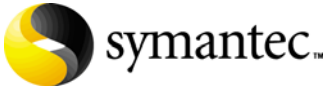
Veritas™ Cluster Server Agent Developer's Guide

AIX, Linux, Solaris

5.1



Veritas Cluster Server Agent Developer's Guide

The software described in this book is furnished under a license agreement and may be used only in accordance with the terms of the agreement.

Product version: 5.1

Document version: 5.1.0

Legal Notice

Symantec Corporation
350 Ellis Street
Mountain View, CA 94043

http://www.symantec.com

Technical Support

Symantec Technical Support maintains support centers globally. Technical Support's primary role is to respond to specific queries about product features and functionality. The Technical Support group also creates content for our online Knowledge Base. The Technical Support group works collaboratively with the other functional areas within Symantec to answer your questions in a timely fashion. For example, the Technical Support group works with Product Engineering and Symantec Security Response to provide alerting services and virus definition updates.

Symantec's maintenance offerings include the following:

- A range of support options that give you the flexibility to select the right amount of service for any size organization
- Telephone and Web-based support that provides rapid response and up-to-the-minute information
- Upgrade assurance that delivers automatic software upgrade protection
- Global support that is available 24 hours a day, 7 days a week
- Advanced features, including Account Management Services

For information about Symantec's Maintenance Programs, you can visit our Web site at the following URL:

www.symantec.com/business/support/index.jsp

Contacting Technical Support

Customers with a current maintenance agreement may access Technical Support information at the following URL:

www.symantec.com/business/support/contact_techsupp_static.jsp

Before contacting Technical Support, make sure you have satisfied the system requirements that are listed in your product documentation. Also, you should be at the computer on which the problem occurred, in case it is necessary to replicate the problem.

When you contact Technical Support, please have the following information available:

- Product release level
- Hardware information
- Available memory, disk space, and NIC information
- Operating system
- Version and patch level
- Network topology
- Router, gateway, and IP address information
- Problem description:
  - Error messages and log files
  - Troubleshooting that was performed before contacting Symantec
  - Recent software configuration changes and network changes

Licensing and registration

If your Symantec product requires registration or a license key, access our non-technical support Web page at the following URL:

customercare.symantec.com

Customer service

Customer Care information is available at the following URL:

customercare.symantec.com

Customer Service is available to assist with the following types of issues:

- Questions regarding product licensing or serialization
- Product registration updates, such as address or name changes
- General product information (features, language availability, local dealers)
- Latest information about product updates and upgrades
- Information about upgrade assurance and maintenance contracts
- Information about the Symantec Buying Programs
- Advice about Symantec's technical support options
- Nontechnical presales questions
- Issues that are related to CD-ROMs or manuals

Documentation feedback

Your feedback on product documentation is important to us. Send suggestions for improvements and reports on errors or omissions. Include the title and document version (located on the second page), and chapter and section titles of the text on which you are reporting. Send feedback to:

sfha_docs@symantec.com

Maintenance agreement resources

If you want to contact Symantec regarding an existing maintenance agreement, please contact the maintenance agreement administration team for your region as follows:

Asia-Pacific and Japan	customercare_apac@symantec.com
Europe, Middle-East, and Africa	semea@symantec.com
North America and Latin America	supportsolutions@symantec.com

Additional enterprise services

Symantec offers a comprehensive set of services that allow you to maximize your investment in Symantec products and to develop your knowledge, expertise, and global insight, which enable you to manage your business risks proactively.

Enterprise services that are available include the following:

Symantec Early Warning Solutions	These solutions provide early warning of cyber attacks, comprehensive threat analysis, and countermeasures to prevent attacks before they occur.
Managed Security Services	These services remove the burden of managing and monitoring security devices and events, ensuring rapid response to real threats.
Consulting Services	Symantec Consulting Services provide on-site technical expertise from Symantec and its trusted partners. Symantec Consulting Services offer a variety of prepackaged and customizable options that include assessment, design, implementation, monitoring, and management capabilities. Each is focused on establishing and maintaining the integrity and availability of your IT resources.
Educational Services	Educational Services provide a full array of technical training, security education, security certification, and awareness communication programs.

To access more information about Enterprise services, please visit our Web site at the following URL:

www.symantec.com

Select your country or language from the site index.

Contents

Chapter 1 Introduction

About VCS agents 16
How agents work 17
About the agent framework 17
Resource type definitions 17
About agent functions (entry points) 18
About on-off, on-only, and persistent resources 18
About attributes 19
About intentional offline of applications 22
About developing an agent 23
Considerations for the application 23
High-level overview of the agent development process 24

Chapter 2 Agent entry point overview

About agent entry points 26
Supported entry points 26
How the agent framework interacts with entry points 26
Agent entry points described 27
About the monitor entry point 27
About the info entry point 28
About the online entry point 30
About the offline entry point 30
About the clean entry point 31
About the action entry point 33
About the attr_changed entry point 34
About the open entry point 34
About the close entry point 35
About the shutdown entry point 35
Return values for entry points 36
Considerations for using C++ or script entry points 37
About the VCSAgStartup routine 37
About the agent information file 39
Example agent information file (UNIX) 39
Implementing the agent XML information file 42
About the ArgList and ArgListValues attributes 42

ArgListValues attribute for agents registered as V50 and later 43
Overview of the name-value tuple format 43
About the entry point timeouts 44
ArgListValues attribute for agents registered as V40 and earlier 45

Chapter 3 Creating entry points in C++

About creating entry points in C++ 50
Entry point examples in this chapter 51
Data Structures 52
Syntax for C++ entry points 53
Syntax for C++ VCSAgStartup 53
Syntax for C++ monitor 54
Syntax for C++ info 55
Syntax for C++ online 59
Syntax for C++ offline 60
Syntax for C++ clean 61
Syntax for C++ action 62
Syntax for C++ attr_changed 64
Syntax for C++ open 66
Syntax for C++ close 66
Syntax for C++ shutdown 68
Agent framework primitives 69
VCSAgRegisterEPStruct 69
VCSAgSetCookie 69
VCSAgSetCookie2 69
VCSAgRegister 71
VCSAgUnregister 72
VCSAgGetCookie 73
VCSAgStrlcpy 74
VCSAgStrlcat 74
VCSAgSnprintf 74
VCSAgCloseFile 74
VCSAgDelString 74
VCSAgExec 75
VCSAgExecWithTimeout 76
VCSAgGenSnmpTrap 78
VCSAgSendTrap 78
VCSAgLockFile 78
VCSAgInitEntryPointStruct 79
VCSAgSetStackSize 79
VCSAgUnlockFile 79
VCSAgDisableCancellation 80
VCSAgRestoreCancellation 80

VCSAgSetEntryPoint 81
VCSAgValidateAndSetEntryPoint 81
VCSAgSetLogCategory 81
VCSAgGetProductName 81
VCSAgIsMonitorLevelOne 81
VCSAgIsMonitorLevelTwo 81
VCSAgMonitorReturn 82
VCSAgSetResEPTimeout 82
VCSAgDecryptKey 82
VCSAgGetConfDir 82
VCSAgGetHomeDir 83
VCSAgGetLogDir 83
VCSAgGetSystemName 83
Agent Framework primitives for container support 84
VCSAgISContainerCapable 84
VCSAgExecInContainerWithTmo 84
VCSAgExecInContainerWithTimeout 84
VCSAgGetUID 85
VCSAgIsPidInContainer 85
VCSAgIsProcInContainer 85
VCSAgGetContainerID2 86
VCSAgGetContainerName2 86

Chapter 4 Creating entry points in scripts

About creating entry points in scripts 88
Rules for using script entry points 88
Parameters and values for script entry points 88
ArgList attributes 89
Examples 89
Syntax for script entry points 90
Syntax for the monitor script 90
Syntax for the online script 90
Syntax for the offline script 90
Syntax for the clean script 91
Syntax for the action script 91
Syntax for the attr_changed script 91
Syntax for the info script 91
Syntax for the open script 92
Syntax for the close script 92
Syntax for the shutdown script 92
Example script entry points 93

Chapter 5 Logging agent messages

About logging agent messages 98
Logging in C++ and script-based entry points 98
Agent messages: format 99
C++ agent logging APIs 100
Agent application logging macros for C++ entry points 101
Agent debug logging macros for C++ entry points 102
Severity arguments for C++ macros 103
Initializing function_name using VCSAG_LOG_INIT 104
Log category 105
Examples of logging APIs used in a C++ agent 106
Script entry point logging functions 109
Using functions in scripts 110
VCSAG_SET_ENVS 110
VCSAG_LOG_MSG 113
VCSAG_LOGDBG_MSG 114
Example of logging functions used in a script agent 116

Chapter 6 Building a custom agent

Files for use in agent development 118
Creating the type definition file for a custom agent 119
Naming convention for the type definition file 119
Requirements for creating the agentTypes.cf file 119
Example: FileOnOffTypes.cf 119
Example: Type definition for a custom agent that supports intentional offline 119
Adding the custom type definition to the configuration 120
Building a custom agent on UNIX 120
Implementing entry points using scripts 120
Example: Using script entry points on UNIX 122
Example: Using VCSAgStartup() and script entry points on UNIX 124
Implementing entry points using C++ 125
Example: Using C++ entry points on UNIX 125
Example: Using C++ and script entry points on UNIX 128
Installing the custom agent 131
Defining resources for the custom resource type 131
Sample resource definition 131

Chapter 7 Testing agents

About testing agents 134
Using debug messages 134
Debugging agent functions (entry points). 134
Debugging the agent framework 135
Using the engine process to test agents 135
Test commands 136
Using the AgentServer utility to test agents 137

Chapter 8 Static type attributes

About static attributes 142
Overriding static type attributes 142
Static type attribute definitions 143
ActionTimeout 143
AEPTimeout 143
AgentClass 143
AgentFailedOn 143
AgentPriority 143
AgentReplyTimeout 144
AgentStartTimeout 144
ArgList 144
AttrChangedTimeout 145
CleanTimeout 145
CloseTimeout 145
ContainerOpts 145
ConfInterval 146
FaultOnMonitorTimeouts 147
FireDrill 147
InfoInterval 147
InfoTimeout 148
IntentionalOffline 148
Leveltwomonitorfrequency 148
LogDbg 148
LogFileSize 149
ManageFaults 149
MonitorInterval 150
MonitorStatsParam 150
MonitorTimeout 151
NumThreads 152
OfflineMonitorInterval 152
OfflineTimeout 152
OnlineRetryLimit 153

OnlineTimeout 153
OnlineWaitLimit 153
OpenTimeout 153
Operations 153
RegList 154
RestartLimit 155
ScriptClass 155
ScriptPriority 155
SupportedActions 156
ToleranceLimit 156

Chapter 9 State transition diagrams

State transitions 158
State transitions with respect to ManageFaults attribute 167

Chapter 10 Internationalized messages

Creating SMC files 176
SMC format 176
Example SMC file 176
Formatting SMC files 177
Naming SMC files, BMC files 177
Message examples 177
Using format specifiers 178
Converting SMC files to BMC files 179
Storing BMC files 179
Displaying the contents of BMC files 179
Using BMC Map Files 180
Location of BMC Map Files 180
Creating BMC Map Files 180
Example BMC Map File 180
Updating BMC Files 181

Appendix A Using pre-5.0 VCS agents

Using pre-5.0 VCS agents and registering them as V51 or later 183
Outline of steps to change V40 agents V51 183
Name-value tuple format for agents registered as V51 or later 184
Sourcing ag_i18n_inc modules in script entry points 186
Guidelines for using pre-VCS 4.0 Agents 187
Log messages in pre-VCS 4.0 agents 188
Mapping of log tags (pre-VCS 4.0) to log severities (VCS 4.0) 188
How Pre-VCS 4.0 Messages are Displayed by VCS 4.0 and Later 189

Comparing Pre-VCS 4.0 APIs and VCS 4.0 Logging Macros 189
Pre-VCS 4.0 Message APIs 190
VCSAgLogConsoleMsg 190
VCSAgLogI18NMsg 191
VCSAgLogI18NMsgEx 192
VCSAgLogI18NConsoleMsg 193
VCSAgLogI18NConsoleMsgEx 194

Index 195

Chapter 1

Introduction

- About VCS agents
- How agents work
- About developing an agent

About VCS agents

Agents are programs that manage resources, such as a disk group or a mount point, within a server farm environment. Each type of resource requires an agent. The agent acts as an intermediary between VCS and the resources 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 the 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 *Bundled Agents Reference Guide*.

Agents packaged separately from the product for use with VCS are referred to as *high availability agents*. They include agents for Oracle, for example. Contact your Symantec sales representative for information on how to purchase these agents for your configuration.

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

How agents work

A single agent can manage multiple resources of the same type on one system. For example, the NIC agent manages all NIC resources. The resources to be managed are those defined within the VCS configuration.

The VCS daemon that runs on the systems hosting the applications is HAD.

When the VCS process HAD comes up on a system, it automatically starts the agents required for the types of resources that are to be managed on the system.

The daemon provides the agents the specific configuration information for those resources.

An agent carries out the commands from VCS to bring resources online, monitor their status, and take them offline, as needed. When an agent crashes or hangs, VCS detects the fault and restarts the agent.

About the agent framework

The 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 configuration attributes, such as RestartLimit and MonitorInterval. When an agent's code is built in C++, the agent framework is compiled in the agent with an include statement. When an agent is built using script languages, such as shell or Perl, the provided agent binary: Script51Agent on UNIX or the DefaultAgent.dll on Windows provides the agent framework functions. The agent framework handles much of the complexity that need not concern the agent developer.

Resource type definitions

The agent for each type of resource requires a resource type definition that 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 attributes to provide error checking, and to provide default values for certain attributes that affect all resources of that resource type.

One of the attributes that is defined for the IP resource type is the Address attribute, which stores the actual IP address of a specific IP resource. This attribute is defined as a 'string-scalar' as the use of periods to denote an IP address makes the acceptable values for the attribute a string.

About agent functions (entry points)

An entry point is either a C++ function or a script (shell or Perl, for example) used by the agent to carry out a specific task on a resource. The agent framework supports a specific set of entry points, each of which is expected to do a different task and return. For example, the online entry point brings a resource online.

See "Supported entry points" on page 26.

An agent developer implements the entry points for a resource type that the agent uses to carry out the required tasks on the resources of that type. For example, in the online entry point for the Mount resource type, the agent developer includes the logic to mount a file system based on the parameters provided to the entry point. These parameters are attributes for a particular resource, for example, mount point, device name, and mount options. In the monitor entry point, the agent developer checks the state of the mount resource and returns a code to indicate whether the mount resource is online or offline.

See "About agent entry points" on page 26.

About on-off, on-only, and persistent resources

Different types of resources require different types of control, requiring implementation of different entry points. Resources can be classified as on-off, on-only, or persistent.

- On-off resources

  Most resources are on-off, meaning agents start and stop them as required. For example, VCS assigns an IP address to a specified NIC when bringing a resource online and removes the assigned IP address when taking the resource 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 on-off resources, all entry points can be implemented.

- On-only resources

  An on-only resource is brought online, but it is not taken offline when the associated service group is taken offline. For example, in the case of the FileOnOnly resource, the engine creates the specified file when required, but does not delete the file if the associated service group is taken offline. For agents of on-only resources, the offline entry point is not needed or invoked.

