "Resource could not be brought
             down because, the attempt to remove the file(%s)
             failed with error(%d)", (CHAR *)(*attr_val), errno);
     }
     return 0;
}
```

```
void file_open(const char *res_name, void **attr_val)
{
        VCSAG_LOG_INIT("file_open");

}
VCSEXPORT void VCSDECL VCSAgStartup()
{
        VCSAG_LOG_INIT("VCSAgStartup");
        VCSAgV40EntryPointStruct ep;

        ep.open                 = NULL;
        ep.close                = NULL;
        ep.monitor              = file_monitor;
        ep.online               = file_online;
        ep.offline              = file_offline;
        ep.clean                = file_clean;
        ep.attr_changed         = NULL;
        ep.shutdown             = NULL;
        ep.action               = NULL;
        ep.info                 = NULL;

        VCSAgSetLogCategory(2001);

        char *s = setlocale(LC_ALL, NULL);
        VCSAG_LOGDBG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS, "Locale is
             %s", s);

        VCSAgRegisterEPStruct(V40, &ep);
}
```

Script Entry Point Logging Functions

For script based entry points, VCS 4.0 provides functions, or *wrappers*, to call the VCS `halog` command for message logging purposes. While the `halog` command can be called directly within the script to log messages, the following entry point logging functions are easier to use and less error-prone:

VCSAG_SET_ENVS - sets and exports entry point environment variables

VCSAG_LOG_MSG - passes normal agent message strings and parameters to the `halog` utility

VCSAG_LOGDBG_MSG - passes debug message strings and parameters to the `halog` utility

VCSAG_SET_ENVS

The VCSAG_SET_ENV function is used in each script-based entry point file. Its purpose is to set and export environment variables that identify the agent's category ID, the agent's name, the resource's name, and the entry point's name. With this information set up in the form of environment variables, the logging functions can handle messages and their arguments in the unified logging format without repetition within the scripts.

The VCSAG_SET_ENV function sets the following environment variables for a resource:

VCSAG_LOG_CATEGORY	Sets the category ID. For custom agents, VERITAS assigns the category ID. See the category ID description in "UMI" on page 70. NOTE: For VCS bundled agents, the category ID is pre-assigned, based on the platform (Solaris, Linux, AIX, HP-UX, or Windows) for which the agent is written.
VCSAG_LOG_AGENT_NAME	The absolute path to the agent. For example: `/opt/VRTSvcs/bin/`*Application* Since the entry points are invoked using their absolute paths, this environment variable is set at invocation. If the agent developer wishes, this agent name can also be hardcoded and passed as an argument to the VCSAG_SET_ENVS function
VCSAG_LOG_SCRIPT_NAME	The absolute path to the entry point script. For example: `/opt/VRTSvcs/bin/`*Application*`/online` Since the entry points are invoked using their absolute paths, this environment variable is set at invocation. The script name variable is overridable.
VCSAG_LOG_RESOURCE_NAME	The resource is specified in the call within the entry point: VCSAG_SET_ENVS $*resource_name*

VCSAG_SET_ENVS Examples, Shell Script Entry Points

The VCSAG_SET_ENVS function must be called before any of the other logging functions.

- A minimal call:

```
VCSAG_SET_ENVS ${resource_name}
```

- Setting the category ID:

```
VCSAG_SET_ENVS ${resource_name} ${category_ID}
VCSAG_SET_ENVS ${resource_name} 1062
```

- Overriding the default script name:

```
VCSAG_SET_ENVS ${resource_name} ${script_name}
VCSAG_SET_ENVS ${resource_name} "monitor"
```

- Setting the category ID and overriding the script name:

```
VCSAG_SET_ENVS ${resource_name} ${script_name} ${category_id}
VCSAG_SET_ENVS ${resource_name} "monitor" 1062
```

Or,

```
VCSAG_SET_ENVS ${resource_name} ${category_id} ${script_name}
VCSAG_SET_ENVS ${resource_name} 1062 "monitor"
```

VCSAG_SET_ENVS Examples, Perl Script Entry Points

- A minimal call:

```
VCSAG_SET_ENVS ($resource_name);
```

- Setting the category ID:

```
VCSAG_SET_ENVS ($resource_name, $category_ID);
VCSAG_SET_ENVS ($resource_name, 1062);
```

- Overriding the script name:

```
VCSAG_SET_ENVS ($resource_name, $script_name);
VCSAG_SET_ENVS ($resource_name, "monitor");
```

- Setting the category ID and overriding the script name:

```
VCSAG_SET_ENVS ($resource_name, $script_name, $category_id);
VCSAG_SET_ENVS ($resource_name, "monitor", 1062);
```

Or,

```
VCSAG_SET_ENVS ($resource_name, $category_id, $script_name);
VCSAG_SET_ENVS ($resource_name, 1062, "monitor");
```

VCSAG_LOG_MSG

The VCSAG_LOG_MSG function can be used to pass normal agent messages to the `halog` utility. At a minimum, the function must include the severity, the message within quotes, and a message ID. Optionally, the function can also include parameters and specify an encoding format.

Severity Levels (*sev*)	"C" - critical, "E" - error, "W" - warning, "N" - notice, "I" - information; place error code in quotes
Message (*msg*)	A text message within quotes; for example: "One file copied"
Message ID (*msgid*)	An integer between 0 and 65535
Encoding *Format*	UTF-8, ASCII, or UCS-2 in the form: "-encoding *format*"
Parameters	Parameters (up to six), each within quotes

VCSAG_LOG_MSG Examples, Shell Script Entry Points

◆ Calling a function without parameters or encoding format:

```
VCSAG_LOG_MSG "<sev>" "<msg>" <msgid>
VCSAG_LOG_MSG "C" "Two files found" 140
```

◆ Calling a function with one parameter, but without encoding format:

```
VCSAG_LOG_MSG "<sev>" "<msg>" <msgid> "<param1>"
VCSAG_LOG_MSG "C" "$count files found" 140 "$count"
```

◆ Calling a function with a parameter and encoding format:

```
 VCSAG_LOG_MSG "<sev>" "<msg>" <msgid> "-encoding <format>"
"<param1>"
 VCSAG_LOG_MSG "C" "$count files found" 140 "-encoding utf8"
"$count"
```

Note that if encoding format and parameters are passed to the functions, the encoding format must be passed before any parameters.

VCSAG_LOG_MSG Examples, Perl Script Entry Points

- Calling a function without parameters or encoding format:

```
VCSAG_LOG_MSG ("<sev>", "<msg>", <msgid>);
VCSAG_LOG_MSG ("C", "Two files found", 140);
```

- Calling a function with one parameter, but without encoding format:

```
VCSAG_LOG_MSG ("<sev>", "<msg>", <msgid>, "<param1>";
VCSAG_LOG_MSG ("C", "$count files found", 140, "$count");
```

- Calling a a function with one parameter and encoding format:

```
 VCSAG_LOG_MSG ("<sev>", "<msg>", <msgid>, "-encoding <format>",
"<param1>");
 VCSAG_LOG_MSG ("C", "$count files found", 140, "-encoding utf8",
"$count");
```

Note that if encoding format and parameters are passed to the functions, the encoding format must be passed before any parameters.

VCSAG_LOGDBG_MSG

This function can be used to pass debug messages to the `halog` utility. At a minimum, the severity must be indicated along with a message. Optionally, the encoding format and parameters may be specified.

Severity (*dbg*)	An integer indicating a severity level, 1 to 21.
Message (*msg*)	A text message in quotes; for example: “One file copied”
Encoding *Format*	UTF-8, ASCII, or UCS-2 in the form: “-encoding *format*”
Parameters	Parameters (up to six), each within quotes

VCSAG_LOGDBG_MSG Examples, Shell Script Entry Points

- Calling a function without encoding or parameters:

```
VCSAG_LOGDBG_MSG <dbg> "<msg>"
VCSAG_LOGDBG_MSG 1 "This is string number 1"
```

- Calling a function with a parameter, but without encoding format:

```
VCSAG_LOGDBG_MSG <dbg> "<msg>" "<param1>"
VCSAG_LOGDBG_MSG 2 "This is string number $count" "$count"
```

- Calling a function with a parameter and encoding format:

```
VCSAG_LOGDBG_MSG <dbg> "<msg>" "-encoding <format>" "$count"
VCSAG_LOGDBG_MSG 2 "This is string number $count" "$count"
```

VCSAG_LOGDBG_MSG Examples, Perl Script Entry Points

- Calling a function:

```
VCSAG_LOGDBG_MSG (<dbg>, "<msg>");
VCSAG_LOGDBG_MSG (1 "This is string number 1");
```

- Calling a function with a parameter, but without encoding format:

```
VCSAG_LOGDBG_MSG (<dbg>, "<msg>", "<param1>");
VCSAG_LOGDBG_MSG (2, "This is string number $count", "$count");
```

- Calling a function with a parameter and encoding format:

```
VCSAG_LOGDBG_MSG <dbg> "<msg>" "-encoding <format>" "<param1>"
VCSAG_LOGDBG_MSG (2, "This is string number $count", "-encoding
   utf8", "$count");
```

Using the Functions in Scripts

The script-based entry points require a line that specifies the file defining the logging functions. Include the following line exactly once in each script. The line should precede the use of any of the log functions.

- Shell Script include file

```
. ${VCS_HOME:-/opt/VRTSvcs}/bin/ag_i18n_inc.sh
```

- Perl Script include file

```
use ag_i18n_inc;
```

Example of Logging Functions Used in Script Agent

The following example shows the use of VCSAG_SET_ENVS and VCSAG_LOG_MSG functions in a shell script for the `online` entry point.

```
!#/bin/ksh

ResName=$1

# Parse other input arguments
 :
 :
VCSHOME="${VCS_HOME:-/opt/VRTSvcs}"

. $VCSHOME/bin/ag_i18n_inc.sh

# Assume the category id assigned by VERITAS for this custom agent
#is 10061
VCSAG_SET_ENVS $ResName 10061

# Online entry point processing
 :
 :

# Successful completion of the online entry point
VCSAG_LOG_MSG "N" "online succeeded for resource $ResName" 1
"$ResName"

exit 0
```

Building a Custom VCS Agent 6

The `VRTSvcs` package installed by the VCS installation program includes the following files to facilitate agent development. Note that custom agents are not supported by VERITAS Technical Support.

Script Agents

Description	Pathname
Ready-to-use VCS agent that includes a built-in implementation of the `VCSAgStartup` entry point.	$VCS_HOME/bin/ScriptAgent ScriptAgent cannot be used with C++ entry points.

C++ Agents

Description	Pathname
Directory containing a sample C++ agent and `Makefile`.	$VCS_HOME/src/agent/Sample
Sample `Makefile` for building a C++ agent.	$VCS_HOME/src/agent/Sample/Makefile
Entry point templates for C++ agents.	$VCS_HOME/src/agent/Sample/`agent.C`

Compiling is not required if all entry points are implemented using scripts. A copy of ScriptAgent is sufficient.

Compiling is required to build the agent if any entry points are implemented using C++. We recommend the following procedures for developers implementing entry points using C++:

1. Edit `agent.C` to customize the implementation; `agent.C` is located in the directory `$VCS_HOME/src/agent/Sample`.

2. After completing the changes to `agent.C`, invoke the make command to build the agent. The command is invoked from `$VCS_HOME/src/agent/Sample`, where the Makefile is located.

Additional Recommendations

We also recommend naming the agent binary `resource_typeAgent`. Place the agent in the directory `$VCS_HOME/bin/resource_type`.

The agent binary for Oracle would be `$VCS_HOME/bin/Oracle/OracleAgent`, for example. If the agent file is different, for example `/foo/ora_agent`, the `types.cf` file must contain the following entry:

```
...
 Type Oracle (
   ...
   static str AgentFile = "/foo/ora_agent"
   ...
)
```

If entry points are implemented using scripts, the script file must be placed in the directory `$VCS_HOME/bin/`*`resource_type`*. It must be named correctly (if necessary, review "Script Agents" on page 16).

If all entry points are scripts, all scripts should be in the directory `$VCS_HOME/bin/`*`resource_type`*. Copy the ScriptAgent into the agent directory as `$VCS_HOME/bin/`*`resource_type`*`/`*`resource_type`*`Agent`.

For example, if the online entry point for Oracle is implemented using Perl, the online script must be: `$VCS_HOME/bin/Oracle/online`.

Building a VCS Agent for FileOnOff Resources

The following sections describe different ways to build a VCS agent for “FileOnOff” resources. For test purposes, instructions for installing the agent on a single VCS system are also provided. For multi-system configurations, you must install the agent on each system in the cluster.

The examples assume that VCS is installed under `/opt/VRTSvcs`. If your VCS installation directory is different, change the commands accordingly.

A FileOnOff resource represents a regular file. The FileOnOff `online` entry point creates the file if it does not already exist. The FileOnOff `offline` entry point deletes the file. The FileOnOff `monitor` entry point returns online and confidence level 100 if the file exists; otherwise, it returns offline. The examples in this chapter use the following type and resource definitions:

```
// Define the resource type called FileOnOff (in
FileOnOffTypes.cf).
type FileOnOff (
str PathName;
static str ArgList[] = { PathName };
)