- Persistent resources

  A persistent resource cannot be brought online or taken offline, yet the resource must be present in the configuration to allow the resource to be monitored. For example, a NIC resource cannot be started or stopped, but it is required to be operational in order for the associated IP address to

function properly. The agent monitors persistent resources to ensure their status and operation. An agent for a persistent resource does not require or invoke the online or offline entry points. It uses only the monitor entry points.

About attributes

VCS has the following types of attributes, depending on the object the attribute applies to.

Resource type attributes

Attributes associated with resource types in VCS. These can be further classified as:

- **Type-independent**—Attributes that all agents (or resource types) understand. Examples: RestartLimit and MonitorInterval; these can be set for any resource type.
  Typically, these attributes are set for all resources of a specific type. For example, if you set the MonitorInterval for the IP resource type, the same value applies to all resources of type IP. You can also override the values of these attributes, that is, you can configure a different attribute value for each resource of this type.
- **Type-dependent**—Attributes that apply to a particular resource type. Examples: The MountPath attribute applies only to the Mount resource type. The Address attribute applies only to the IP resource type.
  Attributes defined in the types file (types.*platform*. xml) apply to all resources of the resource type. When you configure resources, you can assign resource-specific values to these attributes, which appear in the main.cf file.
  For example, the PathName attribute for the FileOnOff resource type is type-dependent, and can take a resource-specific value when configured.
- **Static**—These attributes apply to all resource types and can have a different value per resource type. You can override some static attributes and assign them resource-specific values. These attributes are prefixed with the term `static` and are not included in the resource's argument list.

Attribute data types

VCS supports the following data types for attributes.

String — A string is a sequence of characters. If the string contains double quotes, the quotes must be immediately preceded by a backslash. 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 or eth0 does not require quotes as it contains only letters and numbers. However a string defining an IP address contains periods and requires quotes, such as: “192.168.100.1”

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.

Boolean — A boolean is an integer, the possible values of which are 0 (false) and 1 (true).

Attribute dimensions

VCS attributes have the following dimensions.

Scalar — A scalar has only one value.

For example:

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

Vector — A vector is an ordered list of values. Each value is indexed using a positive integer beginning with zero.

Use a comma (,) or a semi-colon (;) to separate values.

A set of brackets ([]) after the attribute name denotes that the dimension is a vector.

Example snippet from the type definition file for an agent:

```
str BackupSys[]
```

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

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

For example, an agent’s ArgList is defined as:

```
static str ArgList[] = {RVG, DiskGroup, Primary, SRL,
Links}
```

Keylist

A keylist is an unordered list of strings, and each string is unique within the list.

Use a comma (,) or a semi-colon (;) to separate values.

For example, to designate the list of systems on which a service group will be started with VCS (usually at system boot):

```
AutoStartList = {SystemA; SystemB; SystemC}
```

For example:

```
keylist BackupVols = {}
```

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

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

Association

An association is an unordered list of name-value pairs.

Use a comma (,) or a semi-colon (;) to separate values.

A set of braces ({}) after the attribute name denotes that an attribute is an association.

For example, to designate the list of systems on which the service group is configured to run and the system's priorities:

```
SystemList = {SystemA=1, SystemB=2, SystemC=3}
```

For example:

```
int BackupSysList {}
```

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

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

Attribute scope across systems: global and local attributes

An attribute whose value is the same across all systems on which the service group is configured *global* in scope. An attribute whose value applies on a per-system basis is *local* in scope.

The at operator (@) indicates the system to which a local value applies.

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 =
    "166.98.16.105" }
Device@sysb = { le0 = "166.98.16.104", qfe3 =
    "166.98.16.106" }
NetMask = "255.255.255.0"
ArpDelay = 5
RouteOptions@sysa = "default 166.98.16.103 0"
RouteOptions@sysb = "default 166.98.16.104 0"
```

)

In the preceding example, the value of the NetMask attribute is “255.255.255.0” on all systems, whereas the values of the Device attribute and the RouteOptions attribute are different on sysa and sysb.

Attribute life: temporary attributes

You can define temporary attributes in the types file. The values of temporary attributes remain in memory as long as the VCS HAD process is running. Values of temporary attributes are not available when the HAD process is restarted.

These attribute values are not stored in the main.cf file.

Temporary attributes cannot be converted to permanent, and vice-versa. When you save a configuration, VCS saves the temporary attribute definitions and their default values in the type definition file.

You can modify attribute values only while VCS is running.

About intentional offline of applications

Certain agents can identify when an application has been intentionally shut down outside of VCS control.

If an administrator intentionally shuts down an application outside of VCS control, VCS does not treat it as a fault. VCS sets the service group state as offline or partial, depending on the state of other resources in the service group.

This feature allows administrators to stop applications without causing failovers.

See “IntentionalOffline” on page 148.

About developing an agent

Before creating the agent, some considerations and planning are required, especially regarding the type of the resource for which the agent is being created.

Considerations for the application

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

- The application must be capable of being started by a defined procedure. There must be some means of starting the application's external resources such as file systems that store databases, or IP addresses used for listener processes, and so on.
- 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 uniquely. Monitoring can be simple or in-depth so as to achieve a high level of confidence in the operation of the application. Monitoring an application becomes more effective when the monitoring procedure resembles the actual activity of the application's user.
- For failover capability, the application must be capable of storing data on shared disks rather than locally or in memory, and each system must be capable of accessing the data and all information required to run the application.
- The application must be crash-tolerant. 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 server farm; 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 server farm.

High-level overview of the agent development process

The steps to create and implement an agent are described by example in later chapters.

Developing the entry points

Decide whether to implement the agent entry points using C++ code, scripts, or a combination of the two.

See “Considerations for using C++ or script entry points” on page 37.

Create the entry points.

See Chapter 3, “Creating entry points in C++”.

See Chapter 4, “Creating entry points in scripts”.

Building the agent

Build the agent, create required files, and place the agent in specific directories.

The types file contains definitions of resource types that come bundled with VCS. The file name is:

types.cf

A custom resource type definition should be placed in a file that specifies the name of the custom resource. for example:

MyResourceTypes.cf

This file is referenced as an “include” statement in theVCS configuration file, `main`.cf.

See “Creating the type definition file for a custom agent” on page 119.

For more information about the resource type definition, see

Chapter 6, “Building a custom agent” on page 117.

For building an agent, sample files are provided.

Testing the agent

Test the agent using the Agent Server utility or by defining the resource type in a configuration.

See Chapter 7, “Testing agents” on page 133.

Chapter 2

Agent entry point overview

- About agent entry points
- Agent entry points described
- Return values for entry points
- Considerations for using C++ or script entry points
- About the ArgList and ArgListValues attributes
- About the agent information file

About agent entry points

Developing an agent involves developing the *entry points* that the agent can call to perform operations on a resource, such as to bring a resource online, to take a resource offline, or to monitor the resource.

Supported entry points

The agent framework supports the following entry points:

- `monitor` - determines the status of a resource
- `info` - provides information about an online resource
- `online` - brings a resource online
- `offline` - takes a resource offline
- `clean` - terminates ongoing tasks associated with a resource
- `action` - starts a defined action for a resource
- `attr_changed` - responds to a resource's attribute's value change
- `open` - initializes a resource before the agent starts to manage it
- `close` - deinitializes a resource before the agent stops managing it
- `shutdown` - called when the agent shuts down

See "Agent entry points described" on page 27.

How the agent framework interacts with entry points

The agent framework ensures that only one entry point is running for a given resource at one time. If multiple requests are received or multiple events are scheduled for the same resource, the agent queues them and processes them 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 already being monitored or that has a monitor request pending.

The agent framework is multithreaded. This means a single agent process can run entry points for multiple resources simultaneously. However, if an agent receives a request to take a given resource offline and simultaneously receives a request to close it, it calls the `offline` entry point first. The `close` entry point is called only after the `offline` request returns or times out. If the `offline` request is received for one resource, and the `close` request is received for another, the agent can call both simultaneously.

The entry points supported by agent framework are described in the following sections. With the exception of `monitor`, other entry points are optional. Each may be implemented in C++ or scripts.

Agent entry points described

This section describes each entry point in detail.

About the monitor entry point

The `monitor` entry point typically contains the logic to determine the status of a 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 agent 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.

By default, the monitor entry point runs every sixty seconds (the default value of the MonitorInterval attribute) when a resource is online.

When a resource is expected to be offline, the entry point runs every 300 seconds (the default value for the `OfflineMonitorInterval` attribute).

The `monitor` entry point receives a resource name and `ArgList` attribute values as input (see “ArgList reference attributes” on page 144).

The entry point returns the resource status and the confidence level.

See “Return values for entry points” on page 36.

The entry point returns confidence level only when the resource status is online. The confidence level is informative only and is not used by the engine. It can be referenced by examining the value of ConfidenceLevel attribute.

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.

See “Syntax for script entry points” on page 90.

About the info entry point

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 the `info` entry point collects is stored in the "temp" attribute `ResourceInfo`.

See "About the ResourceInfo attribute" on page 29.

See the *Administrator's Guide* for information about "temp" attributes.

The (script) 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 key value
```

or

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

Refer to the `hares` manual page for more information on modifying values of string-association attributes.

See "About the ResourceInfo attribute" on page 29.

See "Syntax for C++ entry points" on page 53.

Return values for info entry point

- 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; 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 a user kills the `info` entry point (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.

  See "About logging agent messages" on page 98.
- 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.

About the ResourceInfo attribute

The ResourceInfo attribute is a string association that stores name-value pairs. By default, there are three such name-value pairs:

- State indicates the status (valid, invalid, Stale) of the information contained in the ResourceInfo attribute.
- Msg indicates the output of the info entry point, if any.
- TS indicates the timestamp of when the ResourceInfo attribute was last updated.

These 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 ResourceInfo (string-association) 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 configuration file.

You can display the value of the ResourceInfo by 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. Other key data, including Msg and TS keys, are not touched. You can manually clear values of the ResourceInfo attribute by 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.

Invoking the info entry point

You can invoke the info entry point from the command line for a given online resource using the hares -refreshinfo command.

By setting the InfoInterval attribute to some value other than 0, you can configure the agent to invoke the info entry point periodically for an online resource.

See "InfoInterval" on page 147.

About the online entry point

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 waits the number of seconds indicated by the return value before calling the `monitor` entry point for the resource.

About the offline entry point

The `offline` entry point takes 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 waits the number of seconds indicated by the return value to call the `monitor` entry point for the resource.

About the clean entry point

The `clean` entry point is called 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
};
```

For script-based Clean entry points, the Clean reason is passed as an integer:

```
0 => offline hung
1 => offline ineffective
2 => online hung
3 => online ineffective
4 =>  unexpected offline
5 => monitor hung
```

The above is an enum type, so same integer value is passed irrespective of whether the entry point is written in C++ or is script-based.

- VCSAgCleanOfflineHung
  The `offline` entry point did not complete within the expected time.
  See "OfflineTimeout" on page 152.
- VCSAgCleanOfflineIneffective
  The `offline` entry point was ineffective. The `monitor` entry point returned a status other than OFFLINE after the scheduled invocation of the `offline` entry point for the resource.
- VCSAgCleanOnlineHung
  The `online` entry point did not complete within the expected time.

  (See "OnlineTimeout" on page 153.)
- VCSAgCleanOnlineIneffective
  The `online` entry point was ineffective. The `monitor` entry point scheduled for the resource after the `online` entry point invocation returned a status other than ONLINE.
- 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 147.)

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 "RestartLimit" on page 155.
- Automatically retries the `online` entry point when the attempt to bring a resource online fails.
  See "OnlineRetryLimit" on page 153.
- Enables the 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 run successfully, that is, return an exit code of 0.

About the action entry point

Runs a prespecified action on a resource. Use the entry point to run non-periodic actions like suspending a database or resuming the suspended database.

The SupportedActions attribute is a keylist attribute that lists all the actions that are intended on being supported. Each action is identified by name, or action_token.

See "SupportedActions" on page 156.

For an agent, all action entry points must be either C++ or script-based; you cannot use both C++ and scripts.

Make sure the action scripts reside within an `actions` directory under the agent directory. Create a script for each action. Use the correct action_token as the script name.

For example, a script called suspend defines the actions to be performed when the action_token "suspend" is invoked via the hares -action command.

For C++ entry points, actions are implemented via a switch statement that defines a case for each possible action_token.

See "Syntax for C++ action" on page 62.

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] [-user user@domain] [-domain domaintype]
```

The following example commands show the invocation of the `action` entry point using the example action tokens, `DBSuspend` and `DBResume`:

```
hares -action DBResource DBSuspend -actionargs dbsuspend -sys
Sys1
```

Also,

```
hares -action DBResource DBResume -actionargs dbstart -sys Sys1
```

See also "RegList" on page 154.

Return values for action entry point

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.

The agent framework limits the output of the script-based action entry point to 2048 bytes.

Output refers to information that the script prints to stdout or stderr. When users run the hares -action command, the command prints this output. The output is also logged to the HAD log file.

About the attr_changed entry point

This entry point provides a way to respond to resource attribute value changes. The `attr_changed` entry point is called when a resource attribute is modified, and only if that resource attribute is registered with the agent framework for notification.

See “VCSAgRegister” on page 71.

See “VCSAgUnregister” on page 72.

Registering can be accomplished either though the VCSAgRegister method for C++ based agents or by definition in the RegList for script based agents.

See “RegList” on page 154.

The `attr_changed` 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.

About the open entry point

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

The `open` entry point is also called whenever the `Enabled` attribute for the resource changes from 0 to 1. The entry point receives the resource name and `ArgList` attribute values as input and returns no value.

A resource can be brought online, taken offline, and monitored only if it is managed by an agent. For an agent to manage a resource, the value of the resource’s `Enabled` attribute must be set to 1. T he entry point creates the environment needed for other entry points to function. For example, the entry point could create files required by other entry points for the resource, or perform some resource-specific setup.
points for the resource, or perform some resource-specific setup.

About the close entry point

The `close` entry point is called whenever the `Enabled` attribute for a resource changes from 1 to 0, or when a resource is deleted from the configuration on a running server farm and the state of the resource permits running the close entry point.

Note that a resource is monitored only if it is managed by an agent. For an agent to manage a resource, the resource's `Enabled` attribute value must be set to 1.

See the table below to find out which states of the resource allow running of the close entry point when the resource is deleted on a running server farm. 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.

Table 2-1 States in which CLOSE entry point runs - based on operations type of resource

Resource Type	Online State	Offline State	Probing	Going Offline Waiting	Going Online Waiting
None (persistent)	Yes	N/A	Yes	Yes	N/A
OnOnly	Yes	Yes	Yes	Yes	Yes
OnOff	No	Yes	Yes	No	No

The open and close entry points are related in the sense that the open entry point creates the environment needed for other entry points, while the close entry points cleans the setup created by open entry point.

About the shutdown entry point

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

Return values for entry points

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

Table 2-2 Return values for entry points

Entry Point	Return Values
Monitor	C++ Based Returns ResStateValues: ■ VCSAgResOnline ■ VCSAgResOffline ■ VCSAgResUnknown ■ VCSAgResIntentionalOffline Script-Based Exit values: ■ 99 - Unknown ■ 100 - Offline ■ 101-110 - Online ■ 200 - Intentional Offline ■ Other values - 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, that is, check resource state immediately.
Offline	Integer specifying number of seconds to wait before monitor can check the state of the resource; typically 0, that is, check resource state 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. See Chapter 9, "State transition diagrams" on page 157 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

Considerations for using C++ or script entry points

You may implement an entry point 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 no system overhead for creating a new process is required when they are called. Also, since the entry point invocation is just a function call, the execution of the entry point is relatively faster. However, if the functionality of an entry point needs to be changed, the agent would need to be recompiled to make the changes take effect.
- The advantage to using scripts is that you can modify the entry points dynamically. However, to run the script, a new process is created for each entry point invocation, so the execution of an entry point is relatively slower and uses more system resource compared to the C++ implementation.

Note that you may use C++ or scripts 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 were written in script, the agent must create a new process to run the monitor entry point each time it is called.

See Chapter 3, “Creating entry points in C++” on page 49.

See Chapter 4, “Creating entry points in scripts” on page 87.

About the VCSAgStartup routine

When an agent starts, it uses the routine named `VCSAgStartup` to initialize the agent’s data structures.

If you implement entry points using scripts

If you implement all of the agent's entry points as scripts:

On UNIX, use the Script51Agent binary.

The built-in implementation of VCSAgStartup() in these binaries initializes the agent's data structures such that it causes the agent to look for and execute the scripts for the entry points.

See Chapter 4, “Creating entry points in scripts” on page 87.

If you implement all or some of the entry points in C++

If you develop an agent with at least one entry point implemented in C++, you must implement the function VCSAgStartup() and use the required C++ primitives to register the C++ entry point with the agent framework.

Example: VCSAgStartup with C++ and script entry points

When using C++ to implement an entry point, use the VCSAgInit API and specify the entry point and the function name. In the following example, the function `my_shutdown` is defined as the Shutdown entry point.

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

void VCSAgStartup()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(v51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPShutdown, my_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:

UNIX: $VCS_HOME`/bin/resource_type/<entry_point>`

About the agent information file

The 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 on resources managed by the agent.

Example agent information file (UNIX)

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="version">
    <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>5.0</minvcsversion>
    <!--The agent vendor name-->
    <vendor>Symantec</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:

Table 2-3 Agent information in the agent information XML file

Agent Information	Example
Agent name	`name="FileOnOff"`
Version	`version="x.y"`
Agent description	`<agent_description>Creates, removes, and monitors files.</agent_description>`
Platform. For example Linux, Windows 2000 i386, or Solaris Sparc, or HP-UX .	`<platform>Windows</platform>`
Agent vendor	`<vendor>Symantec</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>`
Compatibility with Cluster Server; 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. The following table describes them. Refer also to the previous XML file example for the FileOnOff agent and see how the `PathName` attribute information is included in the file.

Table 2-4 Description of attribute argument details in XML file

Argument	Description
type	Possible values for attribute type, such as "str" for strings. See "Attribute data types" on page 20.
dimension	Values for the attribute dimension, such as "Scalar;" See "About attributes" on page 19.
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 whether the attribute must be configured to bring the resource online. The GUI displays such attributes with a special indication. If no value is specified for an attribute where the `mustconfigure` argument is true, the resource state becomes "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 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.

Table 2-4 Description of attribute argument details in XML file

Argument	Description
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 the agent XML information file

When the agent XML information file is created, you can implement it as follows:

To implement the agent XML information file in the GUI

1. Make sure the XML file, *agent*.xml, is in the directory $VCS_HOME/bin/*resource_type*
2. Make sure that the command server is running on each node in the server farm.
3. Restart the GUI to have the agent's information shown in the GUI.

About the ArgList and ArgListValues attributes

The ArgList attribute specifies which attributes need to be passed to agent entry points. The agent framework populates the ArgListValues attribute with the list of attributes and their associated values.

In C++ agents, the value of the ArgListValues attribute is 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);
```

In script agents, the value of the ArgListValues attribute is passed as command-line arguments to the entry point script.

ArgListValues attribute for agents registered as V50 and later

For agents registered as V50 or later, the ArgListValues attribute specifies the attributes and their values in tuple format.

- For scalar attributes, there are three components that define the ArgListValues attribute.
  - The name of the attribute
  - The number of elements in the value, which for scalar attributes is always 1
  - The value itself
- For non-scalar attributes (vector, keylist, and association), for each attribute there are N+2 components in the ArgListValues attribute, where N equals the number of elements in the attribute's value.
  - The name of the attribute
  - The number of elements in the attribute's value
  - The remaining N elements correspond to the attribute's value. Note that N could be zero.

Overview of the name-value tuple format

Pre-5.0 agents required that the arguments passed to the entry point to be in the order indicated by the ArgList attribute as it was defined in the resource type. The order of parsing the arguments was determined by their position the resource type definition.

With the agent framework for V50 and later, agents can use entry points that can be passed attributes and their values in a format of name-value tuples. Such a format means that attributes and their values are parsed by the name of the attribute and not by their position in the ArgList Attribute.

The general tuple format for attributes in the ArgList is:

<name> <number_of_elements_in_value> <value>

Scalar attribute format

For *scalar* attributes, whether string, integer, or boolean, the formatting is:

<attribute_name> 1 <value>

Example is:

DiskGroupName 1 mydg

Vector attribute format

For *vector* attributes, whether string or integer, the formatting is:

<attribute_name> <number_of_values_in_vector> <values_in_vector>

Examples are:

MyVector 3 aa cc dd
MyEmptyVector 0

Keylist attribute format

For string *keylist* attributes, the formatting is:

<attribute_name> <number_of_keys_in_keylist> <keys>

Examples are:

DiskAttr 4 hdisk3 hdisk4 hdisk5 hdisk6
DiskAttr 0

Association attribute format

For association attributes, whether string or integer, the formatting is:

<attribute_name> <number_of_keys_and_values> <values_of_keylist>

Examples are:

MyAssoc 4 key1 val1 key2 val2
MyAssoc 0

About the entry point timeouts

Use the AEPTimeout attribute to append the timeout value for a particular entry the list of arguments passed to the entry point.

This feature does not apply to pre-V50 agents.

If you set AEPTimeout to 1, the agent framework passes the timeout value for an entry point as an argument for the entry point in the name-value tuple format.

The name of the attribute that gets passed is called AEPTimeout.

This makes the task of retrieving information about entry point timeout values easy for agent developers. Instead of looking for different strings like MonitorTimeout and CleanTimeout, agent developers just need to look for the string AEPTimeout.

For example, if an agent uses an attribute called PathName set to /tmp/foo, the parameters passed to the monitor entry point are:

If AEPTimeout is set to 0	<resource-name> PathName 1 /tmp/foo.
If AEPTimeout set to 1	<resource-name> PathName 1 /tmp/foo AEPTimeout 1 <value of MonitorTimeout attribute>

Applying the same example for the clean entry point, the parameters are:

If AEPTimeout is set to 0	<resource-name> <clean reason> PathName 1 /tmp/foo
If AEPTimeout is set to 1	<resource-name> <clean reason> PathName 1 /tmp/foo AEPTimeout 1 <value of CleanTimeout attribute>

If the timeout attribute is overridden at the resource level, this mechanism takes care of passing the overridden value to the entry points for that resource.

See "AEPTimeout" on page 143.

ArgListValues attribute for agents registered as V40 and earlier

For agents registered as V40 and earlier, the ArgListValues attribute is an ordered list of attribute values. The attribute values are listed in the same order as in the ArgList attribute.

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" }
)
```

Then, for V50 and later, the parameter attr_val is:

```
attr_val[0] = "IntAttr"
attr_val[1] = "1"       // Number of components in
                        // IntAttr attr value
attr_val[2] = "100"     // Value of IntAttr
attr_val[3] = "StringAttr"
attr_val[4] = "1"       // Number of components in
                        // StringAttr attr value
attr_val[5] = "Oracle" // Value of StringAttr
attr_val[6] = "VectorAttr"
attr_val[7] = "3"       // Number of components in
```

```
                        // VectorAttr attr value
attr_val[8] = "vol1"
attr_val[9] = "vol2"
attr_val[10] = "vol3"
attr_val[11] = "AssocAttr"
attr_val[12] = "4"      // Number of components in
                        // AssocAttr attr value
attr_val[13] = "disk1"
attr_val[14] = "1024"
attr_val[15] = "disk2"
attr_val[16] = "512"
attr_val[17] = NULL     // Last element
```

Or, for V40 and earlier, the parameter attr_val is:

```
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.
```

Chapter 3

Creating entry points in C++

- About creating entry points in C++
- Data Structures
- Syntax for C++ entry points
- Agent framework primitives
- Agent Framework primitives for container support

About creating entry points in C++

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`*.

If you develop an agent with at least one entry point implemented in C++, you must implement the function VCSAgStartup() and use the required C++ primitives to register the C++ entry point with the agent framework.

A sample file containing templates for creating an agent using C++ entry points is located in:

UNIX: $VCS_HOME/src/agent/Sample

You can use C++ to develop agents for monitoring applications that run in containers, including non-global zones. VCS provides APIs for container support.

See “Agent Framework primitives for container support” on page 84.

Entry point examples in this chapter

In this chapter, the example entry points are shown for an agent named Foo. The example agent has the following resource type definition:

In the types.cf format:

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

For this resource type, the entry points defined are as follows:

Entry Point	What it does in this agent
online	Creates a file as specified by the Pathname attribute
monitor	Checks for the existence of a file specified by the PathName attribute
offline	Deletes the file specified by the PathName attribute
clean	Forcibly deletes the file specified by the PathName attribute
action	Runs a prespecified action
info	Populates the ResourceInfo attribute with the values of the attributes specified by the PathName attribute

Data Structures

The VCSAgResState enumeration supports the following return codes from the monitor entry point.

- VCSAgResOffline
- VCSAgResOnline
- VCSAgResUnknown
- VCSAgResIntentionalOffline (only in V51 and later agents)

```
// 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.
    };
```

Syntax for C++ entry points

This section describes the syntax for C++ entry points.

Syntax for C++ VCSAgStartup

```
void VCSAgStartup();
```

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, offline
// and clean entry points are implemented in C++ and the
// respective function names are res_monitor, res_online,
// res_offline, and res_clean.

#include "VCSAgApi.h"
void VCSAgStartup()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);
}



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) {
...
}
```

Syntax for C++ 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 "Return values for entry points" on page 36.

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()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
}
```

Syntax for C++ 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 non-default keys to the three default keys State, Msg, and TS and initialize the name-value data pairs in the ResourceInfo attribute. This invocation indicates to the entry point that the current value of the ResourceInfo attribute contains only the basic three keys State, Msg, and TS.
2	Update only the non-default key-value data pairs in the ResourceInfo attribute, not the default keys State, Msg, and TS. This invocation indicates that ResourceInfo attribute contains non-default keys in addition to the default keys and only the non-default keys are to be updated. Attempt to add keys with this invocation will result in errors.

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++

Set the `VCSAgValidateAndSetEntryPoint()` parameter to the name of the entry point's function (`res_info`).

Allocate the `info` output buffer in the entry point as shown in the example below. The buffer can be any size (the example uses 80), but the agent framework truncates it to 2048 bytes. For the optional name-value pairs, name and value each have a limit of 4096 bytes (the example uses 15).

Example V51 entry point:

```
extern "C" unsigned int res_info(const char *res_name,
VCSAgResInfoOp resinfo_op, void **attr_val, char **info_output,
char ***opt_update_args, char ***opt_add_args)
{
    struct stat stat_buf;
    int i;
    char **args = NULL;
    char *out = new char [80];

    *info_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 The attr_val array will look like "PathName"
    // "1" "<pathname value>" ... Assuming that PathName is the
    // first attribute in the attr_val array, the value
    // of this attribute will be in index 2 of this attr_val
    // array

    if (attr_val[2]) {

        if ((strlen((CHAR *)(attr_val[2])) != 0) &&
            (stat((CHAR *)(attr_val[2]), &stat_buf) == 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[2]);
            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;

} // End of entry point definition
```

Syntax for C++ online

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

You may select any name for the function.

Set the VCSAgValidateAndSetEntryPoint() parameter to the name of the entry point's function.

In the following example, the function res_online is defined as the Online entry point.

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()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
}
```

Syntax for C++ offline

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

You may select any name for the function.

Set the VCSAgValidateAndSetEntryPoint() parameter to the name of the entry point's function.

In the following example, the function res_offline is defined as the Offline entry point.

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()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);
}
```

Syntax for C++ clean

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

You may select any name for the function.

Set the VCSAgValidateAndSetEntryPoint() parameter to the name of the entry point's function.

In the following example, the function res_clean is defined as the Clean entry point.

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()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);
}
```

Syntax for C++ action

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

The parameters passed to the C++ action entry point are described as follows using the example that the user fires

$> hares -action res1 myaction...

from the command-line or the equivalent from the GUI.

- res_name: This is an input parameter. The name of the resource in whose context the action entry point is being invoked. In the above example, res_name would be set to "res1".
- action_token: This is an input parameter. This gives the name of the action that the user wants to run. In the above example, action_token would be set to "myaction".

  If the user ran

  ```
  $> hares -action res1 youraction ...
  ```

  then the same function above will get invoked but action_token will be set to "youraction". This parameter enables different actions to be implemented for the same agent which will all get handled in the same function above.
- attr_val: This is an input parameter. This contains the ArgListValues of the resource for which the action is invoked.
- args: This is an input parameter. This contains the list of strings that are passed to the "-actionargs" switch when invoking the "hares -action" command.

  ```
  $> hares -action res1 myaction -actionargs foo bar fubar -sys
  ...
  ```

  would give "foo", "bar" and "fubar" in the args parameter.
- action_output: This is an output parameter. Any output that the agent developer wants the user to see as a result of invoking the "hares -action" command needs to be filled into the buffer whose pointer is given by this parameter. The maximum number of characters that will be displayed to the user is 2048 (2K).

Use the VCSAgValidateAndSetEntryPoint() API to register the name of the function that implements the action entry-point for the agent.

For example:

```
extern "C"
unsigned int res_action (const char *res_name, const char
    *token,void **attr_val, char **args, char
    *action_output)
```

```
    {
const int output_buffer_size = 2048;
    //
    // 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 for which no action is implemented yet
    //
VCSAgSnprintf(action_output, output_buffer_size, "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;
}
}
```

Syntax for C++ 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 "About the ArgList and ArgListValues attributes" on page 42.

You may select any name for the function.

Set the VCSAgValidateAndSetEntryPoint() parameter to the name of the entry point's function.

In the following example, the function res_attr_changed is defined as the attr_changed entry point.

Note: This entry point is called only if you register for change notification using the primitive "VCSAgRegister" on page 71, or the agent parameter RegList (see "RegList" on page 154).

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()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);
}
```

Syntax for C++ open

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

You may select any name for the function.

Set the VCSAgValidateAndSetEntryPoint() parameter to the name of the entry point's function.

In the following example, the function res_open is defined as the Open entry point.

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()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPOpen, res_open);
    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);
}
```

Syntax for C++ close

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

You may select any name for the function.

Set the VCSAgValidateAndSetEntryPoint() parameter to the name of the entry point's function.

In the following example, the function res_close is defined as the Close entry point.

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()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPClose, res_close);
    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);
}
```

Syntax for C++ shutdown

```
void res_shutdown();
```

You may select any name for the function.

Set the VCSAgValidateAndSetEntryPoint() parameter to the name of the entry point's function.

In the following example, the function res_shutdown is defined as the Shutdown entry point.