// Define a FileOnOff resource (in main.cf).
include FileOnOffTypes.cf
FileOnOff FileOnOffRes (
  PathName =  "/tmp/VRTSvcs_file1"
  Enabled = 1
)
```

The resource name and `ArgList` attribute values are passed to the script entry points as command-line arguments. For example, in the preceding configuration, script entry points receive the resource name as the first argument, and PathName as the second.

Using Script Entry Points

The following example shows how to build the FileOnOff agent without writing and compiling any C++ code. This example implements the `online`, `offline`, and `monitor` entry points only.

1. Create the directory `/opt/VRTSvcs/bin/FileOnOff`:

    ```
    # mkdir /opt/VRTSvcs/bin/FileOnOff
    ```

2. Use the VCS agent `/opt/VRTSvcs/bin/ScriptAgent` as the FileOnOff agent. Copy this file to `/opt/VRTSvcs/bin/FileOnOff/FileOnOffAgent`, or create a link.

   To copy the agent binary:

   ```
   # cp /opt/VRTSvcs/bin/ScriptAgent
     /opt/VRTSvcs/bin/FileOnOff/FileOnOffAgent
   ```

   To create a link to the agent binary:

   ```
   # ln -s /opt/VRTSvcs/bin/ScriptAgent
     /opt/VRTSvcs/bin/FileOnOff/FileOnOffAgent
   ```

3. Implement the `online`, `offline`, and `monitor` entry points using scripts.

   a. Using any editor, create the file `/opt/VRTSvcs/bin/FileOnOff/online` with the contents:

   ```
   # !/bin/sh
   # Create the file specified by the PathName attribute.
   touch $2
   ```

   b. Create the file `/opt/VRTSvcs/bin/FileOnOff/offline` with the contents:

   ```
   # !/bin/sh
   # Remove the file specified by the PathName attribute.
   rm $2
   ```

   c. Create the file `/opt/VRTSvcs/bin/FileOnOff/monitor` with the contents:

   ```
   # !/bin/sh
   # Verify file specified by the PathName attribute exists.
   if test -f $2
   then exit 110;
   else exit 100;
   fi
   ```

4. Additionally, you can implement the `info` and `action` entry points. For the action entry point, create a subdirectory named “actions” under the agent directory, and create scripts with the same names as the action_tokens within the subdirectory.

Using VCSAgStartup() and Script Entry Points

The following example shows how to build the FileOnOff agent using your own `VCSAgStartup` entry point. This example implements the `VCSAgStartup`, `online`, `offline`, and `monitor` entry points only.

1. Create the directory /opt/VRTSvcs/src/agent/FileOnOff:

   ```
   # mkdir /opt/VRTSvcs/src/agent/FileOnOff
   ```

2. Copy the contents from the directory /opt/VRTSvcs/src/agent/Sample to the directory you created in the previous step:

   ```
   # cp /opt/VRTSvcs/src/agent/Sample/*
      /opt/VRTSvcs/src/agent/FileOnOff
   ```

3. Change to the new directory:

   ```
   cd /opt/VRTSvcs/src/agent/FileOnOff
   ```

4. Edit the file agent.C and modify the VCSAgStartup() function (the last several lines) to match the following example:

   ```
   void VCSAgStartup() {
      VCSAgV40EntryPointStruct ep;

      // Set all the entry point fields to NULL because
      // this example does not implement any of them
      // using C++.

      ep.open = NULL;
      ep.close = NULL;
      ep.monitor = NULL;
      ep.online = NULL;
      ep.offline = NULL;
      ep.attr_changed = NULL;
      ep.clean = NULL;
      ep.shutdown = NULL;
      ep.action = NULL;
      ep.info = NULL;
      VCSAgSetLogCategory(10041);
      VCSAgRegisterEPStruct(V40, &ep);
   }
   ```

5. Compile agent.C and build the agent by invoking make. (Makefile is provided.)

   ```
   # make
   ```

6. Create the directory /opt/VRTSvcs/bin/FileOnOff:

   ```
   # mkdir /opt/VRTSvcs/bin/FileOnOff
   ```

7. Install the FileOnOff agent built in step 5.

   ```
   # make install AGENT=FileOnOff
   ```

8. Implement the `online`, `offline`, and `monitor` entry points, as instructed in step 3 on page 90.

Using C++ and Script Entry Points

The following example shows how to build the FileOnOff agent using your own `VCSAgStartup` entry point, the C++ version of the `monitor` entry point, and the script version of `online` and `offline` entry points. This example implements the `VCSAgStartup`, `online`, `offline`, and `monitor` entry points only.

1. Create the directory `/opt/VRTSvcs/src/agent/FileOnOff`:

```
# mkdir /opt/VRTSvcs/src/agent/FileOnOff
```

2. Copy the contents from the directory `/opt/VRTSvcs/src/agent/Sample` to the directory you created in the previous step:

```
# cp /opt/VRTSvcs/src/agent/Sample/*
  /opt/VRTSvcs/src/agent/FileOnOff
```

3. Change to the new directory:

```
# cd /opt/VRTSvcs/src/agent/FileOnOff
```

4. Edit the file `agent.C` and modify the `VCSAgStartup()` function (the last several lines) to match the following example:

```
void VCSAgStartup()
{
VCSAgV40EntryPointStruct ep;

// This example implements only the monitor entry
// point using C++. Set all the entry point
// fields, except monitor, to NULL.
```

```
ep.open = NULL;
ep.close = NULL;
ep.monitor = file_monitor;
ep.online = NULL;
ep.offline = NULL;
ep.attr_changed = NULL;
ep.clean = NULL;
ep.shutdown = NULL;
ep.action = NULL;
ep.info = NULL;
VCSAgSetLogCategory(10041);
VCSAgRegisterEPStruct(V40, &ep);
}
```

5. Modify the `file_monitor()` function:

```
// This is a C++ implementation of the monitor entry
// point for the FileOnOff resource type. This function
// determines the status of a FileOnOff resource by
// checking if the corresponding file exists. It is
// assumed that the complete pathname of the file will
// be passed as the first ArgList attribute.

VCSAgResState file_monitor(const char *res_name, void
   **attr_val,int *conf_level) {
// Initialize the OUT parameters.
VCSAgResState state = VCSAgResUnknown;
*conf_level = 0;

if (attr_val) {
  // Get the pathname of the file.
  const char *path_name = (const char *) attr_val[0];
  // Determine if the file exists.
  struc stat stat_buf;
  if (stat(path_name, &stat_buf) == 0) {
    state = VCSAgResOnline;
    *conf_level = 100;
  }
  else {
    state = VCSAgResOffline;
    *conf_level = 0;
  }
}

// Return the status of the resource.

return state;
}
```

6. Compile `agent.C` and build the agent by invoking `make`. (`Makefile` is provided.)

```
# make
```

7. Create the directory `/opt/VRTSvcs/bin/FileOnOff`:

```
# mkdir /opt/VRTSvcs/bin/FileOnOff
```

8. Install the `FileOnOff` agent built in step 6.

```
# make install AGENT=FileOnOff
```

Note Implement the `online` and `offline` entry points as instructed in step 3 on page 90.

Using C++ Entry Points

The example in this section shows how to build the FileOnOff agent using your own `VCSAgStartup` entry point and the C++ version of `online`, `offline`, and `monitor` entry points. This example implements the `VCSAgStartup`, `online`, `offline`, and `monitor` entry points only.

1. Edit the file `agent.C` and modify the `VCSAgStartup()` function (the last several lines) to match the following example:

```
void VCSAgStartup() {
  VCSAgV40EntryPointStruct ep;

  // This example implements online, offline, and monitor
  // entry points using C++. Set the corresponding fields
  // of VCSAgV40EntryPointStruct passed to VCSAgRegisterEPStruct.
  // Set all other fields to NULL.

  ep.open = NULL;
  ep.close = NULL;
  ep.monitor = file_monitor;
  ep.online = file_online;
  ep.offline = file_offline;
  ep.attr_changed = NULL;
  ep.clean = NULL;
  ep.shutdown = NULL;
  ep.action = NULL;
  ep.info = NULL;

  VCSAgSetLogCategory(2001);

  VCSAgRegisterEPStruct(V40, &ep);
}
```

2. Modify `file_online()` and `file_offline()`:

```
  // This is a C++ implementation of the online entry
  // point for the FileOnOff resource type. This function
  // brings online a FileOnOff resource by creating the
  // corresponding file. It is assumed that the complete
  // pathname of the file will be passed as the first
  // ArgList attribute.

unsigned int
file_online(const char *res_name, void **attr_val) {
   VCSAG_LOG_INIT(file_online);
  if (attr_val) {
    // Get the pathname of the file.
    const char *path_name = (const char *) attr_val[0];


  // Create the file
    int fd = creat (path_name,S_IRUSR | S_IWUSR);
    if (fd < 0) {
      // if creat() failed, send a log message to
      // the console.
      char msg [1024];
        VCSAG_LOG_MSG(VCS_ERROR, 1001, VCS_DEFAULT_FLAGS, "creat ()
        "failed for for file(%s)", path_name);
    }
    else {
      close(fd);
    }
  }

  // Completed onlining resource. Return 0 so monitor
  // can start immediately. Note that return value
  // indicates how long agent framework must wait before
  // calling the monitor entry point to check if online
  // was successful.

  return 0;
}

// This is a C++ implementation of the offline entry
// point for the FileOnOff resource type. This function
// takes offline a FileOnOff resource by deleting the
// corresponding file. It is assumed that the complete
// pathname of the file will be passed as the first
// ArgList attribute.

unsigned int
file_offline(const char *res_name, void **attr_val) {
```

```
  VCSAG_LOG_INIT("file_offline");
  if (attr_val) {
    // Get the pathname of the file.
    const char *path_name = (const char *)
                  attr_val[0];

    // Delete the file
    remove (path_name);
}
  // Completed offlining resource. Return 0 so monitor
  // can start immediately. Note that return value
  // indicates how long agent framework must wait before
  // calling the monitor entry point to check if offline
  // was successful.

  return 0;
  }
```

3. Modify `file_monitor()`, as shown on step 5 on page 93.

4. Compile `agent.C` and build the agent by invoking `make`. (`Makefile` is provided.)

   `#` **`make`**

5. Create the directory `/opt/VRTSvcs/bin/FileOnOff`:

   `#` **`mkdir  /opt/VRTSvcs/bin/FileOnOff`**

6. Install the FileOnOff agent built in step 4.

   `#` **`make install AGENT=FileOnOff`**

Setting Agent Attributes 7

The VCS agents can be customized for a resource type by setting the values of the agent attributes. Agent attributes are predefined static attributes of the resource type. They can be assigned values when defining the resource type in `types.cf`, and they can be set dynamically using the command `hatype -modify` (described in the *VERITAS Cluster Server User's Guide*).

Note VCS allows you to specify priorities and scheduling classes for VCS processes. For details on the additional attributes included with this feature, and for instructions on initializing them in the `types.cf` file or setting them from the command line, see "Scheduling Class and Priority Configuration Support" on page 110.

Overriding Static Attributes

Typically, the value of a static attribute of a resource type applies to all resources of the type. However, the value of a static attribute for a specific resource may be modified (overridden) without affecting the value of the attribute for other resources of that type. In this chapter, the description of each agent attribute indicates whether the attribute's values are overridable.

Users can override the values of overridable static attributes two ways:

- By explicitly defining the attribute in a resource definition in the `main.cf` configuration file
- By using the `hares` command from the command line with the `-override` option

The values of the overridden attributes may be displayed using the `hares -display` command. The overridden values of static attributes may be removed by using the `hares -undo_override` option from the command line.

See `hares` manual page and the *VERITAS Cluster Server User's Guide* for a additional information about overriding the values of static attributes.

Agent Attribute Definitions

Each agent attribute is listed and defined in the following sections.

ActionTimeout

After the `hares -action` command has instructed the agent to perform a specified action, the action entry point has the time specified by the `ActionTimeout` attribute (scalar-integer) to perform the action. The value of `ActionTimeout` may be set for individual resources. The default is 40 seconds. The `ActionTimeout` attribute value can be overridden.

AgentFile

The default name of the agent file to be executed is:

`$VCS_HOME/bin/`*`resource_type`*`/`*`resource_type`*`Agent.`

The `AgentFile` attribute value cannot be overridden.

AgentReplyTimeout

The engine restarts the agent if it has not received the periodic heartbeat from the agent for the number of seconds specified by this attribute. The default value of 130 seconds works well for most configurations. Increase this value if the engine is restarting the agent. This may occur when the system is heavily loaded or if the number of resources exceeds three or four hundred. (See the command `haagent -display` in the chapter on administering VCS from the command line in the *VERITAS Cluster Server User's Guide.*) Note that the engine will also restart a crashed agent. The `AgentReplyTimeout` attribute value cannot be overridden.

AgentStartTimeout

After the engine has started the agent, this is the amount of time the engine waits for the initial agent "handshake" before attempting to restart. Default is 60 seconds. The `AgentStartTimeout` attribute value cannot be overridden.

ArgList

An ordered list of attributes whose values are passed to the `open`, `close`, `online`, `offline`, `monitor`, and `clean` entry points. Default is empty list. The `ArgList` attribute value cannot be overridden.

ArgList Reference Attributes

Reference attributes refer to attributes of a different resource. If the value of a resource attribute is defined as the name of another resource, the `ArgList` of the first resource can refer to an attribute of the second resource using the `:` operator.

For example, if the resource `ArgList` resembles the following code sample (in which the value of `attr3` is the name of another resource), the entry points are passed the values of the `attr1`, `attr2` attributes of the first resource, and the value of the `attr_A` attribute of the second resource.