For example:

```
#include "VCSAgApi.h"

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

void VCSAgStartup()
{
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPShutdown,
res_shutdown);
    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);
}
```

Agent framework primitives

Primitives are C++ methods implemented by the agent framework. The following sections define the primitives.

See also:

"Agent Framework primitives for container support" on page 84

VCSAgRegisterEPStruct

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

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

VCSAgSetCookie

Deprecated. See "VCSAgSetCookie2" on page 69.

VCSAgSetCookie2

```
void *VCSAgSetCookie2(const char *name, void *cookie)
```

This primitive requests the agent framework to store a cookie given by the void *cookie parameter. If there is a value already associated with the cookie, the primitive sets the new value and atomically returns the old value. If there is no value associated with the cookie, the primitive returns NULL.

This value, which is transparent to the agent framework, can be obtained by calling the primitive `VCSAgGetCookie()`. A cookie is not stored permanently. It is lost when the 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();
                           VCSAgSetCookie2(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();
                           VCSAgSetCookie2(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 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 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 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 agent framework get the cookie set by an earlier call to `VCSAgSetCookie2()`. 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() {
                ...
}
void  res_open(const char *res_name, void **attr_val) {
                if (VCSAgGetCookie(res_name) == NULL) {
                            void *key = get_key();
                            VCSAgSetCookie2(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();
                            VCSAgSetCookie2(res_name, key);
                }
                // Use the key for testing if the resource is
                // online, and set the state accordingly.
                ...
                return state;
}
```

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

VCSAgCloseFile

```
void VCSAgCloseFile(void *vp)
```

Thread cleanup handler to close a file. The input (that is, vp) must be a file descriptor.

VCSAgDelString

```
void VCSAgDelString(void *vp)
```

Thread cleanup handler to delete a (char *). The input (vp) must be a pointer to memory allocated using “new char[xx]”.

VCSAgExec

```
int VCSAgExec(const char *path, char *const argv[], char *buf, long
buf_size, unsigned long *exit_codep)
```

Fork a new process, exec a program, wait for it to complete, and return the status. Also, capture the messages from stdout and stderr to buf. Caller must ensure that buf is of size >= buf_size.

VCSAgExec is a forced cancellation point. Even if the C++ entry point that calls VCSAgExec disables cancellation before invoking this API, the thread can get cancelled inside VCSAgExec. Therefore, the entry point must make sure that it pushes appropriate cancellation cleanup handlers before calling VCSAgExec. The forced cancellation ensures that a service thread running a timed-out entry point does not keep running or waiting for the child process created by this API to exit, but instead honors a cancellation request when it receives one.

Explanation of arguments to the function:

path	Name of the program to be executed.
argv	Arguments to the program. argv[0] must be same as path. The last entry of argv must be NULL. (Same as execv syntax)
buf	Buffer to hold the messages from stdout or stderr. Caller must supply it. This function will not allocate. When this function returns, buf will be NULL-terminated.
bufsize	Size of buf. If the total size of the messages to stdout/stderr is more than bufsize, only the first (buf_size - 1) characters will be returned.
exit_codep	Pointer to a location where the exit code of the executed program will be stored. This value should interpreted as described by wait() on Unix

Return value: VCSAgSuccess if the execution was successful.

Example:

```
//
// ...
//
char **args = new char* [3];
char buf[100];
unsigned int status;

args[0] = "/usr/bin/ls";
args[1] = "/tmp";
args[2] = NULL;
```

```
int result = VCSAgExec(args[0], args, buf, 100, &status);

if (result == VCSAgSuccess) {

// Windows NT:
printf("Exit code of %s is %d\n", args[0], status);

// Unix:
if (WIFEXITED(status)) {
printf("Child process returned %d\n", WEXITSTATUS(status));
}
else {
printf("Child process terminated abnormally(%x)\n", status);
}

}
else {
printf("Error executing %s\n", args[0]);
}
//
// ...
//
```

VCSAgExecWithTimeout

```
int VCSAgExecWithTimeout(const char *path, char *const argv[],
unsigned int timeout, char *buf, long buf_size, unsigned long
*exit_codep)
```

Fork a new process, exec a program, wait for it to complete, return the status. If the process does not complete within the timeout value, kill it. Also, capture the messages from stdout or stderr to buf. The caller must ensure that buf is of size >= buf_size. VCSAgExecWithTimeout is a forced cancellation point. Even if the C++ entry point that calls VCSAgExecWithTimeout disables cancellation before invoking this API, the thread can get cancelled inside VCSAgExecWithTimeout. So the entry point needs to make sure that is pushes appropriate cancellation cleanup handlers before calling VCSAgExecWithTimeout. The forced cancellation ensures that a service thread running a timed out entry point does not keep running or waiting for the child process created by this API to exit but instead honors a cancellation request when it receives one.

Explanation of arguments to the function:

path	Name of the program to be executed.
argv	Arguments to the program. argv[0] must be same as path. The last entry of argv must be NULL. (Same as execv syntax).
timeout	Number of seconds within which the process should complete its execution. If zero is specified, this API defaults to VCSAgExec(), meaning the timeout is to be ignored. If the timeout value specified exceeds the time left for the entry point itself to timeout, the maximum possible timeout value is automatically used by this API. For example, if the timeout value specified in the API is 40 seconds, but the entry point itself times out after the next 20 seconds, the agent internally sets the timeout value for this API to 20-3=17 seconds. The 3 seconds are a grace period between the timeout for the process created using this API and the entry point process timeout.
buf	Buffer to hold the messages from stdout/stderr. The caller must supply it. This function does not allocate. When this function returns, buf is NULL-terminated.
bufsize	Size of buf. If the total size of the messages to stdout/stderr is more than bufsize, only the first (buf_size - 1) characters is returned.
exit_codep	Pointer to a location where the exit code of the executed program is stored. This value should interpreted as described by wait() on Unix

Return value: VCSAgSuccess if the execution is successful.

VCSAgGenSnmpTrap

```
void VCSAgGenSnmpTrap(int trap_num, const char *msg, VCSAgBool
is_global)
```

This API is used to send a notification via SNMP and/or SMTP. The ClusterOutOfBand trap is used to send notification messages from the agent entry points.

Explanation of arguments to the function:

trap_num	The trap identifier. This number is appended to the agents trap oid to generate a unique trap oid for this event.
msg	The notification message to be sent.
is_global	A boolean value indicating whether or not the event for which the notification is being generated is local to the system where the agent is running.

VCSAgSendTrap

```
void VCSAgSendTrap(const CHAR *msg)
```

This API is used to send a notification through the notifier process. The input (that is, msg) is the notification message to be sent.

VCSAgLockFile

```
int VCSAgLockFile(const char *fname, VCSAgLockType ltype,
VCSAgBlockingType btype, VCSAgErrnoType *errp)
```

Get a read or write (that is, shared or exclusive) lock on the given file. Both blocking and non-blocking modes are supported. Returns 0 if the lock could be obtained, or returns VCSAgErrWouldBlock if non-blocking is requested and the lock is busy. Otherwise returns -1. Each thread is considered a distinct owner of locks.

Mt-safe; deferred cancel safe.

Warning: Do not do any operations on the file (ex, open, or close) within this process, except through the VCSAgReadLockFile(), VCSAgWriteLockFile(), and VCSAgUnlockFile() interfaces.

VCSAgInitEntryPointStruct

```
void VCSAgInitEntryPointStruct(VCSAgAgentVersion agent_version)
```

This primitive enables agents to initialize the agent framework and the entry point struct depending on the agent framework version passed to this API.

Examples:

```
VCSAgInitEntryPointStruct(V50);
VCSAgInitEntryPointStruct(V51);
```

If the agent registers with a version greater than V40, the following entry points are supported:

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

For information on available registration version numbers, check the VCSAgApiDefs.h header file.

VCSAgSetStackSize

```
void VCSAgSetStackSize(int i)
```

The agent framework sets the default stack size for threads in agents to 1MB. Use VCSAgStackSize to set the calling thread's stack size to the specified value.

VCSAgUnlockFile

```
int VCSAgUnlockFile(const char *fname, VCSAgErrnoType *errp)
```

Release read or write (i.e shared or exclusive) lock on the given file. Returns 0, if the lock could be released, or else returns -1.

Mt-safe; deferred cancel safe.

Mt-safe; deferred cancel safe.

Warning: Do not do any operations on the file (ex, open, or close) within this process, except through the VCSAgReadLockFile(), VCSAgWriteLockFile(), and VCSAgUnlockFile() interfaces.

VCSAgDisableCancellation

```
int VCSAgDisableCancellation(int *old_statep)
```

If successful, return 0 and set old_statep to the previous cancellation state.

VCSAgRestoreCancellation

```
int VCSAgRestoreCancellation(int desired_state)
```

If successful, return 0 and the desired_state is set as the current cancellation state.

VCSAgSetEntryPoint

This primitive is deprecated.

See “VCSAgValidateAndSetEntryPoint” on page 81

VCSAgValidateAndSetEntryPoint

```
void VCSAgValidateAndSetEntryPoint(VCSAgEntryPoint ep, f_ptr)
```

This primitive enables an agent developer to register any C++ entry point with the agent framework.

VCSAgEntryPoint is an enumerated data type defined in VCSAgApiDefs.h.

For example:

```
VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, my_monitor_func);
```

VCSAgSetLogCategory

```
void VCSAgSetLogCategory(int cat_id)
```

Sets the log category of the agent to the value specified in cat_id.

VCSAgGetProductName

```
const CHAR *VCSAgGetProductName()
```

An API for C++ entry points to be able to get the name of the product for logging purposes.

VCSAgIsMonitorLevelOne

VCSAgBool VCSAgIsMonitorLevelOne();

See “VCSAgIsMonitorLevelTwo” on page 81.

VCSAgIsMonitorLevelTwo

VCSAgBool VCSAgIsMonitorLevelTwo();

VCSAgIsMonitorLevelOne and VCSAgIsMonitorLevelTwo can be used by the monitor entry point to find out whether the actions to be taken by the monitor entry point should correspond to basic monitoring or detailed monitoring.

These APIs work together with the LevelTwoMonitorFrequency attribute.

See “Leveltwomonitorfrequency” on page 148.

For example, if you set LevelTwoMonitorFrequency to 5, then every MonitorInterval seconds, VCSAgIsMonitorLevelOne() returns TRUE. Additionally, every 5th monitor cycle, VCSAgIsMonitorLevelTwo() API also returns TRUE. This helps users configure detailed monitoring more effectively

and the agent developer does not need to make any calculations to find out when to execute the logic related to detail monitoring.

VCSAgMonitorReturn

```
VCSAgResState VCSAgMonitorReturn(VCSAgResState state, s32
conf_level, const CHAR *conf_msg)
```

VCSAgResState state: The state of the resource as found by the monitor entry point.

int conf_level: The confidence level with which the resource was found to be online. This can be a number from 10 to 100.

const char * conf_msg: If the resource is being reported as ONLINE from the monitor entry point with a confidence level lower than 100, this parameter accepts a string containing the reason for the lower confidence level for the resource. If the confidence level reported is 100 or if the resource state is reported as Offline or IntentionalOffline, the confidence message will get automatically cleared even if agent developer provides a confidence message string to this API.

VCSAgSetResEPTimeout

```
void VCSAgSetResEPTimeout(s32 tmo)
```

This API allows an agent entry point to extend its timeout value dynamically from within the entry point's execution context. This might be required if a command executed from the entry point takes longer than expected to complete and the entry point does not want to timeout. Symantec recommends using this API with caution because the intent of timeouts is to make sure that entry points finish on time.

VCSAgDecryptKey

```
VCSAgDecryptKey(char *key, char *outbuf, int buflen);
```

This API lets you decrypt an encrypted string passed in the ArgListValues by the user. Typically users encrypt string attribute values for passwords using the encryption commands provided by VCS. An entry point can use this API to decrypt the encrypted string and get the original string.

VCSAgGetConfDir

```
void VCSAgGetConfDir(char *buf, int bufsize)
```

Returns the name of the VCS configuration directory.

If the VCS_CONF environment variable is set, the command returns the value of the variable, otherwise it returns the default value. .

Caller must supply the buffer

VCSAgGetHomeDir

```
void VCSAgGetHomeDir(char *buf, int bufsize)
```

Returns the name of VCS home directory. If the VCS_HOME environment is configured, the command returns the value of the the variable, otherwise it returns the default value.

Caller must supply the buffer

VCSAgGetLogDir

```
VCSAgGetLogDir(char *buf, int bufsize)
```

Returns the name of VCS log directory. If the VCS_LOG environment variable is set, the command returns the value of the variable, otherwise it returns the default value if not set.

Caller must supply the buffer

VCSAgGetSystemName

```
void VCSAgGetSystemName(char *buf, int bufsize)
```

Returns the name of the system on which the agent is currently running.

Caller must supply the buffer

Agent Framework primitives for container support

The following APIs are for use in agents that run in AIX WPARs, XRM containers and Solaris zones. Note that zones are supported by Solaris version 10 and above.

Note that the earlier APIs for zones support are deprecated:

- VCSAgGetContainerName
- VCSAgGetContainerID
- VCSAgExecInContainer
- VCSAgISZoneCapable

VCSAgISContainerCapable

```
VCSAgBool VCSAgIsContainerCapable();
```

This API returns either True or False.

- For solaris zones
  If the agent is running on a Solaris 10 (or higher version) system, the API returns True; otherwise it returns False.
  Agents can use this API to decide whether or not to perform zone-specific operations like comparing the zone_id field in the psinfo structure with the ID of the zone name specified in the resource configuration to confirm whether the found process is indeed the process the agent is looking for.
- For XRM
  If the agent is running on a system that has xrm available, the API returns True; otherwise it returns False.
- For WPARs
  If the agent is running on a system that has WPARs available, the API returns True; otherwise it returns False.

VCSAgExecInContainerWithTmo

This API is deprecated.

See “VCSAgExecInContainerWithTimeout” on page 84.

VCSAgExecInContainerWithTimeout

```
int VCSAgExecInContainerWithTimeout((const CHAR *path, CHAR *const
argv[]  u32 timeout, CHAR *buf, long buf_size, unsigned long
    *exit_codep);
```

This API is similar to the VCSAgExecWithTimeout API. This API can be used by an agent only to execute a particular command or script in a specific container on the system. If there are no containers configured on the system, or if the agent has no need to exec a script in a specific container, the VCSAgExecWithTimeout API should be used.

Memory for buf and exit_codep should be allocated by the calling function.

VCSAgGetUID

```
int VCSAgGetUID(const CHAR *user, int *uid, int *euid, int
*home_exists);
```

This API checks if the given user is valid inside the container as specified in the resource object. The API returns the uid and euid of the user either inside the container if container info is set for the resource or on the global container if container info is not set for the resource. The home_exists parameter indicates if the specified user's home directory exists within the container.

Memory for uid, euid and home_exists must be allocated by the calling function

The API returns 0 on success and 1 on failure

VCSAgIsPidInContainer

```
int VCSAgIsPidInContainer(VCSPID pid);
```

This API checks if the given pid is running inside the container as specified in the resource object.

Return values

- 1 if the proc pid is running inside the container
- 0 if the proc pid is not running inside the container
- -1 if the API cannot verify the container info for the process. This is possible if ContainerType is an invalid value.

VCSAgIsProcInContainer