```
{ attr1, attr2, attr3:attr_A }
```

AttrChangedTimeout

Maximum time (in seconds) within which the `attr_changed` entry point must complete or else be terminated. Default is 60 seconds. The `AttrChangedTimeout` attribute value can be overridden.

CleanTimeout

Maximum time (in seconds) within which the `clean` entry point must complete or else be terminated. Default is 60 seconds. The `CleanTimeout` attribute value can be overridden.

CloseTimeout

Maximum time (in seconds) within which the `close` entry point must complete or else be terminated. Default is 60 seconds.The `CloseTimeout` attribute value can be overridden.

ComputeStats

An attribute that indicates whether or not VCS is to keep track of monitor statistics for a given resource. A value of 1 indicates VCS is to keep track of the monitor time for the resource. A value of 0 indicates that VCS is not keep track of monitor statistics for a resource. See “MonitorStatsParam” on page 104. The default is 0.

ConfInterval

Specifies an interval in seconds. When a resource has remained online for the designated interval (all `monitor` invocations during the interval reported ONLINE), any earlier faults or restart attempts of that resource are ignored. This attribute is used with ToleranceLimit to allow the `monitor` entry point to report OFFLINE several times before the resource is declared FAULTED. If `monitor` reports OFFLINE more often than the number set in ToleranceLimit, the resource is declared FAULTED. However, if the resource remains online for the interval designated in ConfInterval, any earlier reports of OFFLINE are not counted against ToleranceLimit.

The agent framework uses the values of MonitorInterval (MI), MonitorTimeout (MT), and ToleranceLimit (TL) to determine how low to set the value of ConfInterval. The agent framework ensures that ConfInterval (CI) cannot be less than that expressed by the following relationship:

(MI + MT) * TL + MI + 10

Lesser specified values of ConfInterval are ignored. For example, assume that the values are 60 for MI, 60 for MT, and 0 for TL. If you specify any value lower than 70 for CI, the agent framework ignores the specified value and sets the value to 70. However, you can successfully specify and set CI to any value over 70.

ConfInterval is also used with RestartLimit to prevent VCS from restarting the resource indefinitely. VCS attempts to restart the resource on the same system according to the number set in RestartLimit within ConfInterval before giving up and failing over. However, if the resource remains online for the interval designated in ConfInterval, earlier attempts to restart are not counted against RestartLimit. Default is 600 seconds. The `ConfInterval` attribute value can be overridden.

FaultOnMonitorTimeouts

Indicates the number of consecutive monitor failures to be treated as a resource fault. A monitor attempt is considered a failure if it does not complete within the time specified by the MonitorTimeout attribute. When a monitor fails as many times as the value specified by this attribute, the corresponding resource is brought down by calling the `clean` entry point. The resource is then marked FAULTED, or it is restarted, depending on the value set in the Restart Limit attribute.

When FaultOnMonitorTimeouts is set to 0, monitor failures are not considered indicative of a resource fault. Default is 4. The `FaultOnMonitorTimeouts` attribute value can be overridden.

Note This attribute applies only to online resources. If a resource is offline, no special action is taken during monitor failures.

FireDrill

A "fire drill" refers to the process of bringing up a database or application on a secondary or standby system for the purpose of doing some processing on the secondary data, or to verify that the application is capable of onlining on the secondary in case of a primary fault. The `FireDrill` attribute specifies whether a resource type has fire drill enabled or not. A value of 1 for the `FireDrill` attribute indicates a fire drill is enabled. A value of 0 indicates a fire drill is not enabled. The default is 0.

Refer to the *VERITAS Cluster Server User's Guide* for details of how to set up and implement a fire drill.

The `FireDrill` attribute cannot be overridden.

InfoInterval

Specifies the interval, in seconds, between successive invocations of the `info` entry point for a given resource. The default value of the `InfoInterval` attribute is 0, which specifies that the agent framework is not to schedule the `info` entry point periodically; the entry point can be invoked, however, by the user from the command line. The `InfoInterval` attribute value can be overridden.

InfoTimeout

A scalar integer specifying the time the agent framework is to allow for completion of the `info` entry point. The `InfoTimeout` attribute value can be overridden. The default is 30 seconds.

LogDbg

LogDbg is a resource type-level attribute that specifies which debug messages are to be logged to the agent log. The LogDbg attribute applies only for those agent entry points written in C++.

By default, LogDbg is an empty list, meaning that no debug messages are logged for a resource type. Users can modify this attribute for a given resource type, specifying that debug messages of specific severities be printed to the agent log.

For example, if you want to log debug messages for the FileOnOff resource type with severity levels DBG_3 and DBG_4, use the hatype command:

```
# hatype -modify FileOnOff LogDbg -add DBG_3 DBG_4
# hatype -display FileOnOff -attribute LogDbg
TYPE         ATTRIBUTE          VALUE
FileOnOff    LogDbg             DBG_3  DBG_4
```

The FileOnOff agent entry points can now log debug messages with severity DBG_3 and DBG_4. They are printed to the agent log file. An example line in the agent log would now resemble:

```
.
.
2003/06/06 11:02:35 VCS DBG_3 V-16-50-0 FileOnOff:f1:monitor:This
is a debug message
       FileOnOff.C:file_monitor[28]
```

LogDbg is an overrideable attribute. For example, for a specific critical resource, the attribute value can be set to obtain debug messages of particular severity level in addition to those severity levels already set for the resource type. From the command line, this can be done using the hares command. For example:

```
# hares -override f1 LogDbg
# hares -modify f1 LogDbg DBG_5
# hares -display f1 -attribute LogDbg
Resource      Attribute        System         Value
f1            LogDbg           global         DBG_5
```

The FileOnOff agent log would now include debug messages for the f1 resource at severity level DBG_5 in addition to debug messages at the severity levels DBG_3 and DBG_4 enabled for the resource type.

The LogDbg attribute value can be overridden.

Note Values of LogDbg overridden for a resource are effective only if the agent uses the macro VCSAG_RES_LOG_MSG, which is the only macro that checks if a particular debug severity is enabled for the resource before writing the message to the log file. Please refer to "Logging Agent Messages" on page 69 for more information.

LogFileSize

Sets the size of an agent log file. Value must be specified in bytes. Minimum is 65536 bytes (64KB). Maximum is 134217728 bytes (128MB). Default is 33554432 bytes (32MB). For example,

```
# hatype -modify FileOnOff LogSize 2097152
```

Values specified less than the minimum acceptable value will be changed 65536 bytes. Values specified greater than the maximum acceptable value will be changed to 134217728 bytes. Therefore, out-of-range values displayed for the command:

```
# hatype -display restype -attribute LogSize
```

will be those entered with the `-modify` option, not the actual values. The `LogFileSize` attribute value cannot be overridden.

ManageFaults

A service group level attribute. `ManageFaults` specifies if VCS manages resource failures within the service group by calling `clean` entry point for the resources. This attribute value can be set to ALL or NONE. Default = ALL.

If set to NONE, VCS does not call `clean` entry point for any resource in the group. User intervention is required to handle resource faults/failures. When `ManageFaults` is set to NONE and one of the following events occur, the resource enters the ADMIN_WAIT state:

1 - The `offline` entry point did not complete within the expected time. Resource state is ONLINE|ADMIN_WAIT

2 - The `offline` entry point was ineffective. Resource state is ONLINE|ADMIN_WAIT

3 - The `online` entry point did not complete within the expected time. Resource state is OFFLINE|ADMIN_WAIT

4 - The `online` entry point was ineffective. Resource state is OFFLINE|ADMIN_WAIT

5 - The resource was taken offline unexpectedly. Resource state is OFFLINE|ADMIN_WAIT

6 - For the online resource the `monitor` entry point consistently failed to complete within the expected time. Resource state is ONLINE| MONITOR_TIMEDOUT|ADMIN_WAIT

MonitorInterval

Duration (in seconds) between two consecutive monitor calls for an ONLINE or transitioning resource. Default is 60 seconds. The `MonitorInterval` attribute value can be overridden.

MonitorStatsParam

Refer to the *VERITAS Cluster Server User's Guide* for details about the `MonitorStatsParam` attribute, and the `MonitorTimeStats` attribute that is updated by VCS. See also "ComputeStats" on page 99.

`MonitorStatsParam` is a resource type-level attribute, which stores the required parameter values for calculating monitor time statistics.

```
static str MonitorStatsParam = { Frequency = 10, ExpectedValue =
  3000, ValueThreshold = 100, AvgThreshold = 40 }
```

- *Frequency*: Defines the number of monitor cycles after which the average monitor cycle time should be computed and sent to the engine. The value for this attribute is must be between 1 and 30 and is set to 0 by default.
- *ExpectedValue*: The expected monitor time in milliseconds for all resources of this type. Default=3000.
- *ValueThreshold*: The acceptable percentage difference between the expected monitor cycle time (*ExpectedValue*) and the actual monitor cycle time. Default=100.
- *AvgThreshold*: The acceptable percentage difference between the benchmark average and the moving average of monitor cycle times. Default=40.

The `MonitorStatsParam` attribute values can be overridden.

MonitorTimeout

Maximum time (in seconds) within which the `monitor` entry point must complete or else be terminated. Default is 60 seconds. The `MonitorTimeout` attribute value can be overridden.

The determination of the value of the `MonitorTimeout` attribute can be assisted by the use of the `MonitorStatsParam` attribute.

NumThreads

Number of threads used within the agent process for managing resources. This number does not include the three threads used for other internal purposes. Default is 10 threads. The `NumThreads` attribute value cannot be overridden.

OfflineMonitorInterval

Duration (in seconds) between two consecutive monitor calls for an OFFLINE resource. If set to 0, OFFLINE resources are not monitored. Default is 300 seconds. The `OfflineMonitorInterval` attribute value can be overridden.

Note In previous releases, agents monitored offline resources once every minute by default. To reduce monitoring overhead, this frequency was changed to once every five minutes. This interval may be adjusted to meet specific configuration requirements.

OfflineTimeout

Maximum time (in seconds) within which the `offline` entry point must complete or else be terminated. Default is 300 seconds. The `OfflineTimeout` attribute value can be overridden.

OnlineRetryLimit

Number of times to retry `online`, if the attempt to online a resource is unsuccessful. This attribute is meaningful only if `clean` is implemented. Default is 0. The `OnlineRetryLimit` attribute value can be overridden.

OnlineTimeout

Maximum time (in seconds) within which the `online` entry point must complete or else be terminated. Default is 300 seconds. The `OnlineTimeout` attribute value can be overridden.

OnlineWaitLimit

Number of monitor intervals to wait after completing the online procedure, and before the resource becomes online. If the resource is not brought online after the designated monitor intervals, the online attempt is considered ineffective. This attribute is meaningful only if the `clean` entry point is implemented.

If clean is not implemented, the agent continues to periodically run `monitor` until the resource comes online.

If `clean` is implemented, when the agent reaches the maximum number of monitor intervals it assumes that the online procedure was ineffective and runs `clean`. The agent then notifies the engine that the online failed, or retries the procedure, depending on whether or not the OnlineRetryLimit is reached. Default is 2. The `OnlineWaitLimit` attribute value can be overridden.

OpenTimeout

Maximum time (in seconds) within which the `open` entry point must complete or else be terminated. Default is 60 seconds. The `OpenTimeout` attribute value can be overridden.

Operations

Indicates the valid operations of resources of the resource type. The values are OnOff (can be onlined and offlined), OnOnly (can be onlined only), and None (cannot be onlined or offlined). Default is OnOff. The `Operations` attribute value cannot be overridden.

ResourceInfo

The `ResourceInfo` (association-string) is a temporary attribute, the scope of which is set by the engine to be global for failover groups or local for parallel groups. Because `ResourceInfo` is a temporary attribute, its values are never dumped to the `main.cf` file.

The values of the `ResourceInfo` attribute are expressed in three mandatory keys: `State, Msg, and TS.` For `State`, the possible values are "Valid," (the default), "Invalid," and "Stale"). `Msg` (default is "", a null string) contains the output from the entry point. `TS` contains the time at which the attribute is has been updated or modified. These mandatory keys are updated only by the agent framework, not the entry point. The entry point can define and add other keys (name-value pairs) and update them.

The value of the `ResourceInfo` attribute can be displayed using the `hares` command. The output of `hares -display` shows the first 20 characters of the current value; the output of `hares -value` *`resource`* `ResourceInfo` shows all name-value pairs in the keylist.

The resource for which the `info` entry point is invoked must be online. When a resource goes offline or faults, the `State` key is marked "Stale" because the information is not current. If the `info` entry point exits abnormally, the `State` key is marked "Invalid" because not all of the information is known to be valid. The other key data, including `Msg` and `TS` keys, are not touched. The values of the `ResourceInfo` attribute can be manually cleared using the `hares -flushinfo` command. This command deletes any optional keys for the `ResourceInfo` attribute and sets the three mandatory keys to their default values.

See the `hares` manual page.

RestartLimit

Affects how the agent responds to a resource fault (see "FaultOnMonitorTimeouts" on page 101 and "ToleranceLimit" on page 109). A non-zero `RestartLimit` causes VCS to invoke the `online` entry point instead of failing over the service group to another system. VCS attempts to restart the resource according to the number set in RestartLimit before it gives up and fails over. However, if the resource remains online for the interval designated in ConfInterval, earlier attempts to restart are not counted against RestartLimit. Default is 0. The `RestartLimit` attribute value can be overridden.

Note The agent will not restart a faulted resource if the `clean` entry point is not implemented. Therefore, the value of the `RestartLimit` attribute applies only if `clean` is implemented.

RegList

RegList is a type-level attribute of the type keylist that can be used to store, or register, a list of certain resource level attributes. The agent calls the attr_changed entry point for a resource when the value of an attribute listed in RegList is modified. The RegList attribute is useful where a change in the values of important attributes require specific actions that can be executed from the attr_changed entry point.

By default, the attribute RegList is not included in a resource's type definition, but it can be added using either of the two methods shown below.

Assume the RegList attribute is added to the FileOnOff resource type definition and its value is defined as PathName. Thereafter, when the value of the PathName attribute for a FileOnOff resource is modified, the attr_changed entry point is called.

◆ Method one is to modify the types definition file (types.cf, for example) to include the RegList attribute. Add a line in the definition of a resource type that resembles:

```
static keylist RegList = { attribute1_name, attribute2_name, ...}
```

For example, if the type definition is for the FileOnOff resource and the name of the attribute to register is PathName, the modified type definition would resemble:

```
.
.
.
type FileOnOff (
    str PathName
    static keylist RegList = { PathName }
    static str ArgList[] = { PathName }
)
.
.
```

◆ Method two is to use the haattr command to add the RegList attribute to a resource type definition and then modify the value of the type's RegList attribute using the hatype command; the commands are:

```
haattr -add -static resource_type RegList -keylist
hatype -modify resource_type RegList attribute_name
```

For example:

```
# haattr -add -static FileOnOff RegList -keylist
# hatype -modify FileOnOff RegList PathName
```

The RegList attribute cannot be overridden.

SupportedActions

The SupportedActions (string-keylist) attribute lists all possible actions defined for an agent, including those defined by the agent developer. The engine validates the *action_token* value specified in the `hares -action` *`resource action_token`* command against the SupportedActions attribute. It is the responsibility of the agent developer to initialize the SupportedActions attribute in the resource type definition and update the definition for each new action added to the `action` entry point code or script. See "action" on page 25. This attribute serves as a reference for users of the command line or the graphical user interface.

An example definition of a resource type may resemble:

```
Type DBResource (
  static str ArgList[] = { Sid, Owner, Home, User, Pwork, StartOpt,
    ShutOpt }
  static keylist SupportedActions = { VRTS_GetRunningServices,
     DBRestrict, DBUndoRestrict, DBSuspend, DBResume }
  str Sid
  str Owner
  str Home
  str User
  str Pword
  str StartOpt
  str ShutOpt
```

In the `SupportedActions` attribute definition above, `VRTS_GetRunningServices` is a VERITAS predefined action, and the actions following it are defined by the developer. The `SupportedActions` attribute value cannot be overridden.

ToleranceLimit

A non-zero ToleranceLimit allows the `monitor` entry point to return OFFLINE several times before the ONLINE resource is declared FAULTED. If the `monitor` entry point reports OFFLINE more times than the number set in ToleranceLimit, the resource is declared FAULTED. However, if the resource remains online for the interval designated in ConfInterval, any earlier reports of OFFLINE are not counted against ToleranceLimit. Default is 0. The `ToleranceLimit` attribute value can be overridden.

Scheduling Class and Priority Configuration Support

VCS allows you to specify priorities and scheduling classes for VCS processes. VCS supports the following scheduling classes:

- RealTime (specified as "RT" in the configuration file)
- TimeSharing (specified as "TS" in the configuration file)
- SRM scheduler (Solaris only) (specified as "SHR" in the configuration file)

Priority Ranges

The following table displays the platform-specific priority range for RealTime, TimeSharing, and SRM scheduling (SHR) processes.

Platform	Scheduling Class	Default Priority Range Weak / Strong	Priority Range Using #ps Commands
AIX	RT TS	126 / 52 60	126 / 52 Priority varies with CPU consumption. **Note** On AIX, use **`#ps -ael`**
HP-UX	RT TS	127 / 0 N/A	127 / 0 N/A **Note** On HP-UX, use **`#ps -ael`**
Solaris	RT TS SHR	0 / 59 -60 / 60 -60 / 60	100 /159 N/A N/A **Note** On Solaris, use **`#ps -ae -o pri, args`**

Default Scheduling Classes and Priorities

The following table lists the default class and priority values used by VCS. Note that the default priority value is platform-specific. Therefore, when priority is set to 0, VCS converts the priority to a value specific to the platform on which the system is running. For TS, the default priority equals the strongest priority supported by the TimeSharing class. For RT, the default priority equals two less than the strongest priority supported by the RealTime class. So, if the strongest priority supported by the RealTime class is 59, the default priority for the RT class is 57. For SHR (on Solaris only), the default priority is the strongest priority support by the SHR class.

Process	**Default Scheduling Class**	**Default Priority**		
		AIX	**HP-UX**	**Solaris**
Engine	RT	52 (Strongest + 2)	2 (Strongest + 2)	57 (Strongest - 2)
Process created by Engine	TS	60	N/A	60 (Strongest)
Agent	TS	60	N/A	0
Script	TS	60	N/A	0

Attributes for Scheduling Class and Priorities

The following attributes are used to set the class and the priority for VCS agent and script processes.

AgentClass

Indicates the scheduling class for the VCS agent process. “TS” is default.

AgentPriority

Indicates the priority in which the agent process runs. 0 is default.

ScriptClass

Indicates the scheduling class of the script processes (for example, online) created by the agent. “TS” is default.

ScriptPriority

Indicates the priority of the script processes created by the agent. 0 is default.

Initializing Attributes in the Configuration File

The following configuration shows how to initialize these attributes through configuration files. The example shows attributes of a FileOnOff resource.

```
type FileOnOff (
  static str AgentClass = RT
  static str AgentPriority = 10
  static str ScriptClass = RT
  static str ScriptPriority = 40
  static str ArgList[] = { PathName }
  str PathName
)
```

Setting Attributes Dynamically from the Command Line

▼ To update the AgentClass

Type:

hatype -modify *resource_type* **AgentClass** *value*

For example, to set the AgentClass attribute of the FileOnOff resource to Realtime, type:

hatype -modify FileOnOff AgentClass "RT"

▼ To update the AgentPriority

Type:

hatype -modify *resource_type* **AgentPriority** *value*

For example, to set the AgentPriority attribute of the FileOnOff resource to 10, type:

hatype -modify FileOnOff AgentPriority "10"

▼ To update the ScriptClass

Type:

hatype -modify *resource_type* **ScriptClass** *value*

For example, to set the ScriptClass of the FileOnOff resource to RealTime, type:

hatype -modify FileOnOff ScriptClass "RT"

▼ To update the ScriptPriority

Type:

hatype -modify *resource_type* **ScriptPriority** *value*

For example, to set the ScriptClass of the FileOnOff resource to RealTime, type:

```
hatype -modify FileOnOff ScriptPriority "40"
```

Note: For the attributes AgentClass and AgentPriority, changes are effective immediately. For ScriptClass and ScriptPriority, changes become effective for scripts issued after the execution of the `hatype` command.

Testing VCS Agents 8

VCS agents can be tested using two methods:

- The VCS engine (see "Using the VCS Engine Process" on page 116)
- The `AgentServer` utility (see "Using the AgentServer Utility" on page 117)

Before testing an agent, make sure you have completed the following tasks:

✔ Built the agent binary and put it in the directory `$VCS_HOME/bin/`*`resource_type`*.

✔ Installed script entry points in the directory `$VCS_HOME/bin/`*`resource_type`*.

And, if you are using the VCS engine process to test the agent, make sure you have:

✔ Defined the resource type in `types.cf`, defined the resources in `main.cf`, and restarted the engine. You may define the new type and resources using the commands listed in the *VERITAS Cluster Server User's Guide*.

Using Debug Messages

You can activate C++ agent debug messages by setting the value of the `LogDbg` attribute of the resource type to `DBG_AGINFO`. This directs the framework to print messages logged with agent debug severity of `DBG_AGINFO`. Debug messages are logged to a specific file:

$VCS_LOG/log/*resource_type*_A.log

Use the `halog` command with the `-addtags` option to set up debug tags for use by script agents. Messages from `halog` are logged to the VCS engine log.

Using the VCS Engine Process

When the VCS engine process "had" becomes active on a system, it automatically starts the appropriate agent processes, based on the contents of the configuration files. A single VCS agent process monitors all resources of the same type on a system.

After the VCS engine process is active, type the following command at the system prompt to verify that the agent has been started and is running:

```
haagent -display resource_type
```

For example, to test the Oracle agent, type:

```
haagent -display Oracle
```

If the Oracle agent is running, the output resembles:

```
#Agent  Attribute   Value
Oracle  AgentFile
Oracle  Faults     0
Oracle  Running     Yes
Oracle  Started     Yes
```

Test Commands

The following examples show how VCS commands can be used to test the agent:

- To activate agent debug messages for C++ agents, type:

  ```
  hatype -modify resource_type LogDbg -add DBG-AGINFO
  ```

- To check the status of a resource, type:

  ```
  hares -display resource_name
  ```

- To bring a resource online, type:

  ```
  hares -online resource_name -sys system
  ```

  This causes the `online` entry point of the corresponding agent to be called.

- To take a resource offline, type:

  ```
  hares -offline resource_name -sys system
  ```

  This causes the `offline` entry point of the corresponding agent to be called.

- To deactivate agent debug messages for C++ agents, type:

  ```
  hatype -modify resource_type LogDbg -delete DBG-AGINFO
  ```

Using the AgentServer Utility

The `AgentServer` utility enables you to test agents without running the VCS engine process.The utility is part of the VCS package and is installed in the directory `$VCS_HOME/bin`. Run the `AgentServer` utility when the VCS engine is not running.

▼ To start the AgentServer and access help

1. Type the following command to start `AgentServer`:

   ```
   # $VCS_HOME/bin/AgentServer
   ```

   The `AgentServer` utility monitors a TCP port for messages from the VCS agents. This port number can be configured by setting `vcstest` to the selected port number in the file `/etc/services`. If `vcstest` is not specified, `AgentServer` uses the default value 14142.

2. When `AgentServer` is started, a message prompts you to enter a command or to type `help` for a complete list of the `AgentServer` commands. We recommend you type `help` to review the commands before getting started.

   ```
   > help
   ```

   Output resembles:

   ```
   The following commands are supported.(Use help for more
   information on using any command.)

   addattr
   addres
   addstaticattr
   addtype
   debughash
   debugmemory
   debugtime
   delete
   deleteres
   modifyres
   modifytype
   offlineres
   onlineres
   print
   proberes
   quit
   startagent
   stopagent
   ```

3. For help on a specific command, type help *command_name* at the `AgentServer` prompt (>). For example, for information on how to bring a resource online, type:

```
>help onlineres
```

The output resembles:

```
Sends a message to an agent to online a resource.
Usage: onlineres <agentid> <resname>
where <agentid> is id for the agent - usually same as
the resource type name.
where <resname> is the name of the resource.
```

▼ **To test the FileOnOff agent**

1. Start the agent for the resource type:

```
>startagent FileOnOff /opt/VRTSvcs/bin/FileOnOff/FileOnOffAgent
```

You receive the following messages:

```
Agent (FileOnOff) has connected.
```

2. `Agent (FileOnOff) is ready to accept commands`. The following are examples of `type.cf` and `main.cf` configuration files that can be referred to when testing the FileOnOff agent:

   - Example `types.cf` definition for the FileOnOff agent:

```
type FileOnOff (
    str PathName
    static str ArgList[] = { PathName }
)
```

   - Example `main.cf` definition for a FileOnOff resource:

```
...
group ga (
...
)        FileOnOff file1 (
  Enabled = 1
  PathName = "/tmp/VRTSvcsfile001"
)
```

In step 3, the sample configuration is set up using AgentServer commands.

3. Complete step a through step f to pass this sample configuration to the VCS agent.

a. Add a type:

```
>addtype FileOnOff FileOnOff
```

b. Add attributes of the type:

```
>addattr FileOnOff FileOnOff PathName str ""
>addattr FileOnOff FileOnOff Enabled int 0
```

c. Add the static attributes to the FileOnOff resource type:

```
>addstaticattr FileOnOff FileOnOff ArgList
   vector PathName
```

d. Add the LogLevel attribute to see the debug messages from the VCS agent:

```
>addstaticattr FileOnOff FileOnOff LogLevel str info
```

e. Add a resource:

```
>addres FileOnOff file1 FileOnOff
```

f. Set the resource attributes:

```
>modifyres FileOnOff file1 PathName str
  /tmp/VRTSvcsfile001
>modifyres FileOnOff file1 Enabled int 1
```

4. After adding and modifying resources, type the following command to obtain the status of a resource:

```
>proberes FileOnOff file1
```

This calls the `monitor` entry point of the FileOnOff agent.

You will receive the following messages indicating the resource status:

```
Resource(file1) is OFFLINE
Resource(file1) confidence level is 0
```

a. To bring a resource online:

```
>onlineres FileOnOff file1
```

This calls the `online` entry point of the FileOnOff agent. The following message is displayed when `file1` is brought online:

```
Resource(file1) is ONLINE
Resource(file1) confidence level is 100
```

b. To take a resource offline:

```
>offlineres FileOnOff file1
```

This calls the `offline` entry point of the FileOnOff agent. A status message similar to the example in step a is displayed when `file1` is taken offline.

5. View the list of VCS agents started by the `AgentServer` process:

```
>print
```

Output resembles:

```
Following Agents are started:
FileOnOff
```

6. Stop the agent:

```
>stopagent FileOnOff
```

7. Exit from the `AgentServer`:

```
>quit
```

State Transition Diagrams 9

This chapter illustrates the state transitions within the agent framework. State transition diagrams are shown separately for the following behaviors:

- Opening a resource
- Resource in a steady state
- Onlining a resource
- Offlining a resource
- Resource fault (without automatic restart)
- Resource fault (with automatic restart)
- Monitoring of persistent resources
- Closing a resource

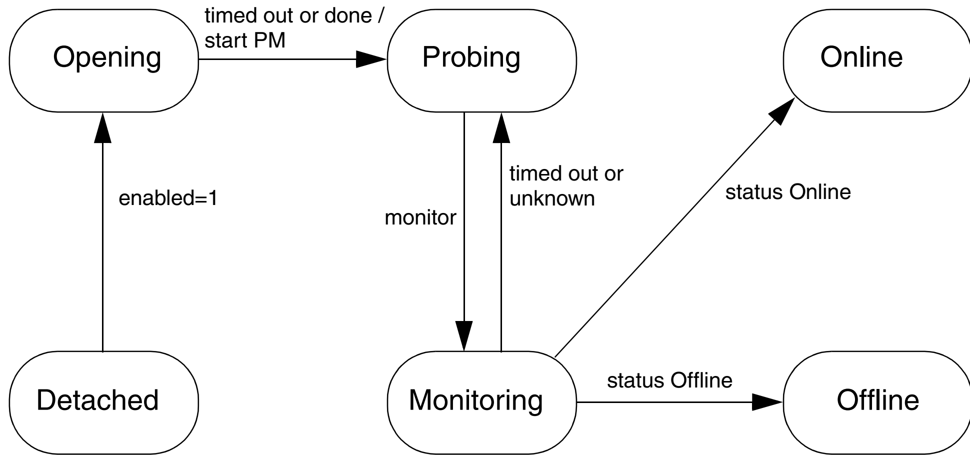
In addition, state transitions are shown for the handling of resources with respect to the `ManageFaults` service group attribute. By default, `ManageFaults` is set to NONE, in which case the clean entry point is not called by VCS. See "ManageFaults" on page 103.The diagrams cover the following conditions:

- Onlining a resource when the `ManageFaults` attribute is set to NONE
- Offlining a resource when the `ManageFaults` attribute is set to NONE
- Resource fault when `ManageFaults` attribute is set to ALL
- Resource fault (unexpected `offline`) when `ManageFaults` attribute is set to NONE
- Resource fault (`monitor` is hung) when `ManageFaults` attribute is set to ALL
- Resource fault (`monitor` is hung) when `ManageFaults` attribute is set to NONE

The states shown in these diagrams are associated with each resource by the agent framework. These states are used only within the agent framework and are independent of the IState resource attribute values indicated by the engine.

The VCS agent writes resource state transition information into the agent log file when the `LogDbg` parameter, a static resource type attribute, is set to the value `DBG_AGINFO`. Agent developers can make use of this information when debugging agents.

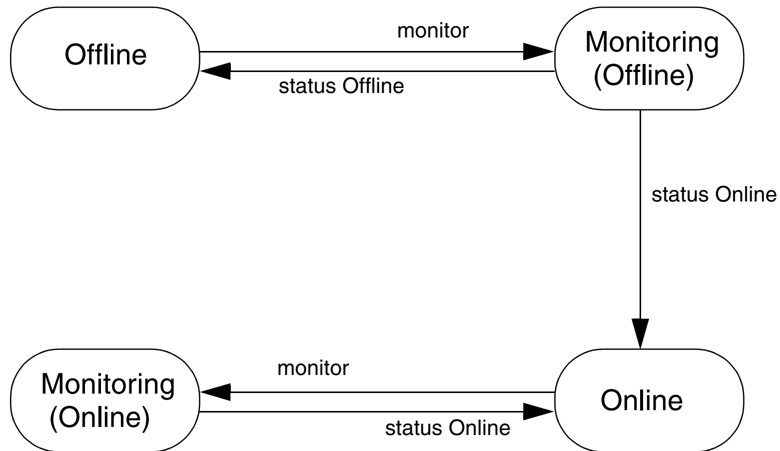
Opening a resource



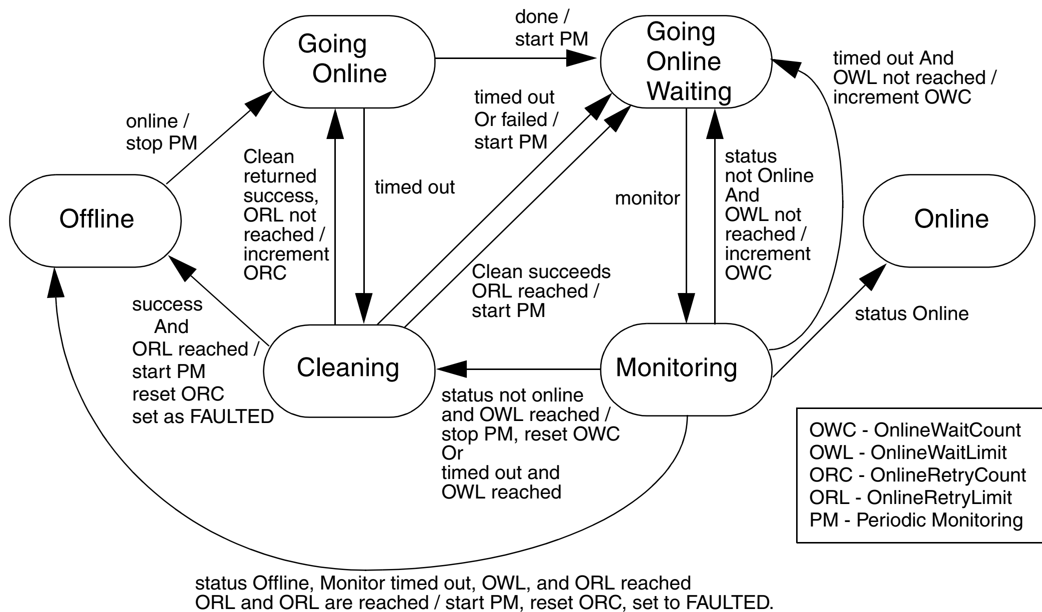
When the agent starts up, each resource starts with the initial state of Detached. In the Detached state (Enabled=0), the agent rejects all commands to online or offline the resource.

When the resource is enabled (Enabled=1), the `open` entry point is invoked. Periodic Monitoring begins after *`open`* times out or succeeds. Depending on the return value of *`monitor`*, the resource transitions to either the Online or the Offline state. In the unlikely event that *`monitor`* times out or returns unknown, the resource stays in a Probing state.

Resource in steady state



When resources are in a steady state of Online or Offline, they are monitored at regular intervals. The intervals are specified by the `MonitorInterval` parameter for a resource in the Online state and the `OfflineMonitorInterval` parameter for a resource in the Offline state. An Online resource that is unexpectedly detected as Offline is considered to be faulted. Refer to diagrams describing faulted resources.



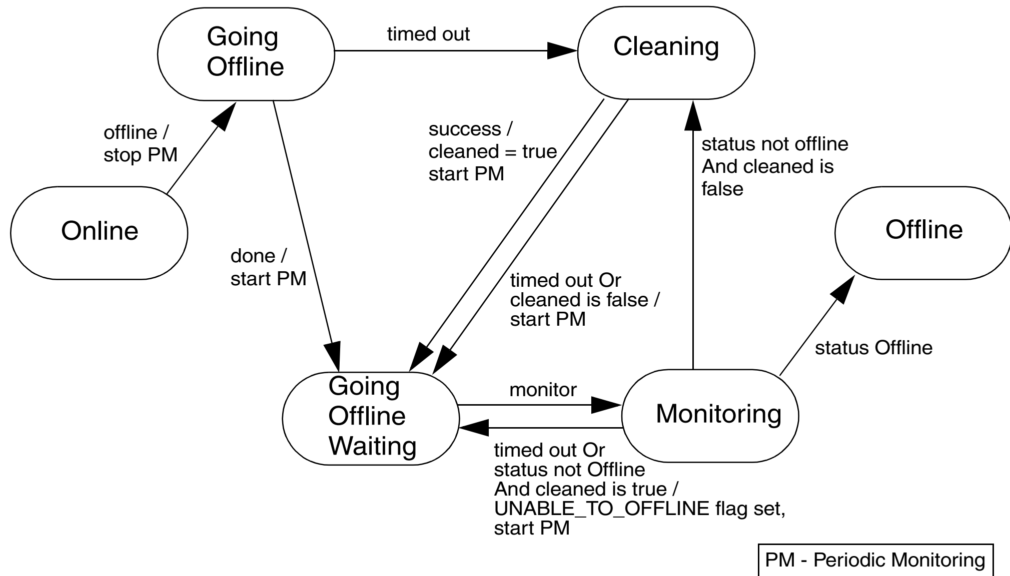
When the agent receives a request from the VCS engine to online the resource, the resource enters the Going Online state, where the `online` entry point is invoked. If `online` completes successfully, the resource enters the Going Online Waiting state where it waits for the next monitor cycle.

If `monitor` returns a status of online, the resource moves to the Online state.

If, however, the `monitor` times out, or returns a status of "not Online" (that is, unknown or offline), the agent returns the resource to the Going Online Waiting state and waits for the next monitor cycle.

When the `OnlineWaitLimit` is reached, the `clean` entry point is invoked.

- If `clean` times out or fails, the resource again returns to the Going Online Waiting state and waits for the next monitor cycle. The agent again invokes the `clean` entry point if the monitor reports a status of "not Online."
- If `clean` succeeds with the `OnlineRetryLimit` reached, and the subsequent monitor reports offline status, the resource transitions to the offline state and is marked FAULTED.
- If `clean` succeeds and the `ORL` is not reached, the resource transitions to the Going Online state where the `online` entry point is retried.



Upon receiving a request from the VCS engine to offline a resource, the agent places the resource in a Going Offline state and invokes the `offline` entry point.

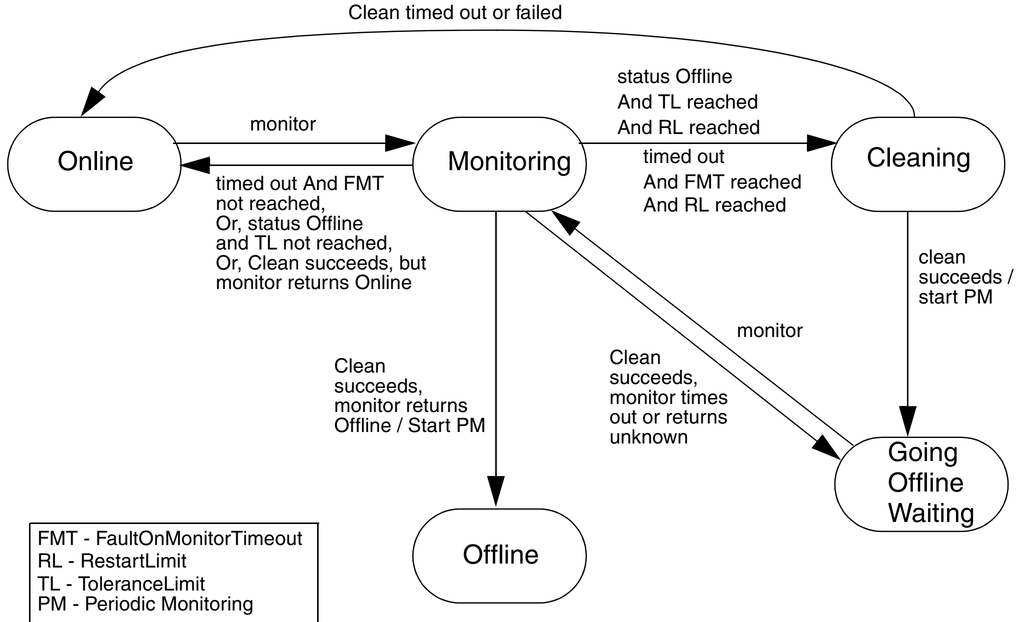
If `offline` succeeds, the resource enters the Going Offline Waiting state where it waits for the next monitor.

If `monitor` returns a status of offline, the resource is marked Offline.

If the monitor times out or return a status "not offline," the agent invokes the `clean` entry point. Also, if, in the Going Offline state, the `offline` entry point times out, the agent invokes `clean` entry point.

- If `clean` fails or times out, the resource is placed in the Going Offline Waiting state and monitored. If `monitor` reports "not offline," the agent invokes the `clean` entry point, where the sequence of events repeats.
- If `clean` returns success, the resource is placed in the Going Offline Waiting state and monitored. If `monitor` times out or reports "not offline," the resource returns to the GoingOfflineWaiting state. The UNABLE _TO_OFFLINE flag is sent to engine.

Resource fault without automatic restart



This diagram describes the activity that occurs when a resource faults and the `RestartLimit` is reached. When the *`monitor`* entry point times out successively and `FaultOnMonitorTimeout` is reached, or monitor returns offline and the `ToleranceLimit` is reached, the agent invokes the *`clean`* entry point.

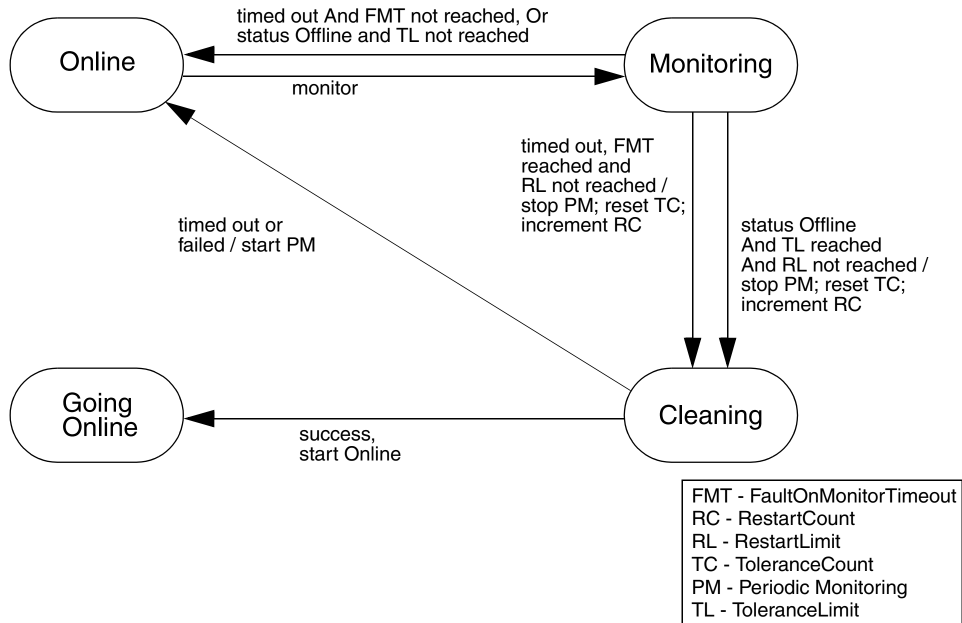
Note For VCS on Windows, the `FaultOnMonitorTimeout` attribute has no significance. Instead, the monitor entry point is allowed to continue to completion.

If *`clean`* fails, or if it times out, the agent places the resource in the Online state as if no fault occurred.

If *`clean`* succeeds, the resource is placed in the Going Offline Waiting state, where the agent waits for the next monitor.

- If *`monitor`* reports Online, the resource is placed back Online as if no fault occurred. If *`monitor`* reports Offline, the resource is placed in an Offline state and marked as FAULTED.
- If *`monitor`* reports unknown or times out, the agent places the resource back into the Going Offline Waiting state, and sets the UNABLE_TO_OFFLINE flag in the engine.

Resource fault with automatic restart

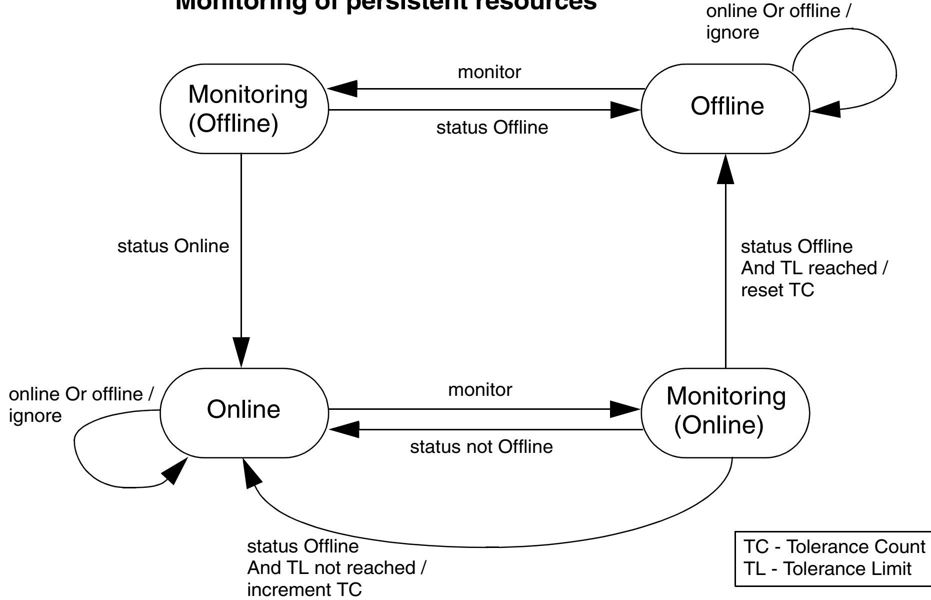


This diagram describes the activity that occurs when a resource faults and the `RestartLimit` is not reached. When the *`monitor`* entry point times out successively and `FaultOnMonitorTimeout` is reached, or `monitor` returns offline and the `ToleranceLimit` is reached, the agent invokes the *`clean`* entry point.