```
int VCSAgIsProcInContainer(void *psinfop);
```

This API checks if the process corresponding to the given psinfo structure is running inside the container as specified in the resource object.

Return values

- 1 if the proc pid is running inside the container
- 0 if the proc pid is not running inside the container

- -1 if the API cannot verify the container info for the process. This is possible if ContainerType is an invalid value.

VCSAgGetContainerID2

```
int VCSAgGetContainerID2()
```

This API retrieves the ID of the container.

Based on the thread that is implementing the entry point, the agent identifies the resource for which this API is invoked and returns the container ID for that resource. The container ID is the ID of the container specified in the ContainerInfo attribute as the value of the Name key.

Return Values

- -1, if the resource or container name is NULL or the container is DOWN or the container is not applicable to the OS version the agent is running on.
- Non-negative container-id, if the container name is valid and the container is UP.

VCSAgGetContainerName2

```
char *VCSAgGetContainerName2();
```

For Solaris Zones, this API retrieves the name of the container, if set for the specified resource.

For XRM, the API retrieves the name of the Execution Context.

For WPARs, the API retrieves the name of the WPAR.

The API returns a pointer to the container name. It is the responsibility of the caller to free the memory associated with the returned pointer.

The name of the container is the value set in the group-level attribute ContainerInfo for the group the resource belongs to.

Chapter 4

Creating entry points in scripts

- About creating entry points in scripts
- Syntax for script entry points
- Example script entry points

About creating entry points in scripts

On UNIX, script agents use the Script51Agent binary that are shipped with the product. The Script51Agent binaries are located at:

`$VCS_HOME/bin/Script51Agent`

You must implement the `VCSAgStartup` entry point using C++.

You may implement other entry points using C++ or scripts. If you implement no other entry points in C++, the `VCSAgStartup` entry point is not required.

On UNIX, use the Script51Agent binary to develop all entry points as scripts.

You can use Perl or shell scripts to develop entry points.

Rules for using script entry points

Script entry points can be executables or scripts, such as shell or Perl (the product includes a Perl distribution).

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

On UNIX platforms

- In the `VCSAgStartup` entry point, if you do not set a C++ function for an entry point using the VCSAgValidateAndSetEntryPoint() API, then the agent framework assumes the entry point is script-based.
  See "About the VCSAgStartup routine" on page 37.
- Verify the name of the script file is the same as the entry point .
- Place the file in the directory `$VCS_HOME/bin/`*`resource_type`*. If, for example, the `online` script for Oracle were implemented using Perl, the `online` script must be:

  ```
  $VCS_HOME/bin/Oracle/online.pl
  ```
- If you write scripts in shell, verify the PATH environment variable includes the directory where sh is installed.

Parameters and values for script entry points

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. See the entry point description for more information.

See "Syntax for script entry points" on page 90.

ArgList attributes

See “About the ArgList and ArgListValues attributes” on page 42.

Examples

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 the VCS configuration file `main.cf` as:

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

The online script for a V51 agent, when invoked for Bar, resembles:

```
online Bar IntAttr 1 100 StringAttr 1 Oracle VectorAttr 3 vol1
vol2 vol3 AssocAttr 4 disk1 1024 disk2 512
```

Syntax for script entry points

The following paragraphs describe the syntax for script entry points.

Syntax for the monitor script

`monitor` *`resource_name ArgList_attribute_values`*

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

- 99 indicates unknown.
- 100 indicates offline.
- 101 indicates online and a confidence level of 10.
- 102–109 indicates online and confidence levels 20–90.
- 110 indicates online and confidence level 100.
- 200 indicates intentional offline.

If the exit value is not one of the above values, the status is considered unknown.

Syntax for the online script

`online` *`resource_name ArgList_attribute_values`*

The exit value is interpreted as the expected time (in seconds) for the online procedure to be effective. It also means the time (in seconds) that must pass before executing the monitor entry point to validate proper operation. The exit value is typically 0.

Syntax for the offline script

`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.

Syntax for the clean script

`clean` *`resource_name clean_reason 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 152.)

1 - The `offline` entry point was ineffective.

2 - The `online` entry point did not complete within the expected time.
(See "OnlineTimeout" on page 153.)

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 147.)

The exit value is 0 (successful) or 1.

Syntax for the action script

`action` *`resource_name`*
`ArgList_attribute_values_AND_action_arguments`

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

The agent framework limits the action entry point output to 2048 bytes.

Syntax for the attr_changed script

`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 71), or the agent parameter `RegList` (see "RegList" on page 154).

Syntax for the info script

`info` *`resource_name resinfo_op ArgList_attribute_values`*

The attribute `resinfo_op` can 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.

Syntax for the open script

`open` *`resource_name ArgList_attribute_values`*

The exit value is ignored.

Syntax for the close script

`close` *`resource_name ArgList_attribute_values`*

The exit value is ignored.

Syntax for the shutdown script

`shutdown`

The exit value is ignored.

Example script entry points

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

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, followed by the values of the remaining ArgList attributes.

- For agents that are registered as less than V50, the entry point expects the values of the attributes in the order the attributes have been specified in the ArgList attribute.
- For agents registered as V50 and greater, the entry point expects the ArgList in tuple format: the name of the attribute, the number of elements in the attribute's value, and the value.

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

Monitor entry point for FileOnOff

When the agent's `monitor` entry point is called by the agent, the entry point expects the name of the resource as the first argument, followed by the values of the remaining ArgList attributes.

- For agents that are registered as less than V50, the entry point expects the values of the attributes in the order the attributes have been specified in the ArgList attribute.
- For agents registered as V50 and greater, the entry point expects the ArgList in tuple format: the name of the attribute, the number of elements in the attribute's value, and the value.

If the file exists it returns exit code 110, indicating the resource is online with 100% confidence. If the file does not exist the monitor returns 100, indicating the resource is offline. If the state of the file cannot be determined, the monitor returns 99.

```
#!/bin/sh
# FileOnOff Monitor script
# Expects Resource Name and Pathname

. $VCS_HOME/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”\
    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
```

Monitor entry point with intentional offline

This script includes the intentional offline functionality for the MyCustomApp agent.

See “About on-off, on-only, and persistent resources” on page 18.

Note that the method to detect intentional offline of an application depends on the type of application. The following example assumes that the application writes a status code into a file if the application is intentionally stopped.

```
#!/bin/sh

. ${CLUSTER_HOME}/bin/ag_i18n_inc.sh

ResName=$1; shift;
VCSAG_SET_ENVS $ResName
// Obtain the attribute values from ArgListValues
parse_arglist_values();
RETVAL=$?

if [ ${RETVAL} -eq ${VCSAG_RES_UNKNOWN} ]; then
    // Could not get all the required attributes from
ArgListValues
    exit $VCSAG_RES_UNKNOWN;
fi

// Check if the application's process is present in the ps
// output
check_if_app_is_running();
RETVAL=$?

if [ ${REVAL} -eq ${VCSAG_RES_ONLINE} ]; then
    // Application process found
    exit $VCSAG_RES_ONLINE;
fi

// Application process was not found; Check if user gracefully
// shutdown the application
grep "MyCustomAppCode 123 : User initiated shutdown command"
${APPLICATION_CREATED_STATUS_FILE}
RETVAL=$?

if [ ${REVAL} -eq 0 ]; then
// Found MyCustomAppCode 123 in the application's status
// file that gets created by the application on graceful
//shutdown
    exit $VCSAG_RES_INTENTIONALOFFLINE;
else
// Did not find MyCustomAppCode 123; hence application has
// crashed or gone down unintentionally
    exit $VCSAG_RES_OFFLINE;
fi

// Monitor should never come here
exit $VCSAG_RES_UNKNOWN;
```

Offline entry point for FileOnOff

When the agent's `offline` entry point is called by the agent, the entry point expects the name of the resource as the first argument, followed by the values of the remaining ArgList attributes.

- For agents that are registered as less than V50, the entry point expects the values of the attributes in the order the attributes have been specified in the ArgList attribute.
- For agents registered as V50 and greater, the entry point expects the ArgList in tuple format: the name of the attribute, the number of elements in the attribute's value, and the value.

When the values are accepted by the entry point, the file is deleted.

```
#!/bin/sh
# FileOnOff Offline script
# Expects ResourceName and Pathname
#
. $VCS_HOME/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;
```

Chapter 5

Logging agent messages

- About logging agent messages
- Logging in C++ and script-based entry points
- C++ agent logging APIs
- Script entry point logging functions

About logging agent messages

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

- For information on creating and managing of messages for internationalization, see Chapter 10, "Internationalized messages" on page 175.
- For information on APIs used by VCS 3.5 and earlier, see "Log messages in pre-VCS 4.0 agents" on page 188.

Logging in C++ and script-based entry points

Developers creating C++ agent entry points can use a set of macros for logging application messages or debug messages. Developers of script-based entry points can use a set of methods, or "wrappers," that call the `halog` utility to generate application or debug messages.

Symantec recommends using the ag_i18n_inc subroutines for logging. The subroutines set the category ID for the messages and provide a header for the log message, which includes the resource name and the entry point name.

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 FileOnOff agent's `online` entry point. The message is generated when the agent attempts to bring online a 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 (filename) failed
with error (Is a Directory)
```

The first four fields of the message above consists of the *timestamp*, an uppercase *mnemonic* that represents the product, 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, must use all capital letters. All VCS bundled agents, enterprise agents, and custom agents use the mnemonic: "VCS"

Severity

The severity of each message displays 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-" that defines the product that the message comes from. This ID is assigned by Symantec.
- The category ID is a number in the range of 0 to 65536 assigned by Symantec. The category ID indicates the agent that message came from. For each custom agent, you must contact Symantec 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 105.
- For script-based entry points, the category is set within the VCSAG_SET_ENVS function
  See “VCSAG_SET_ENVS” on page 110.
- 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 ID. 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.
  See “VCSAG_LOG_MSG” on page 113.

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 (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 110).

C++ agent logging APIs

The 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...)
```

Agent application logging macros for C++ entry points

You can use the macro VCSAG_LOG_MSG within C++ agent entry points to log all messages ranging in severity from CRITICAL to INFORMATION to the agent log file. Use the VCSAG_CONSOLE_LOG_MSG macro to send messages to the HAD log. Where the messages are of CRITICAL or ERROR severity, the message is also logged to the console.

The following table describes the argument fields for the application logging macros:

`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 103.
`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, the 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

Use the macros VCSAG_RES_LOG_MSG and VCSAG_LOGDBG_MSG within agent entry points to log debug messages of a specific severity level to the agent log.

Use the `LogDbg` attribute to specify a debug message severity level. See the description of the `LogDbg` attribute ("LogDbg" on page 148). Set the `LogDbg` attribute at the resource type level. The attribute 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 following table describes the argument fields for the application logging macros:

`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 103.
`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, res_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` header file, these enumerated types are defined as:

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

enum VCSAgDbgSev {
        DBG1,
        DBG2,
        DBG3,
        .
        .
        DBG21,
        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 104.

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 “res_offline” would contain:

```
void res_offline (int a, char *b)
{
        VCSAG_LOG_INIT(“res_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 106 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. In the following example, the log category is set to 10051:

```
    VCSEXPORT void VCSDECL VCSAgStartup()
    {
            VCSAG_LOG_INIT("VCSAgStartup");
           VCSAgInitEntryPointStruct(V51);


           VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor,
res_monitor);
           VCSAgValidateAndSetEntryPoint(VCSAgEPOnline,
res_online);
           VCSAgValidateAndSetEntryPoint(VCSAgEPOffline,
res_offline);
           VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);

           VCSAgSetLogCategory(10051);

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

You do not need to set the log category for debug messages, which is 50 by default.

Examples of logging APIs used in a C++ agent

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

void res_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("res_attr_changed");
}
extern "C" unsigned int
res_clean(const char *res_name, VCSAgWhyClean wc, void
                  **attr_val)
{
        VCSAG_LOG_INIT("res_clean");
        if ((attr_val) && (*attr_val)) {
                if ((remove((CHAR *)(*attr_val)) == 0) || (errno
               == ENOENT)) { return 0;       // Success
               }
        }
        return 1;       // Failure
}

void res_close(const char *res_name, void **attr_val)
{
        VCSAG_LOG_INIT("res_close");
}
//
// Determine if the given file is online (file exists) or
// offline (file does not exist).
//
extern "C" VCSAgResState
res_monitor(const char *res_name, void **attr_val, int
*conf_level)
{
        VCSAG_LOG_INIT("res_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
res_online(const char *res_name, void **attr_val) {
   int fd = -1;
   VCSAG_LOG_INIT("res_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
res_offline(const char *res_name, void **attr_val)
{
        VCSAG_LOG_INIT("res_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 res_open(const char *res_name, void **attr_val)
{
        VCSAG_LOG_INIT("res_open");

}
VCSEXPORT void VCSDECL VCSAgStartup()
{
        VCSAG_LOG_INIT("VCSAgStartup");
       VCSAgInitEntryPointStruct(V51);

       VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor,
res_monitor);
       VCSAgValidateAndSetEntryPoint(VCSAgEPOnline,
res_online);
       VCSAgValidateAndSetEntryPoint(VCSAgEPOffline,
res_offline);
       VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);

        VCSAgSetLogCategory(2001);

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

}
```

Script entry point logging functions

For script based entry points, use the functions described in this section for message logging purposes.

Note: Symantec recommends that you do not use the halog command in script entry points.

The logging functions are available in the ag_i18n_inc module.

VCSAG_SET_ENVS: See “VCSAG_SET_ENVS” on page 110.

VCSAG_LOG_MSG: See “VCSAG_LOG_MSG” on page 113.

VCSAG_LOGDBG_MSG: See “VCSAG_LOGDBG_MSG” on page 114.

VCSAG_CONSOLE_LOG_MSG: Logs a message to the HAD log file.

Using 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;
```

VCSAG_SET_ENVS

The VCSAG_SET_ENVS 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_ENVS function sets the following environment variables for a resource:

VCSAG_LOG_CATEGORY	Sets the category ID. For custom agents, Symantec assigns the category ID. See the category ID description in "UMI" on page 99. NOTE: For 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: UNIX: /opt/VRTSvcs/bin/*resource_type* 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 hard coded 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: UNIX: /opt/VRTSvcs/bin/*resource_type/*online Since the entry points are invoked using their absolute paths, this environment variable is set at invocation. The script name variable is can be overridden.
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. See the *Veritas Cluster Server Administrator's Guide* for more information.
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");
```

Example of logging functions used in a 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
 :
 :
VCS_HOME="${VCS_HOME:-/opt/VRTSvcs}"


. $VCS_HOME/bin/ag_i18n_inc.sh

# Assume the category id assigned by Symantec 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
```

Chapter 6

Building a custom agent

- Files for use in agent development
- Creating the type definition file for a custom agent
- Building a custom agent on UNIX
- Installing the custom agent
- Defining resources for the custom resource type

Files for use in agent development

The VCS installation program provides the following files to aid agent development:

Table 6-1 Script Agents

Description	Pathname
Ready-to-use agent that includes a built-in implementation of the `VCSAgStartup` entry point.	UNIX: $VCS_HOME/bin/Script51Agent

Table 6-2 C++ Agents

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

Creating the type definition file for a custom agent

The agent you create requires a resource type definition file. This file performs the function of providing a general type definition of the resource and its unique attributes.

Naming convention for the type definition file

Name the resource type definition file following the convention *resource_type*`Types.cf`. For example, for the resource type XYZ, the file would be `XYZTypes.cf`.

Requirements for creating the agentTypes.cf file

As you examine the previous example, note the following aspects:

- The name of the agent
- The `ArgList` attribute, its name, type, dimension, and its values, which consist of the other attributes of the resource
- The remaining attributes (in this example case there is only the `PathName` attribute), their names, types, dimensions, and descriptions.

Example: FileOnOffTypes.cf

An example types configuration file for the FileOnOff resource:

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

Example: Type definition for a custom agent that supports intentional offline

```
type MyCustomApp (
        static int IntentionalOffline = 1
        static str ArgList[] = { PathName, Arguments }
        str PathName
        str Arguments
)
```

Adding the custom type definition to the configuration

Once you create the file, place it in the directory:

UNIX: $VCS_HOME/conf/config

Building a custom agent on UNIX

The following sections describe different ways to build an agent, using the "FileOnOff" resource as an example. For test purposes, instructions for installing the agent on a single system are also provided.

The examples assume:

- VCS is installed under `/opt/VRTSvcs` by default. If your installation directory is different, change VCS_HOME accordingly.
- You have created a FileOnOff type definition file.
  See "Creating the type definition file for a custom agent" on page 119.

Note the following about the FileOnOff agent entry points. 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.

Implementing entry points using scripts

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.

See "About creating entry points in scripts" on page 88.

If all entry points are scripts, all scripts should be in the directory `$VCS_HOME/bin/`*`resource_type`*. Copy the Script51Agent into the agent directory as `$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`.

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"
        ...
)
```

Example: Using script entry points on UNIX

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.

Example: implementing entry points using scripts

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

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

2 Use the VCS agent /opt/VRTSvcs/bin/Script51Agent as the FileOnOff agent. Copy this file to the following path:

```
/opt/VRTSvcs/bin/FileOnOff/FileOnOffAgent
```

or create a link.

To copy the agent binary:

```
cp /opt/VRTSvcs/bin/Script51Agent
           /opt/VRTSvcs/bin/FileOnOff/FileOnOffAgent
```

To create a link to the agent binary:

```
ln -s /opt/VRTSvcs/bin/Script51Agent
           /opt/VRTSvcs/bin/FileOnOff/FileOnOffAgent
```

3 Implement the `online`, `offline`, and `monitor` entry points using scripts. Use 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
exit 0
```

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

```
# !/bin/sh
# Remove the file specified by the PathName
# attribute.
rm $2
exit 0
```

- 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.

Example: Using VCSAgStartup() and script entry points on UNIX

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.

Example: implementing agent using VCSAgStartup and script entry points

1. Create the following directory:

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

2. Copy the contents from the sample agent directory 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() {
       VCSAgInitEntryPointStruct(V51);

       // Do not configure any entry points because
       // this example does not implement any of them
       // using C++.

       VCSAgSetLogCategory(10041);
       }
   ```

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

   ```
   make
   ```

6. Create a directory for the agent:

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

7. Install the FileOnOff agent.

   ```
   make install AGENT=FileOnOff
   ```

8. Implement the `online`, `offline`, and `monitor` entry points.
   See “Example: Using script entry points on UNIX” on page 122.

Implementing entry points using C++

To implement 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.
3. Name the agent binary: *`resource_type`*`Agent`.
4. Place the agent in the directory `$VCS_HOME/bin/`*`resource_type`*. For example, the agent binary for Oracle would be `$VCS_HOME/bin/Oracle/OracleAgent`.

Example: Using C++ entry points on UNIX

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.

Example: VCSAgStartup and C++ entry points

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

```
// Description: This functions registers the entry points //
void VCSAgStartup()
    {
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgValidateAndSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);
}
```

2. Modify `res_online()`:

```
// 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 res_online(const char *res_name, void **attr_val) {
    WCHAR      **new_args = NULL;
    DWORD      ret;
    HANDLE     hFile = NULL;
    wchar_t    *pathName = NULL;

VCSAG_LOG_INIT("res_online");
VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS, I,"Inside
    online.");
ret = VCSAgGetEncodedArgList(VCSAgUTF8, attr_val, VCSAgUCS2,
    (void ***)&new_args);
if (ret)
{
    VCSAG_LOG_MSG(VCS_NOTICE, 1001, VCS_DEFAULT_FLAGS,
    I,"Unable to get the arguments");
    goto exit;
}

int index_of_attr = -1;
ret = vcsag_get_attr_value((wchar_t**)new_args, I,"PathName",
    &pathName, index_of_attr, 1);
if(ret !=VCSAG_SUCCESS)
{
    goto exit;
}

hFile = CreateFile(pathName,
GENERIC_READ | GENERIC_WRITE,0, NULL,
    OPEN_ALWAYS,
    res_ATTRIBUTE_NORMAL,(HANDLE)NULL);

if (!hFile || hFile == INVALID_HANDLE_VALUE)
{
    VCSAG_LOG_MSG(VCS_ERROR, 1002, VCS_DEFAULT_FLAGS,
        I,"Unable to create the file"
);
}
else
{
    VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,I,"Online
        successful.");
    CloseHandle(hFile);
}

exit:
if(new_args)
    VCSAgDelEncodedArgList((void**)new_args);
return 0;
}
```

3 Modify `res_offline()`:

```
// Function:    res_offline
// Description: This function deletes the file //

unsigned int res_offline(const char *res_name, void **attr_val)
{
    WCHAR      **new_args = NULL;
    DWORD      ret;
    wchar_t    *pathName = NULL;

VCSAG_LOG_INIT("res_offline");
VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,I,"Inside
    offline.");
ret = VCSAgGetEncodedArgList(VCSAgUTF8, attr_val, VCSAgUCS2,
(void ***)&new_args);

if (ret)
{
    VCSAG_LOG_MSG(VCS_NOTICE, 1001, VCS_DEFAULT_FLAGS,
        I,"Unable to get the arguments");
    goto exit;
}

int index_of_attr = -1;
ret = vcsag_get_attr_value((wchar_t**)new_args, I,"PathName",
&pathName, index_of_attr, 1);
if(ret !=VCSAG_SUCCESS)
{
    goto exit;
}

if (!DeleteFile(pathName))
{
    VCSAG_LOG_MSG(VCS_ERROR, 1003, VCS_DEFAULT_FLAGS,
                    I,"Unable to delete the file");
}
else
{
    VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,I,"offline
        successful.");
}

exit:
if(new_args)
    VCSAgDelEncodedArgList((void**)new_args);
return 0;
}
```

4 Modify the `res_monitor()`, funciton.
See “Example: Using C++ and script entry points on UNIX” on page 128.

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

```
make
```

6 Create the directory for the agent binaries:

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

7 Install the FileOnOff agent.

```
make install AGENT=FileOnOff
```

Example: Using C++ and script entry points on UNIX

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 script versions of `online` and `offline` entry points. This example implements the `VCSAgStartup`, `online`, `offline`, and `monitor` entry points only.

Example: implementing agent using VCSAgStartup, C++, and script entry points

1 Create a directory for the agent:

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

2 Copy the contents from the sample agent 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:

```
// Description: This functions registers the entry points //
void VCSAgStartup()
    {
    VCSAG_LOG_INIT("VCSAgStartup");

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);
}
```

5 Modify the `res_monitor()` function:

```
// Function:    res_monitor
// Description: Determine if the given file is online (file exists)
//              or offline (file does not exist).

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

```
{
    VCSAgResStatestate;
    WCHAR      **new_args = NULL;
    DWORD      ret, attrs;
    wchar_t    *pathName = NULL;

    VCSAG_LOG_INIT("res_monitor");
    VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,I,"Inside
        monitor.");
    ret = VCSAgGetEncodedArgList(VCSAgUTF8, attr_val, VCSAgUCS2,
    (void ***)&new_args);
    if (ret)
    {
            VCSAG_LOG_MSG(VCS_NOTICE, 1001, VCS_DEFAULT_FLAGS,
                    I,"Unable to get the arguments");
        state = VCSAgResUnknown;
        goto exit;
    }

    state = VCSAgResUnknown;
    int index_of_attr = -1;
    ret = vcsag_get_attr_value((wchar_t**)new_args,
I,"PathName",
        &pathName, index_of_attr, 1);

    if(ret !=VCSAG_SUCCESS)
        {
        goto exit;
    }
attrs = GetFileAttributes(pathName);
if (attrs == 0xffffffff)
{
    state = VCSAgResOffline;
    *conf_level = 0;
}
else
{
    state = VCSAgResOnline;
    *conf_level = 100;
}

VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,I,"Monitor
    successful.");
exit:
if(new_args)
    VCSAgDelEncodedArgList((void**)new_args);
return state;
}
```

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

```
make
```

7 Build the script entry points for the agent.
See “Example: Using script entry points on UNIX” on page 122

8 Create a directory for the agent:
```
mkdir /opt/VRTSvcs/bin/FileOnOff
```

9 Install the `FileOnOff` agent.
You have created a FileOnOff type definition file.
See “Creating the type definition file for a custom agent” on page 119.

Installing the custom agent

You can install the custom agent in one of the following directories.

On UNIX:

- A user-defined directory. For example /myagents/custom_type/. Note that you must configure the AgentDirectory attribute for this option.
- /opt/VRTSvcs/bin/custom_type/

Note: Create the custom_type directory at only one of these locations.

Add the agent binary and the script entry points to the custom_type directory.

Note: To package the agent, see the documentation for the operating system. When setting up the Solaris pkginfo file for the installation of agents that are to run in zones, set the following variable: SUNW_PKG_ALLZONES=true.

Defining resources for the custom resource type

When you have created a type definition for the resource and created an agent for it, you can begin to use the agent to control specific resources by adding the resources of the custom type and assigning values to resource attributes.

You can add resources and configure attribute values in the main.cf file.

Sample resource definition

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

Snippet from sample main.cf on UNIX:

```
include types.cf
.
.
.

    FileOnOff temp_file1 (
    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 name of the resource, `temp_file1`
- The name of the attribute, `PathName`
- The value for the `PathName` attribute:
  `On UNIX:"/tmp/test"`

When the resource temp_file1 is brought online on a system by VCS, the FileOnOff agent creates a file “test” in the specified directory on that system.

How the FileOnOff agent uses configuration information

The information in the VCS configuration is passed by the engine to the FileOnOff agent when the agent starts up on a node in the server farm. The information passed to the agent includes: the names of the resources of the type FileOnOff configured on the system, 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` to the entry point. The entry point performs its tasks.

Chapter 7

Testing agents

- About testing agents
- Using debug messages
- Using the engine process to test agents
- Using the AgentServer utility to test agents

About testing agents

Before testing an agent, make sure you have built the agent and have installed and configured the agent.

Using debug messages

You can activate agent framework debug messages by setting the value of the LogDbg attribute. This directs the framework to print messages logged with the specified severity.

See “LogDbg” on page 148.

Debugging agent functions (entry points).

The LogDbg attribute indicates the debug severities enabled for the resource type or agent framework. Debug severities used by agent functions are in the range of DBG_1–DBG_21.

To enable debug logging, use the LogDbg attribute at the type-level.

To set debug severities for a particular resource-type:

```
hatype -modify <resource-type> LogDbg -add <Debug-Severity>
[<Debug-Severity> ...]
```

To remove debug severities for a particular resource-type:

```
hatype -modify <resource-type> LogDbg -delete <Debug-Severity>
[<Debug-Severity> ...]
```

To remove all debug severities:

```
hatype -modify <resource-type> LogDbg -delete -keys
```

Note that you cannot set debug severities for an individual resource.

To debug a specific agent's entry point, see the documentation for that agent. So for bundled agents, see the *Bundled Agent's Reference Guide*.

UNIX: $VCS_LOG/log/*resource_type*_A.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` commands:

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

The debug messages from the FileOnOff agent with debug severities DBG_3 and DBG_4 get printed to the log files. Debug messages from C++ entry points get printed to the agent log file and from script entry points will get printed to the HAD log file. An example line from the agent log file:

.
.

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

Debugging the agent framework

The LogDbg attribute indicates the debug severities enabled for the resource type or agent framework. The debug messages from the agent framework are logged with the following severities:

- DBG_AGDEBUG: Enables most debug logs, which include: debugging commands received from the engine, service thread execution code path, that is when a service thread picks up a resource for running an entry point or for modification of an attribute, printing of environment variables that the agent uses, timer-related processing like sending IAmAlive messages to engine, and so on.
- DBG_AGINFO: Enables debugging messages related to specific entry-point execution, including entry point exit codes, transitioning of resources between various internal-states, printing of ArgListValues before entry point invocation, values of entry-point execution related attributes like RunInContainer and PassCInfo, and so on.
- DBG_AGTRACE: Enables verbose debug logging, the bulk of it being Begin and End messages for almost every function that gets called within the agent-framework, like function-tracing.

Using the engine process to test agents

When the VCS HAD process becomes active on a system, it automatically starts the appropriate agent processes based on the contents of the configuration files.

A single agent process monitors all resources of the same type on a system.

After the VCS HAD 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 to use commands 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 to test agents

The `AgentServer` utility enables you to test agents without running the HAD process. The utility is part of the product package and is installed in the directory

UNIX: $VCS_HOME/bin.

Run the AgentServer utility on any node when the engine process is not running.

To start the AgentServer and access help (UNIX)

1. Type the following command to start `AgentServer`:
   `$`VCS_HOME`/bin/AgentServer`
   The `AgentServer` utility monitors a TCP port for messages from the agents. Configure this port by setting the following string in the /etc/services file to the port number.
   vcstest <port number>/tcp
   If you do not specify this value, AgentServer uses the default value 14142.
2. When AgentServer is started, a message prompts you to enter a command/ For a complete list of the AgentServer commands, type:
   `> 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
startagent
stopagent
quit
```

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 (UNIX)

1 Start the agent for the resource type:

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

You receive the following messages:

```
Agent (FileOnOff) has connected.
Agent (FileOnOff) is ready to accept commands.
```

2 The following are examples of types.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"
)
```

The sample configuration is set up using AgentServer commands.