Note For VCS on Windows, the `FaultOnMonitorTimeout` attribute has no significance. Instead, the monitor entry point is allowed to continue to completion.

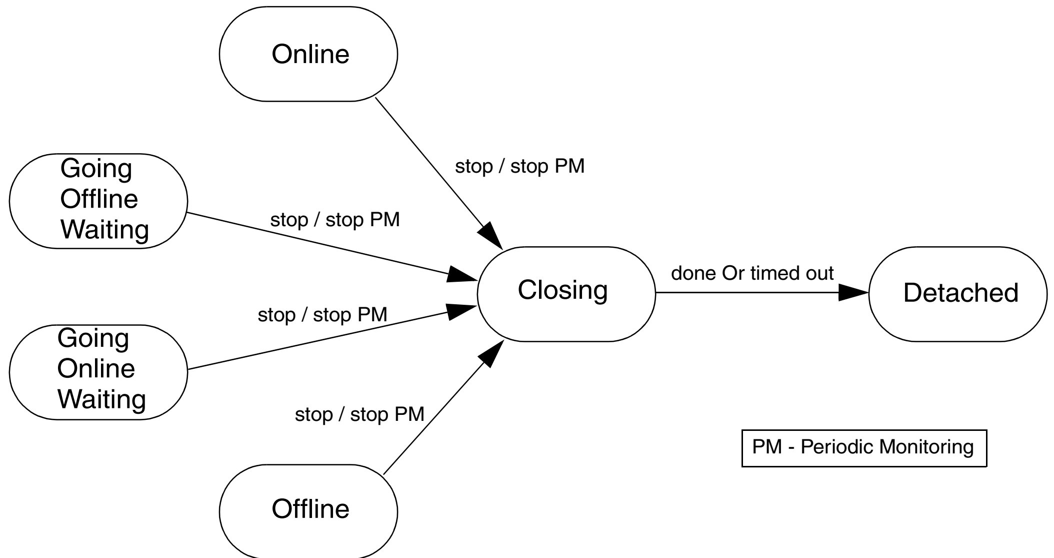
- If *`clean`* succeeds, the resource is placed in the Going Online state and the *`online`* entry point is invoked to restart the resource; refer to the diagram, "Onlining a resource."
- If *`clean`* fails or times out, the agent places the resource in the Online state as if no fault occurred.

Refer to the diagram "Resource fault without automatic restart," for a discussion of activity when a resource faults and the `RestartLimit` is reached.



For a persistent resource in the Online state, if *monitor* returns a status of offline and the `ToleranceLimit` is not reached, the resource stays in an Online state. If *monitor* returns offline and the `ToleranceLimit` is reached, the resource is placed in an Offline state and noted as FAULTED. If *monitor* returns "not offline," the resource stays in an Online state.

Likewise, for a persistent resource in an Offline state, if *monitor* returns offline, the resource remains in an Offline state. If *monitor* returns a status of online, the resource is placed in an Online state.



When the resource is disabled (Enabled=0), the agent stops Periodic Monitoring and the the `close` entry point is invoked. When the `close` entry point succeeds or times out, the resource is placed in the Detached state.

Onlining a resource: ManageFaults attribute = NONE

Offline

online / stop PM

Going Online

timed out / set ADMIN-WAIT; start PM clean reason = ONH

done / insert monitor; start PM

ONH - Online Hang
ONI - Online Ineffective
ORC - Online Retry Count
ORL - Online Retry Limit
OWC - Online Wait Count
OWL - Online Wait Limit
PM - Periodic Monitoring

monitor

Monitoring

status = online

Online

status = offline or unknown or monitor timed out

MSG_RES_ CLRADMINWAIT_ FAULT / reset ADMIN_WAIT; ORC = ORL

Going Online Waiting

NO / start PM

Is OWL reached ?

YES / reset OWC

MSG_RES_CLRADMINWAIT or MSG_RES_RESET_MF / reset ADMIN_WAIT; ManageFaults = 1

Is ORL reached ?

NO / set ADMIN-WAIT; clean reason = ONI

YES

Is status= Offline ?

NO (status unknown or monitor timed out) / start PM

YES / start PM

Offline

Offlining a resource; ManageFaults attribute = NONE

Online

offline /
stop PM

Going
Offline

done /
insert monitor;
start PM

timed out /
set ADMIN_WAIT;
clean reason = OFH

monitor

Monitoring

status = offline

Offline

status =
online or
unknown or
monitor
timed out

MSG_RES_CLRADMIN_WAIT
or MSG_RES_RESET_MF /
clear ADMIN_WAIT;
insert monitor

NO / set
ADMIN_WAIT;
clean reason =
OFI

Going Offline
Waiting

Is
Offline
cleaned
?

MSG_RES_CLRADMINWAIT_FAULT /
clear ADMIN_WAIT;
offline_cleaned = 1;
insert monitor

YES / set
UNABLE_TO_OFFLINE

OFH - Offline Hang
OFI - Offline Ineffective
PM - Periodic Monitoring

Resource fault: ManageFaults attribute = ALL

Online

monitor

Monitoring

Status = offline
start PM

Status = online /
start PM

Status = offline

Offline

NO / increment
tolerance count

Is TL
reached
?

YES / stop PM; faulted = 1
call clean;
increment clean count

monitor

status unknown
or monitor timed out

Cleaning

clean timed out or failed /
Start PM

clean not
implemented
or persistent
resource /
offline_cleaned = 0
reset clean_count

Clean success /
offline_cleaned = 1;
increment restart count;
reset tolerance count

Going Offline
Waiting

Is RL
reached
?

YES

NO /
call online;
set FSTATE_RESTARTING

Going Online

See **Onlining a resource with ManageFaults = ALL**

PM - Periodic Monitoring
RL - Restart Limit
TL - Tolerance Limit

Resource fault (unexpected offline): ManageFaults attribute = NONE

YES / clean reason = 00;
set ADMIN_WAIT

MSG_RES_
CLEARADMIN_WAIT or
MSG_RES_RESET_MF;
reset ADMIN_WAIT;
insert monitor

Is TL
reached
?

NO

Online

monitor

Monitoring

status = offline /
start PM

status = online or
status = unknown /
MONITOR_UNKNOWN
set

Offline

monitor timed out /
MONITOR_TIMEDOUT set

status = online /
reset faulted; reset clean count;
reset offline-cleaned;
start PM

monitor

status = unknown or
monitor timed out

Going Offline
Waiting

MSG_RES_CLEARADMINWAIT_FAULT /
reset ADMIN_WAIT; faulted = 1;
insert monitor

PM - Periodic Monitoring
TL - Tolerance Limit

Resource fault (monitor hung): ManageFaults attribute = ALL

status = offline /
start PM

Offline

Online

monitor

Monitoring

monitor

Going Offline
Waiting

status = online /
reset consecutive
monitor timeout count

status = unknown or
monitor timed out / set
UNABLE_TO_OFFLINE

status = unknown /
set MONITOR_UNKNOWN

monitor timed
out / set
MONITOR_TIMEDOUT

status = online
reset faulted;
reset clean count;
reset consecutive monitor
timeout count

Is
FOMT
reached
?

NO /
increment consecutive
monitor timeout count

YES /
stop PM;
increment clean count;
call clean (MH);
faulted = 1

YES /
insert monitor /
start PM

clean success /
increment consecutive
monitor timeout count

Cleaning

Is RL
reached
?

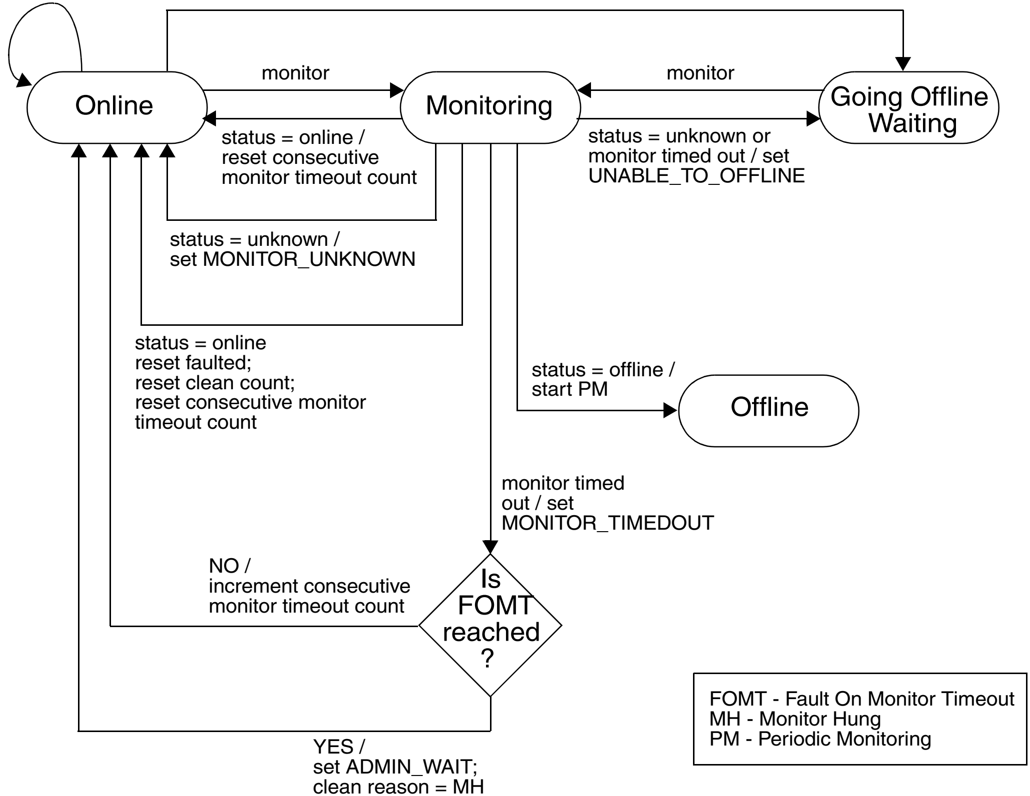
clean failed or
timed out /
start PM

NO / call online;
set FSTATE_RESTARTING

FOMT - Fault On Monitor Timeout
MH - Monitor Hung
PM - Periodic Monitoring
RL - Restart Limit

Going
Offline

Resource fault (monitor hung): ManageFaults attribute = NONE



VCS Internationalized Message Catalogs 10

VCS handles internationalized messages in *binary message catalogs* (BMCs) generated from *source message catalogs* (SMCs).

- A source message catalog (SMC) is a plain text catalog file encoded in ASCII or in UCS-2, a two-byte encoding of Unicode. Developers can create messages using a prescribed format and store them in an SMC.
- A binary message catalog (BMC) is a catalog file in a form that VCS can use. BMCs are generated from SMCs through the use of the `bmcgen` utility.

Each VCS module requires a BMC. For example, the VCS engine (HAD), GAB, and LLT require distinct BMCs, as do each enterprise agent and each custom agent. For agents, a BMC is required for each operating system platform.

Once generated, BMCs must be placed in specific directories that correspond to the module and the language of the message. You can run the `bmcmap` utility within the specific directory to create a BMC *map* file, an ASCII text file that links BMC files with their corresponding module, language, and range of message IDs. The map file enables VCS to manage the BMC files.

You can change an existing SMC file to generate an updated BMC file.

Creating SMC Files

Since Source Message Catalog files are used to generate the Binary Message Catalog files, they must be created in a consistent format.

SMC Format

```
#!language = language_ID
#!module   = module_name
#!version  = version
#!category = category_ID

# comment
message_ID1 {%s:msg}
message_ID2 {%s:msg}
message_ID3 {%s:msg}

# comment
message_ID4 {%s:msg}
message_ID5 {%s:msg}
...
```

Example SMC File

Examine an example SMC file, named `VRTSvcsSunAgent.smc`, based on the SMC format:

```
#!language = en
#!module   = HAD
#!version  = 4.0
#!category = 203

# common library

100 {"%s:Invalid message for agent"}
101 {"%s:Process %s restarted"}
102 {"%s:Error opening /proc directory"}
103 {"%s:online:No start program defined"}
104 {"%s:Executed %s""}
105 {"%s:Executed %s"}
```

Formatting SMC Files

- SMC files must be encoded in UCS-2, ASCII, or UTF-8. See "Naming SMC Files, BMC Files" on page 139 for a discussion of file naming conventions.
- All messages should begin with "%s:" that represents the three-part header "Agent:Resource:EntryPoint" generated by the agent framework.
- The HAD module must be specified in the header for custom agents. See "Example SMC File."
- The minor number of the version (for example, 2.x) can be modified each time a BMC is to be updated. The major number is only to be changed by VCS. The version number indicates to processes handling the messages which catalog is to be used. See "Updating BMC Files."
- In the SMC header, no space is permitted between the "#" and the "!" characters. Spaces can follow the "#" character and regular comments in the file. See the example above.
- SMC filenames must use the extension: `.smc`.
- A message should contain no more than six string format specifiers.
- Message IDs must contain only numeric characters, not alphabetic characters. For example, 2001003A is invalid. Message IDs can range from 1 to 65535.
- Message IDs within an SMC file must be in ascending order.
- A message formatted to span across multiple lines must use the "\n" characters to break the line, not a hard carriage return. Line wrapping is permitted. See the examples that follow.

Naming SMC Files, BMC Files

BMC files, which follow a naming convention, are generated from SMC files. The name of an SMC file determines the name of the generated BMC file. The naming convention for BMC files has the following pattern:

VRTSvcs{Sun | AIX | HP | Lnx | W2K}{Agent_name}.bmc

where the *platform* and *agent_name* are included.

For example:

```
VRTSvcsHPOracle.bmc
```

The name of the SMC file used to generate the BMC file for the preceding example is `VRTSvcsHPOracle.smc`.

Message Examples

- An illegal message, with hard carriage returns embedded with the message:

```
201 {"%s:To be or not to be!
  That is the question"}
```

- A valid message using "\n":

```
10010 {"%s:To be or not to be!\n
That is the question"}
```

- A valid message with text wrapping to the next line:

```
10012 {"%s:To be or not to be!\n
That is the question.\n Whether tis nobler in the mind to
suffer\n the slings and arrows of outrageous fortune\n or to
take arms against a sea of troubles"}
```

Using Format Specifiers

Using the "%s" specifier is appropriate for all message arguments unless the arguments must be reordered. Since the word order in messages may vary by language, a format specifier, %#$s, enables the reordering of arguments in a message; the "#" character is a number from 1 to 99.

In an English SMC file, the entry might resemble:

```
301 {"%s:Setting cookie for proc=%s, PID = %s"}
```

In a language where the position of message arguments need to switch, the same entry in the SMC file for that language might resemble:

```
301 {"%s:Setting cookie for process with PID = %3$s, name = %2$s"}
```

Converting SMC Files to BMC Files

Use the `bmcgen` utility to convert SMC files to BMC files. For example:

```
# bmcgen VRTSvcsSunAgent.smc
```

The file `VRTSvcsSunAgent.bmc` is created in the directory where the SMC file exists. A BMC file must have an extension: `.bmc`.

By default, the `bmcgen` utility assumes the SMC file is a Unicode (UCS-2) file. For ASCII or UTF-8 encoded files, use the `-ascii` option. For example:

```
# bmcgen -ascii VRTSvcsSunAgent.smc
```

Storing BMC Files

By default, BMC files must be installed in `/opt/VRTSvcs/messages/`*`language`*, where *language* is a directory containing the BMCs of a given supported language. For example, the path to the BMC for a Japanese agent on a Solaris system resembles:

`/opt/VRTSvcs/messages/ja/VRTSvcsSunAgent.bmc`.

VCS Languages

The languages supported by VCS are listed as subdirectories, such as `/ja` (Japanese) and `/en` (English), in the directory `/opt/VRTSvcs/messages`.

Displaying the Contents of BMC Files

The `bmcread` command enables you to display the contents of the binary message catalog file. For example, the following command displays the contents of the specified BMC file:

```
# bmcread VRTSvcsAIXAgent.bmc
```

Using BMC Map Files

VCS uses a BMC map file to manage the various BMC files of a given module for a given language. HAD is the module for the VCS engine, bundled agents, enterprise agents, and custom agents. A BMC map file is an ASCII text file that associates BMC files with their category and unique message ID range.

Location of BMC Map Files

Map files, by default, are created in the same directories as their corresponding BMC files: /opt/VRTSvcs/messages/*language*.

Creating BMC Map Files

Developers can add BMCs to the BMC map file. After generating a BMC file:

1. Copy the BMC file to the corresponding directory. For example:

```
# cp VRTSvcsSunOracle.bmc /opt/VRTSvcs/messages/en
```

2. Change to the directory containing the BMC file and run the `bmcmap` utility. For example:

```
# cd /opt/VRTSvcs/messages/en
# bmcmap -create en HAD
```

The `bmcmap` utility scans the contents of the directory and dynamically generates the BMC map. In this case, HAD.bmcmap is created.

Example BMC Map File

In the following example, a BMC named `VRTSvcsHPNewCustomAgent.bmc` is included in the BMC map file for the HAD module and the English language.

```
#
# Copyright(C) 2001 VERITAS Software Corporation.
# ALL RIGHTS RESERVED.
#

Language=en

HAD=VRTSvcsHad VRTSvcsHPAgent VRTSvcsHPNewCustomAgent
```

```
# VCS
VRTSvcsHad.version=1.0
VRTSvcsHad.IDstart=0
VRTSvcsHad.IDend=16501
VRTSvcsHad.Category=__________

# HP Bundled Agents
VRTSvcsHPAgent.version=1.0
VRTSvcsHPAgent.IDstart=100001
VRTSvcsHPAgent.IDend=113501
VRTSvcsHPAgent.Category=__________


# HP NewCustomAgent
VRTSvcsHPNewCustomAgent.version=1.0
VRTSvcsHPNewCustomAgent.IDstart=2017001
VRTSvcsHPNewCustomAgent.IDend=2017040
VRTSvcsHPNewCustomAgent.Category=_________
```

Updating BMC Files

You can update an existing BMC file. This may be necessary, for example, to add new messages or to change a message. This can be done in the following way:

1. If the original SMC file for a given BMC file exists, you can edit it using a text editor. Otherwise create a new SMC file.

    a. Make your changes, such as adding, deleting, or changing messages.

    b. Change the minor number of the version number in the header. For example, change the version from 2.0 to 2.1.

    c. Save the file.

2. Generate the new BMC file using the `bmcgen` command; place the new BMC file in the corresponding language directory.

3. In the directory containing the BMC file, run the `bmcmap` command to create a new BMC map file.

How to Use Pre-VCS 4.0 Agents

The agent framework is the component of VCS that supports all VCS agents by enabling them to communicate with the engine about the definitions of resource types, the values configured for the resource attributes, and entry points they use. VCS 4.1 uses the agent framework implemented in VCS 4.0.

Guidelines for Using Pre-VCS 4.0 Agents

Changes made to the agent framework with VCS 4.0 and later releases affect how the agents developed using the 2.0 or 3.5 agent framework can be used. While it is not necessary, it is recommended that all pre-VCS 4.0 agents be modified to work with the VCS 4.0 and later agent framework so that the new entry points can be used.

Note the following guidelines:

- If the pre-VCS 4.0 agent is implemented strictly in scripts, then the VCS ScriptAgent can be used. If desired, the VCS 4.0 and later `action` and `info` entry points can be used directly.
- If the pre-VCS 4.0 agent is implemented using any C++ entry points, the agent can be used if developers *do not* care to implement the `action` or `info` entry points. The VCS 4.0 and later agent framework assumes all pre-VCS 4.0 agents are version 3.5.
- If the pre-VCS 4.0 agent is implemented using any C++ entry points, and developers *want* to implement the `action` or the `info` entry point, the developers must:
  - Add the `action` or `info` entry point, either C++ or script-based, to the agent.
  - Define the entry points in the VCS primitive, `VCSAg40EntryPointStruct`, and use the primitive `VCSAgRegisterEPStruct` to register the agent as a VCS 4.0 agent.
  - Recompile the agent.

Note Agents developed on the 4.0 and later agent framework are not compatible with the 2.0 or the 3.5 frameworks.

Log Messages in Pre-VCS 4.0 Agents

The log messages in pre-VCS 4.0 agents are automatically converted to the VCS 4.0 and later message format. "Logging Agent Messages" on page 69 describes the VCS 4.0 message format.

Mapping of Log Tags (Pre-VCS 4.0) to Log Severities (VCS 4.0)

For agents, the severity levels of entry point messages for VCS 4.0 and later correspond to the pre-VCS 4.0 entry point message tags as shown in this table:

Log Tag (Pre-VCS 4.0)	Log Severity (VCS 4.0 and later)
TAG_A	VCS_CRITICAL
TAG_B	VCS_ERROR
TAG_C	VCS_WARNING
TAG_D	VCS_NOTE
TAG_E	VCS_INFORMATION
TAG_F through TAG_Z	VCS_DBG1 through VCS_DBG21

How Pre-VCS 4.0 Messages are Displayed by VCS 4.0 and Later

In the following examples, a message written in a VCS 3.5 agent is shown as it would appear in VCS 3.5 and as it appears in VCS 4.0 and later. Note that when messages from pre-VCS 4.0 agents are displayed by VCS 4.0 or later, a category ID of 10000 is included in the unique message identifier portion of the message. The category ID was introduced with VCS 4.0.

- Pre-VCS 4.0 message output:

```
TAG_B 2003/12/08 15:42:30
VCS:141549:Mount:nj_batches:monitor:Mount resource will not go
online because FsckOpt is incomplete
```

- Pre-VCS 4.0 message displayed by VCS 4.0 and later

```
2003/12/15 12:39:32 VCS ERROR V-16-10000-141549
Mount:nj_batches:monitor:Mount resource will not go online because
FsckOpt is incomplete
```

Comparing Pre-VCS 4.0 APIs and VCS 4.0 Logging Macros

The topic "Logging Agent Messages" on page 69 describes how to use the VCS 4.0 logging macros for C++ agents and script-based agents.

For the purpose of comparison, the examples that follow show a pair of messages in C++ that are formatted using the pre-VCS 4.0 API and the VCS 4.0 macros.

- Pre-VCS 4.0 APIs:

```
sprintf(msg,
"VCS:140003:FileOnOff:%s:online:The value for PathName attribute
is not specified", res_name);
   VCSAgLogI18NMsg(TAG_C, msg, 140003,
      res_name, NULL, NULL, NULL, LOG_DEFAULT);
   VCSAgLogI18NConsoleMsg(TAG_C, msg, 140003, res_name,
      NULL,NULL,NULL,LOG_DEFAULT);
```

- VCS 4.0 macros:

```
VCSAG_LOG_MSG(VCS_WARNING, 14003, VCS_DEFAULT_FLAGS,
   "The value for PathName attribute is not specified");
VCSAG_CONSOLE_LOG_MSG(VCS_WARNING, 14003, VCS_DEFAULT_FLAGS,
   "The value for PathName attribute is not specified");
```

Pre-VCS 4.0 Message APIs

The message APIs described in this section of the document are maintained to allow VCS 4.0 and later to work with the agents developed on the 2.0 and 3.5 agent framework.

VCSAgLogConsoleMsg

```
void
VCSAgLogConsoleMsg(int tag, const char *message, int flags);
```

This primitive requests that the VCS agent framework write `message` to the agent log file, $VCS_LOG/log/resource_type_A.log. The `message` must not exceed 4096 bytes. A message greater that 4096 bytes is truncated.

`tag` can be any value from TAG_A to TAG_Z. Tags A–E are enabled by default. To enable other tags, use the `halog` command. `flags` can be zero or more of `LOG_NONE`, `LOG_TIMESTAMP` (prints date and time), `LOG_NEWLINE` (prints a new line), and `LOG_TAG` (prints tag). This primitive can be called from any entry point.

For example:

```
#include "VCSAgApi.h"
...