3 Complete the following steps to pass this sample configuration to the agent.

- Add a type:

```
>addtype FileOnOff FileOnOff
```

- Add attributes of the type:

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

- Add the static attributes to the FileOnOff resource type:

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

- Add the LogLevel attribute to see the debug messages from the agent:

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

- Add a resource:

```
>addres FileOnOff file1 FileOnOff
```

- 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
```

- 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
```

- To take a resource offline:

```
>offlineres FileOnOff file1
```

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

5 View the list of 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
```

Chapter 8

Static type attributes

- About static attributes
- Static type attribute definitions

About static attributes

Predefined static resource type attributes apply to all resource types.

See “Static type attribute definitions” on page 143

When developers create agents and define the resource type definitions for them, the static type attributes become part of the type definition.

Overriding static type attributes

Typically, the value of a static attribute of a resource type applies to all resources of the type. You can override the value of a static attribute for a specific resource without affecting the value of that attribute for other resources of that type. In this chapter, the description of each agent attribute indicates whether the attribute’s values can be overridden.

Users can override the values of static attributes two ways:

- By explicitly defining the attribute in a resource definition
- 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. You can remove the overridden values of static attributes by using the `hares -undo_override` option from the command line.

See the *Administrator’s Guide* for additional information about overriding the values of static attributes.

Note: You cannot override static type attributes for agents that run in virtual machines.

See the *Administrator’s Guide for more information*.

Static type attribute definitions

The following sections describe the static attributes for agents.

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, if overridden.

Whether overridden or not, no matter what value is specified for ActionTimeout, the value is internally limited to 0.5 * MonitorInterval. MonitorInterval attribute description is given below.

The default is 30 seconds. The ActionTimeout attribute value can be overridden.

AEPTimeout

The AEPTimeout (Append Entry Point Timeout) attribute is a boolean attribute. Set this attribute to true to append the timeout value for a particular entry point to the list of arguments passed to the entry point.

For example, if you set AEPTimeout to 1, the agent framework passes the value of the MonitorTimeout attribute to the monitor entry point in the name-value tuple format. Likewise, the value of CleanTimeout gets passed to the clean entry point.

This feature does not apply to pre-V50 agents.

See “About the entry point timeouts” on page 44.

AgentClass

Indicates the scheduling class for agent process.

Default is “TS”.

AgentFailedOn

A keylist attribute indicating the systems on which the agent has failed. This is not a user defined attribute.

Default is an empty keylist.

AgentPriority

Indicates the priority in which the agent process runs.

Default is 0.

AgentReplyTimeout

The HAD process restarts an agent if it has not received *any* messages from the agent for the number of seconds specified by AgentReplyTimeout.

The default value of 130 seconds works well for most configurations. Increase this value if the HAD process is restarting the agent too often during steady state of the server farm. This may occur when the system is heavily loaded or if the number of resources exceeds four hundred. Refer to the description of the command `haagent -display`. Note that the HAD process will also restart a crashed agent.

The AgentReplyTimeout attribute value *cannot* be overridden.

AgentStartTimeout

The value of AgentStartTimeout specifies how long the HAD process waits for the initial agent “handshake” after starting the agent, before attempting to restart it.

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, info, action, and clean entry points.

The default is an 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’s attribute is 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, say, there is a type T1 whose ArgList is of the form:

```
{ Attr1, Attr2, Attr3:Attr_A }
```

where Attr1, Attr2 and Attr3 are attributes of type T1, and say for a resource res1T1 of type T1, Attr3 's value is the name of another resource, res1T2. Then the entry points for res1T1 are passed the values of attributes Attr1 and Attr2 of res1T1 and the value of attribute Attr_A of resource res1T2.

Note that one has to first add the attribute Attr3 to type T1 before adding Attr3:Attr_A to T1's ArgList. Only then should one modify Attr3 for a resource (res1T1) to reference another resource (res1T2). Also, the value of Attr3 can

either be another resource of the same time (res2T1) or a resource of a different type (res1T2).

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.

ContainerOpts

Specifies whether the service group's load dimensions need to be passed to the resource to initialize and set CPU shares for the resource. It also specifies whether the resource needs to run in the context of a container. For each application or resource type that you want to include as part of the zone or project, you need to assign the following values to the ContainerOpts attribute:

- RunInContainer (RIC)
  RunInContainer defines whether the agent framework should run all the script-based entry points for the agent inside the container. If the attribute is set to 1, all script-based entry points are forked off inside the local container that is configured for the corresponding resource. If the attribute is set to 0, even if a resource's service group has ContainerInfo set, the entry point scripts for that resource will still be run in the global container.
- PassCInfo (PCI)
  PassCInfo specifies if you want to pass the container information, defined in the service group's ContainerInfo attribute, to the resource. Specify a value of 1, if you want to pass the container information to the resource. Specify a value of 0, if you do not want to pass the container information to the resource.
- PassLoadInfo (PLI)

PassLoadInfo specifies if you want to pass the service groups load dimensions, defined in the service group's Load attribute, to the resource. The service groups load dimensions are set in the resource's Workload attribute. Specify a value of 1, if you want to pass the service groups load dimensions to the resource. Specify a value of 0, if you do not want to pass the service groups load dimensions to the resource.

Each resource type also contains the Workload attribute, which is a non-static attribute. The value for this attribute is populated from the service group's Load attribute, when the resource's PassLoadInfo (ContainerOpts attribute) key set to 1.

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 HAD from restarting the resource indefinitely. The HAD process 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.

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

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.

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 being brought online 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. The FireDrill attribute cannot be overridden.

Refer to the *Administrator’s Guide* for details of how to set up and implement a fire drill.

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 how long the agent framework allows for completion of the `info` entry point.

The default is 30 seconds. The value of the InfoTimeout attribute is internally capped at MonitorInterval / 2. The InfoTimeout attribute value can be overridden.

IntentionalOffline

Defines how VCS reacts to a configured application being intentionally stopped outside of VCS control.

Add this attribute for agents that support detection of an intentional offline outside of VCS control.

Note that the intentional offline feature is available for agents registered as V51 or later. Use Script51Agent to use this feature.

The value 0 instructs the agent to register a fault and initiate the failover of a service group when the supported resource is taken offline outside of VCS control. The default value for this attribute is 0.

The value 1 instructs VCS to take the resource offline when the corresponding application is stopped outside of VCS control. This attribute does not affect VCS behavior on application failure. VCS continues to fault resources if managed corresponding applications fail.

See “About on-off, on-only, and persistent resources” on page 18.

Leveltwomonitorfrequency

The number of monitor cycles at which the agent framework initiates detailed monitoring. For example, if you set this attribute to 5, the agent framework initiates detailed monitoring every five monitor cycles.

LogDbg

The LogDbg attribute indicates the debug severities enabled for the resource type or agent framework.

Debug severities used by agent functions are in the range of DBG_1–DBG_21. 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, to specify the debug severities that they want to enable, which would cause those debug messages to be printed to the log files.

The debug messages from the agent framework are logged with the following severities:

- DBG_AGDEBUG: Enables most debug logs.
- DBG_AGINFO: Enables debugging messages related to specific entry-point execution.
- DBG_AGTRACE: Enables verbose debug logging.

See "Using debug messages" on page 134.

See "About logging agent messages" on page 98 for information on the APIs available to log debug messages from agent entry points. These APIs expect a debug severity as a parameter, along with the message to be logged. You can choose different debug severities for messages to provide different logging levels for the agent. When you enable a a particular severity in the LogDbg attribute, agent entry points log corresponding messages.

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 LogFileSize 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 LogFileSize
```

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

1 The `offline` entry point did not complete within the expected time. 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 resource or a resource in transition.

Default is 60 seconds. The MonitorInterval attribute value can be overridden.