VCSAgLogConsoleMsg(TAG_A, "Getting low on disk space",
            LOG_TAG|LOG_TIMESTAMP);
...
```

VCSAgLogI18NMsg

```
void
VCSAgLogI18NMsg(int tag, const char *msg,
  int msg_id, const char *arg1_string, const char *arg2_string,
  const char *arg3_string, const char *arg4_string, int flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and four string arguments to the agent log file, $VCS_LOG/log/resource_type_A.log. The `message` must not exceed 4096 bytes. A message greater that 4096 bytes is truncated. The size of all argument strings combined must not exceed 4096 bytes. If the argument string total exceeds 4096 bytes, then each argument is allowed an equal portion of 4096 bytes and truncated if it exceeds the allowed portion.

`tag` can be any value from TAG_A to TAG_Z. Tags A through H are enabled by default. To enable other tags, modify the LogTags attribute of the corresponding resource type. `flags` can be zero or more of `LOG_NONE`, `LOG_TIMESTAMP` (prints date and time), `LOG_NEWLINE` (prints a new line), and `LOG_TAG` (prints tag). This primitive can be called from any entry point.

For example:

```
#include "VCSAgApi.h"
...
char buffer[256];
sprintf(buffer, "VCS:2015001:IP:%s:monitor:Device %s address
  %s", res_name, device, address);

VCSAgLogI18NConsoleMsg(TAG_B, buffer, 2015001, res_name, device,
  address, NULL, LOG_TAG|LOG_TIMESTAMP|LOG_NEWLINE);
```

VCSAgLogI18NMsgEx

```
void
VCSAgLogI18NMsgEx(int tag, const char *msg,
  int msg_id, const char *arg1_string, const char *arg2_string,
  const char *arg3_string, const char *arg4_string,
  const char *arg5_string, const char *arg6_string, int flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and six string arguments to the agent log file, $VCS_LOG/log/resource_type_A.log. The `message` must not exceed 4096 bytes. A message greater that 4096 bytes is truncated. The size of all argument strings combined must not exceed 4096 bytes. If the argument string total exceeds 4096 bytes, then each argument is allowed an equal portion of 4096 bytes and truncated if it exceeds the allowed portion.

`tag` can be any value from TAG_A to TAG_Z. Tags A through H are enabled by default. To enable other tags, modify the LogTags attribute of the corresponding resource type. `flags` can be zero or more of `LOG_NONE`, `LOG_TIMESTAMP` (prints date and time), `LOG_NEWLINE` (prints a new line), and `LOG_TAG` (prints tag). This primitive can be called from any entry point.

For example:

```
#include "VCSAgApi.h"
...
char buffer[256];
sprintf(buffer, "VCS:2015004:Oracle:%s:%s:During scan for
  process %s ioctl failed with return code %s, errno = %s",
res_name, ep_name, proc_name, ret_buf, err_buf);

VCSAgLogI18NConsoleMsgEx(TAG_A, buffer, 2015004, res_name,
ep_name, proc_name, ret_buf, err_buf, NULL, flags);
```

VCSAgLogI18NConsoleMsg

```
void
VCSAgLogI18NConsoleMsg(int tag,
  const char *msg, int msg_id, const char *arg1_string,
  const char *arg2_string, const char *arg3_string,
  const char *arg4_string, int flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and four string arguments to the agent log file, $VCS_LOG/log/resource_type_A.log. The `message` must not exceed 4096 bytes. A message greater that 4096 bytes is truncated. The size of all argument strings combined must not exceed 4096 bytes. If the argument string total exceeds 4096 bytes, then each argument is allowed an equal portion of 4096 bytes and truncated if it exceeds the allowed portion.

`tag` can be any value from TAG_A to TAG_Z. Tags A through E are enabled by default. To enable other tags, use the `halog` command. `flags` can be zero or more of `LOG_NONE`, `LOG_TIMESTAMP` (prints date and time), `LOG_NEWLINE` (prints a new line), and `LOG_TAG` (prints tag). This primitive can be called from any entry point.

For example:

```
#include "VCSAgApi.h"
...

char buffer[256];
sprintf(buffer, "VCS:2015002:IP:%s:monitor:Device %s address
  %s", res_name, device, address);

VCSAgLogI18NConsoleMsg(TAG_B, buffer, 2015002, res_name, device,
  address, NULL, LOG_TAG|LOG_TIMESTAMP|LOG_NEWLINE);
```

VCSAgLogI18NConsoleMsgEx

```
void
VCSAgLogI18NConsoleMsgEx(int tag,
  const char *msg, int msg_id, const char *arg1_string,
  const char *arg2_string, const char *arg3_string,
  const char *arg4_string, const char *arg5_string,
  const char *arg6_string, int flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and six string arguments to the agent log file, $VCS_LOG/log/resource_type_A.log. The `message` must not exceed 4096 bytes. A message greater that 4096 bytes is truncated. The size of all argument strings combined must not exceed 4096 bytes. If the argument string total exceeds 4096 bytes, then each argument is allowed an equal portion of 4096 bytes and truncated if it exceeds the allowed portion.

`tag` can be any value from TAG_A to TAG_Z. Tags A through E are enabled by default. To enable other tags, use the `halog` command. `flags` can be zero or more of `LOG_NONE`, `LOG_TIMESTAMP` (prints date and time), `LOG_NEWLINE` (prints a new line), and `LOG_TAG` (prints tag). This primitive can be called from any entry point.

For example:

```
#include "VCSAgApi.h"
...
...
char buffer[256];
sprintf(buffer, "VCS:2015003:Oracle:%s:%s:During scan for
  process %s ioctl failed with return code %s, errno = %s",
res_name, ep_name, proc_name, ret_buf, err_buf);

VCSAgLogI18NConsoleMsgEx(TAG_A, buffer, 2015003, res_name,
ep_name, proc_name, ret_buf, err_buf, NULL, flags);
```

Index

A

action entry point
- C++ syntax 44
- described 25
- resources to implement for 10
- script entry point 64
- supported actions 109

agent framework 35
- described 2
- entry points registered with 17
- library, C++ 15
- logging APIs 71
- multithreaded 31
- state transitions within 121
- working with pre-4.0 agents 145

agent messages
- formatting 69
- normal in VCSAG_LOG_MSG 83

AgentClass parameter 112
AgentFile parameter 98
AgentPriority parameter 112
AgentReplyTimeout parameter 98
AgentStartTimeout parameter 98
ArgList parameter 99
ArgList reference attributes 99
attr_changed entry point 26
- C++ syntax 46
- script syntax 65

AttrChangedTimeout parameter 99

B

binary message catalog (BMC) files
- converting from SMC files 141
- displaying contents 141
- updating 143

bmcgen utility 141
bmcread utility 141

C

category ID for messages 81
classes, scheduling 110
clean entry point 23
- C++ syntax 43
- enum types 23
- script syntax 64

CleanTimeout parameter 99
close entry point 26
- C++ syntax 49
- script syntax 68

CloseTimeout parameter 99
ComputeStats parameter 99
ConfInterval parameter 100

D

debug message severity level 73
debug messages
- C++ entry points 73
- Perl script entry points 84
- Shell script entry points 84

E

entry points
- action 25
- attr_changed 26
- clean 23
- close 26
- definition 2, 15
- info 21
- monitor 20
- offline 22
- online 22
- open 26
- sample structure 17
- shutdown 26

enum types for clean
- VCSAgCleanMonitorHung 23
- VCSAgCleanOfflineHung 23

VCSAgCleanOfflineIneffective 23
VCSAgCleanOnlineHung 23
VCSAgCleanOnlineIneffective 23
VCSAgCleanUnexpectedOffline 23

F

FaultOnMonitorTimeouts parameter 101
FireDrill parameter 101
formatting agent messages 69

H

had 2
high-availability daemon (had) 2

I

info entry point 21
C++ syntax 37
script example 65
InfoTimeout parameter 101
initializing functions with
VCSAG_LOG_INIT 75

L

localization primitives 58
log category 76
LogDbg parameter 102
LogFileSize parameter 103
logging APIs
C++ 71
script entry points 81

M

ManageFaults parameter 103
message text format 69, 71
mnemonic message field 70
monitor entry point 20
C++ syntax 36, 37
script syntax 63
MonitorLevel parameter 104
MonitorStatsParam parameter 104
MonitorTimeout parameter 104

N

NumThreads parameter 105

O

offline entry point 22
C++ syntax 42
script syntax 63
OfflineMonitorInterval parameter 105
OfflineTimeout parameter 105
online entry point 22
C++ syntax 41
script syntax 63
OnlineRetryLimit parameter 105
OnlineTimeout parameter 105
OnlineWaitLimit parameter 106
OnOff resource type 10
OnOnly resource type 10
open entry point 26
C++ syntax 48
script syntax 67
OpenTimeout parameter 106
Operations parameter 106

P

parameters
AgentClass 112
AgentFile 98
AgentPriority 112
AgentReplyTimeout 98
AgentStartTimeout 98
ArgList 99
AttrChangedTimeout 99
CleanTimeout 99
CloseTimeout 99
ComputeStats 99
ConfInterval 100
FaultOnMonitorTimeouts 101
FireDrill 101
InfoTimeout 101
LogDbg 102
LogFileSize 103
ManageFaults 103
MonitorLevel 104
MonitorStatsParam 104
MonitorTimeout 104
NumThreads 105
OfflineMonitorInterval 105
OfflineTimeout 105
OnlineRetryLimit 105
OnlineTimeout 105
OnlineWaitLimit 106
OpenTimeout 106
Operations 106
RegList 108
RestartLimit 107
ScriptClass 112
ScriptPriority 112
ToleranceLimit 109
persistent resource type 10
primitives

definition 51
for localization 58
VCSAgEncodeString 58
VCSAgGetCookie 56
VCSAgLogI18NMsg 149, 151, 152
VCSAgLogI18NMsgEx 150
VCSAgLogMsg 148
VCSAgRegister 54
VCSAgRegisterEPStruct 51
VCSAgSetCookie 52
VCSAgSnprintf 60
VCSAgStrlcat 60
VCSAgStrlcpy 60
VCSAgUnregister 55
priorities, specifying 110

R

RegList parameter 108
resource
closing (state transition diagram) 129
fault (state transition diagram) 126, 127
monitoring (state transition diagram) 128
offlining (state transition diagram) 125
onlining (state transition diagram) 124
OnOff type 10
OnOnly type 10
opening (state transition diagram) 122
persistent type 10
steady state (state transition diagram) 123
types 10
RestartLimit parameter 107

S

scheduling class and priority 110
ScriptAgent 88, 145
ScriptAgent, location 16
script-based logging functions 80
ScriptClass parameter 112
ScriptPriority parameter 112
severity macros 74
severity message field 70
shutdown entry point 26
C++ syntax 50
script syntax 68
source message catalog (SMC) files
converting to BMC files 138
creating 138
state transition diagram
closing a resource 129
monitoring persistent resources 128
offlining a resource 125
onlining a resource 124
opening a resource 122
resource fault with auto restart 127
resource fault, no auto restart 126
resource in steady state 123

T

timestamp message field 70
ToleranceLimit parameter 109

U

UMI (unique message identifier) 70

V

VCSAG_CONSOLE_LOG_MSG logging macro 71
VCSAG_LOG_INIT initializing function 75
VCSAG_LOG_MSG logging macro 71
VCSAG_LOG_MSG script logging function 80
VCSAG_LOGDBG_MSG logging macro 71
VCSAG_LOGDBG_MSG script logging function 80
VCSAG_RES_LOG_MSG logging macro 71
VCSAG_SET_ENVS script logging function 80
VCSAgEncodeString primitive 58
VCSAgGetCookie primitive 56
VCSAgLogI18NMsg primitive 149, 151, 152
VCSAgLogI18NMsgEx primitive 150
VCSAgLogMsg primitive 148
VCSAgRegister primitive 54
VCSAgRegisterEPStruct primitive 51
VCSAgSetCookie primitive 52
VCSAgSnprintf primitive 60
VCSAgStartup entry point, C++ syntax 35
VCSAgStrlcat primitive 60
VCSAgStrlcpy primitive 60
VCSAgUnregister primitive 55