MonitorStatsParam

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

```
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 HAD. The value of this key can be from 1 to 30. A value of 0 (zero) indicates that the average monitor time need not be computed. This is the default value for this key.
- *ExpectedValue*: The expected monitor time in milliseconds for all resources of this type. Default=100.
- *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.

For more information:

Refer to the *Administrator's Guide*.

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 a suitable value for the MonitorTimeout attribute can be assisted by the use of the MonitorStatsParam attribute.

NumThreads

NumThreads specifies the maximum number of service threads that an agent is allowed to create. Service threads are the threads in the agent that service resource commands, the main one being entry point processing. NumThreads does not control the number of threads used for other internal purposes.

Agents dynamically create service threads depending on the number of resources that the agent has to manage. Until the number of resources is less than the NumThreads value, the addition of a new resource will make the agent create an additional service thread. Also, if the number of resources falls below the NumThreads value as a result of deletion of resources, the agent will correspondingly delete service threads. Since an agent for a type will be started by VCS HAD process only if there is at least one resource for that type in the configuration, an agent will always have at least 1 service thread. Setting NumThreads to 1 will thus prevent any additional service threads from being created even if more resources are added.

The maximum value that can be set for NumThreads is 30.

Default is 10. The NumThreads attribute *cannot* be overridden.

OfflineMonitorInterval

The duration (in seconds) between two consecutive monitor calls for an OFFLINE resource. If set to 0, OFFLINE resources are not monitored.

Default is 0 seconds. The OfflineMonitorInterval attribute value can be overridden.

Note: Since the default value of this attribute is 0, concurrency violations are not detected. If an application that is supposed to be offline on a node is brought online outside of VCS control, the application continues to run since VCS cannot detect this state change.Data is protected using I/O fencing. As mentioned, to avoid this, one can set OfflineMonitorInterval to a non-zero value (apart from overriding it for a specific resource).

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 bring a resource online 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 is brought 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 is brought 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 HAD that the online attempt 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 for the resources of the type. The values are OnOff (can be brought online and taken offline), OnOnly (can be online only), and None (cannot be brought online or taken offline).

Default is OnOff. The Operations attribute value *cannot* be overridden.

RegList

RegList is a type level keylist attribute 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.

RestartLimit

Affects how the agent responds to a resource fault.

Refer also to:

"FaultOnMonitorTimeouts" on page 147

"ToleranceLimit" on page 156.

A non-zero value for RestartLimit causes the invocation of the `online` entry point instead of the failover of the service group to another system. The HAD process attempts to restart the resource according to the number set in RestartLimit before it gives up and attempts failover. However, if the resource remains online for the interval designated in ConfInterval, earlier attempts to restart are not counted against RestartLimit.

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.

Default is 0. The RestartLimit attribute value can be overridden.

ScriptClass

Indicates the scheduling class of the script processes (for example, online) created by the agent.

Default is "TS".

ScriptPriority

Indicates the priority of the script processes created by the agent.

Default is 0.

SupportedActions

The SupportedActions (string-keylist) attribute lists all possible actions defined for an agent, including those defined by the agent developer. The HAD process validates the *action_token* value specified in the `hares -action` *`resource action_token`* command against the SupportedActions attribute. For example, if action_token is not present in SupportedActions, HAD will not allow the command to go through. 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. This attribute serves as a reference for users of the command line or the graphical user interface.

See "About the action entry point" on page 33.

An example definition of a resource type in a VCS *`ResourceType`*`Types.cf` file 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, `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.

Chapter 9

State transition diagrams

- State transitions
- State transitions with respect to ManageFaults attribute

State transitions

This section describes state transitions for:

- Opening a resource
- Resource in a steady state
- Bringing a resource online
- Taking a resource offline
- Resource fault (without automatic restart)
- Resource fault (with automatic restart)
- Monitoring of persistent resources
- Closing a resource

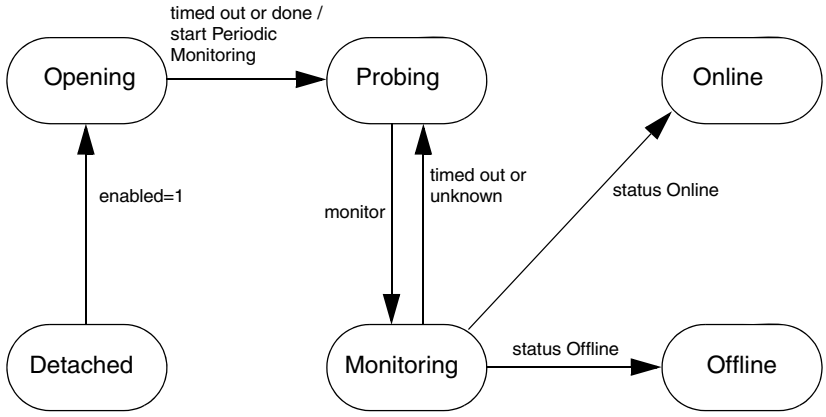
In addition, state transitions are shown for the handling of resources with respect to the `ManageFaults` service group attribute.

See “State transitions with respect to ManageFaults attribute” on page 167.

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 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.

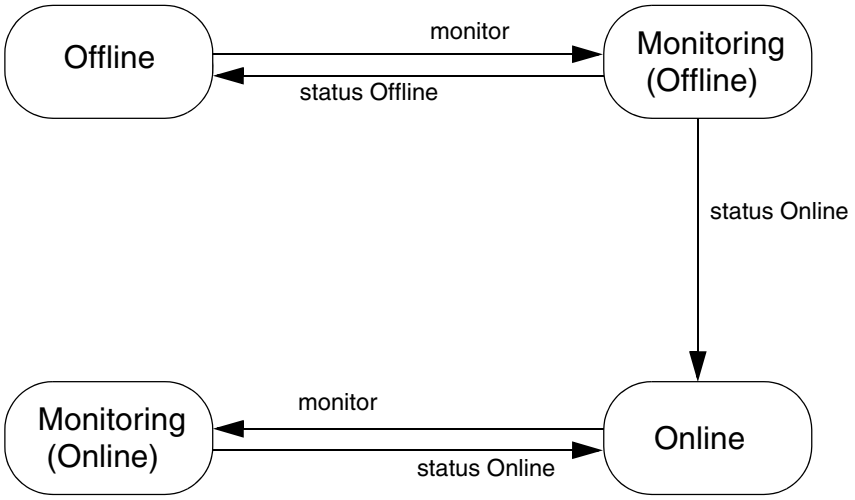
Opening a resource



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 bring a resource online or take it offline.

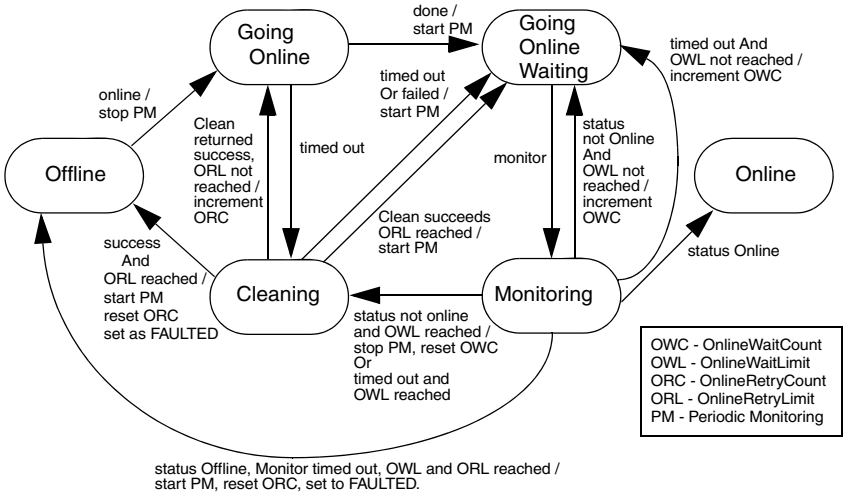
When the resource is enabled (Enabled=1), the `open` entry point is invoked. Periodic Monitoring begins after *`open`* times out or succeeds. Depending on the return value of *`monitor`*, the resource transitions to either the Online or the Offline state. In the unlikely event that *`monitor`* times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

Bringing a resource online: ManageFaults = ALL



When the agent receives a request from the engine to bring the resource online, 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.

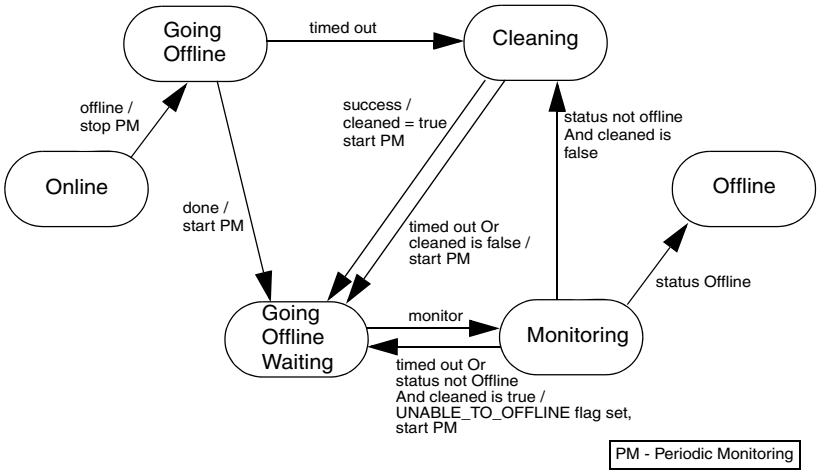
If *monitor* returns a status of online, the resource moves to the Online state.

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.

Taking a resource offline



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

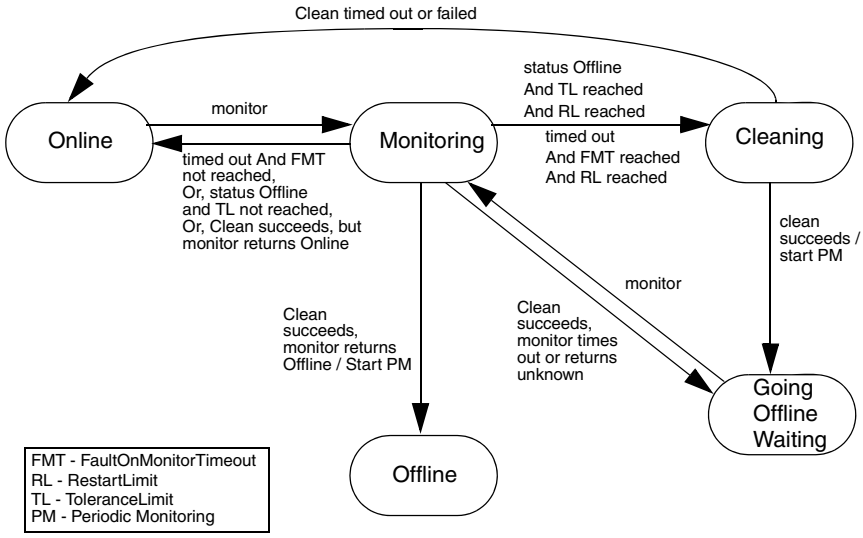
If *offline* succeeds, the resource enters the Going Offline Waiting state where it waits for the next monitor.

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



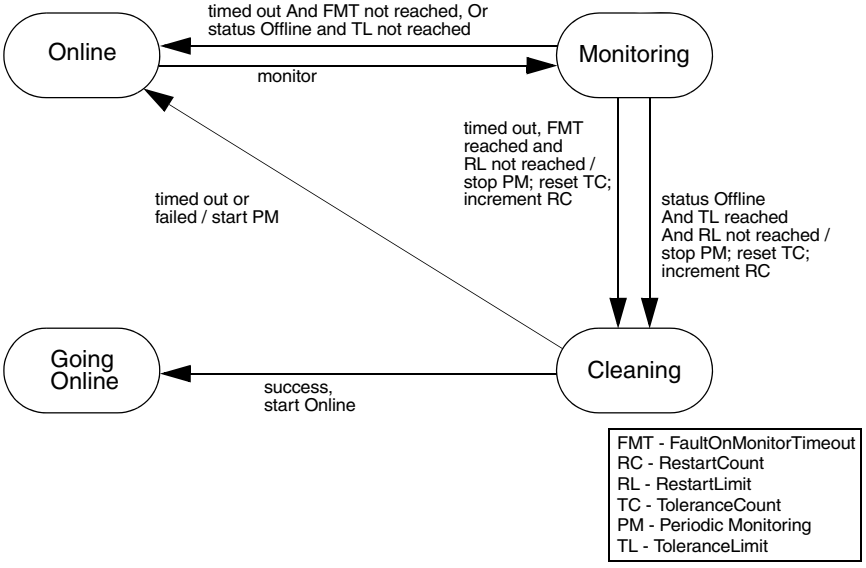
This diagram describes the activity that occurs when a resource faults and the `RestartLimit` is reached. When the *`monitor`* entry point times out successively and `FaultOnMonitorTimeout` is reached, or monitor returns offline and the `ToleranceLimit` is reached, the agent invokes the *`clean`* entry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

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

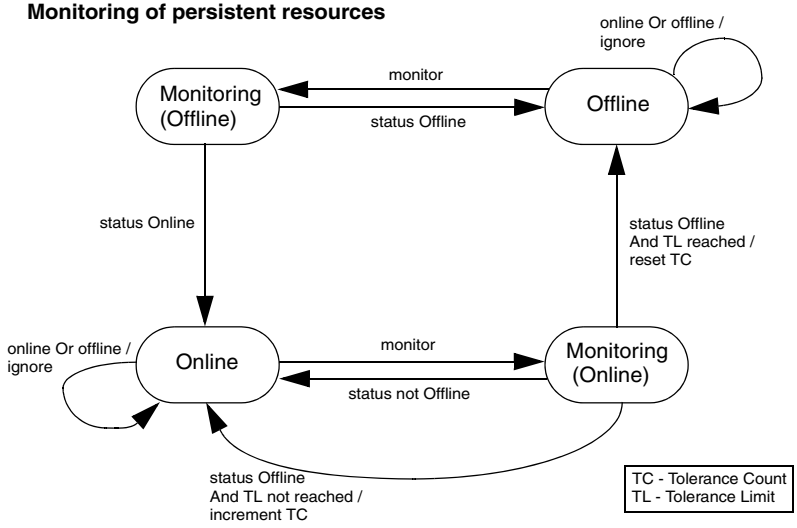


This diagram describes the activity that occurs when a resource faults and the `RestartLimit` is not reached. When the *`monitor`* entry point times out successively and `FaultOnMonitorTimeout` is reached, or `monitor` returns offline and the `ToleranceLimit` is reached, the agent invokes the *`clean`* entry point.

- 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, "Bringing a resource online."
- If *`clean`* fails or times out, the agent places the resource in the Online state as if no fault occurred.

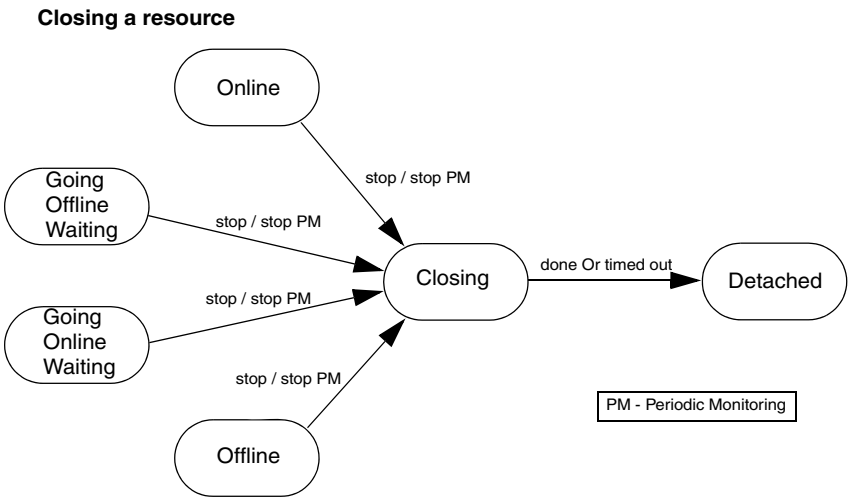
Refer to the diagram "Resource fault without automatic restart," for a discussion of activity when a resource faults and the `RestartLimit` is reached.

Monitoring of persistent resources



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 *close* entry point is invoked. When the *close* entry point succeeds or times out, the resource is placed in the Detached state.

The next set of diagrams illustrate the following state transitions:

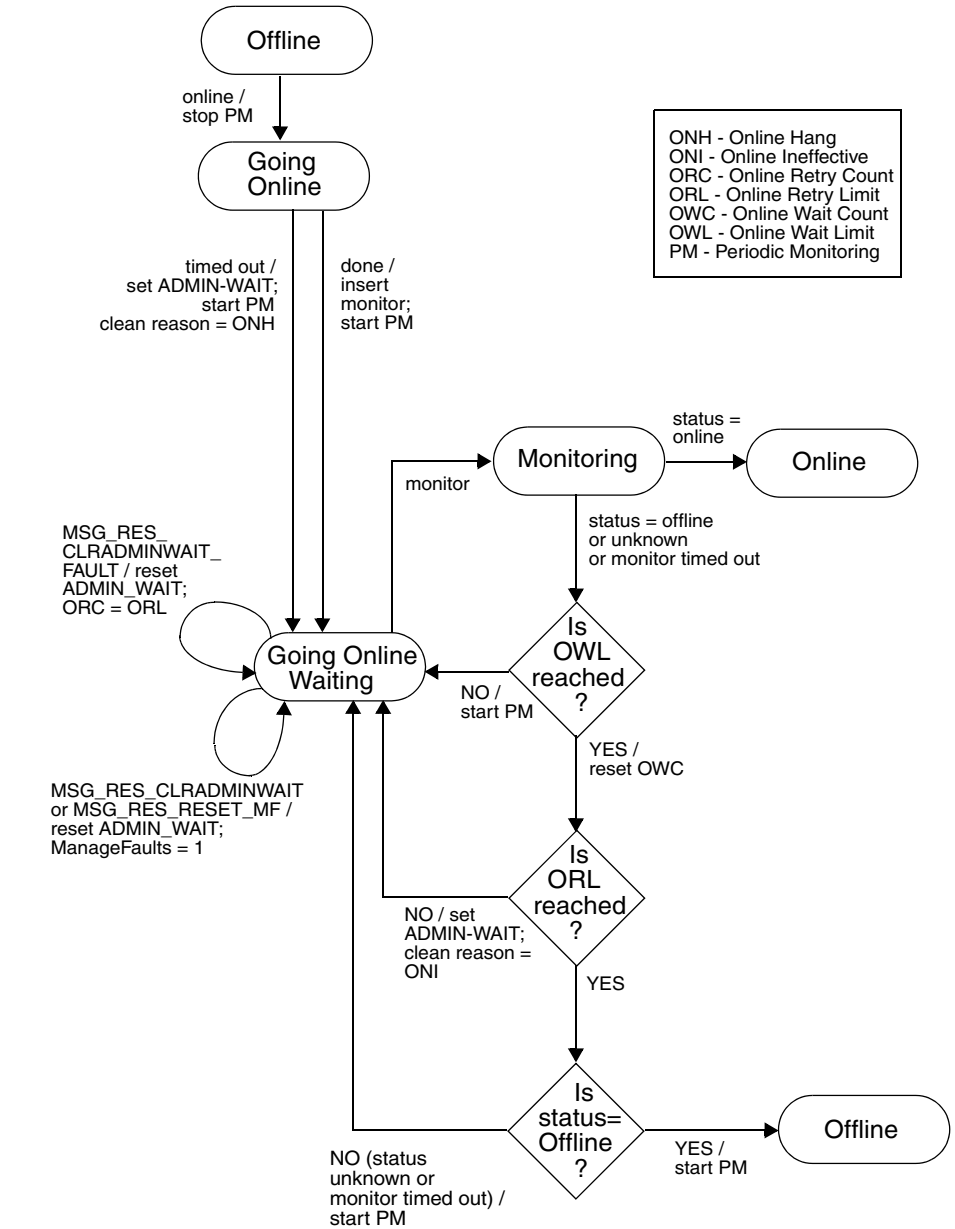
State transitions with respect to ManageFaults attribute

This section shows state transition diagrams with respect to the `ManageFault` attribute.

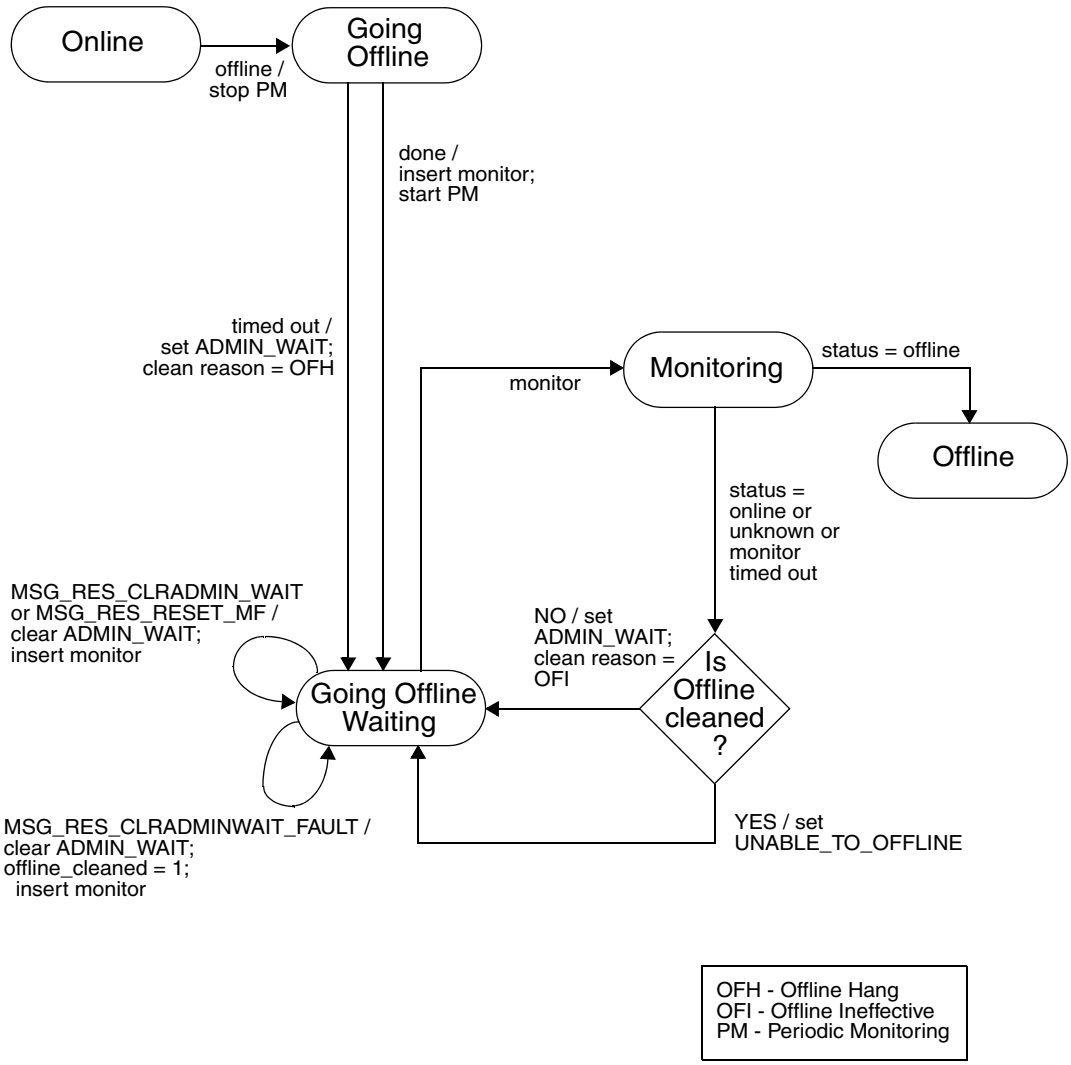
By default, `ManageFaults` is set to ALL, in which case the clean entry point is called by VCS. See “ManageFaults” on page 149.The diagrams cover the following conditions:

- Bringing a resource online when the `ManageFaults` attribute is set to NONE
- Taking a resource offline 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

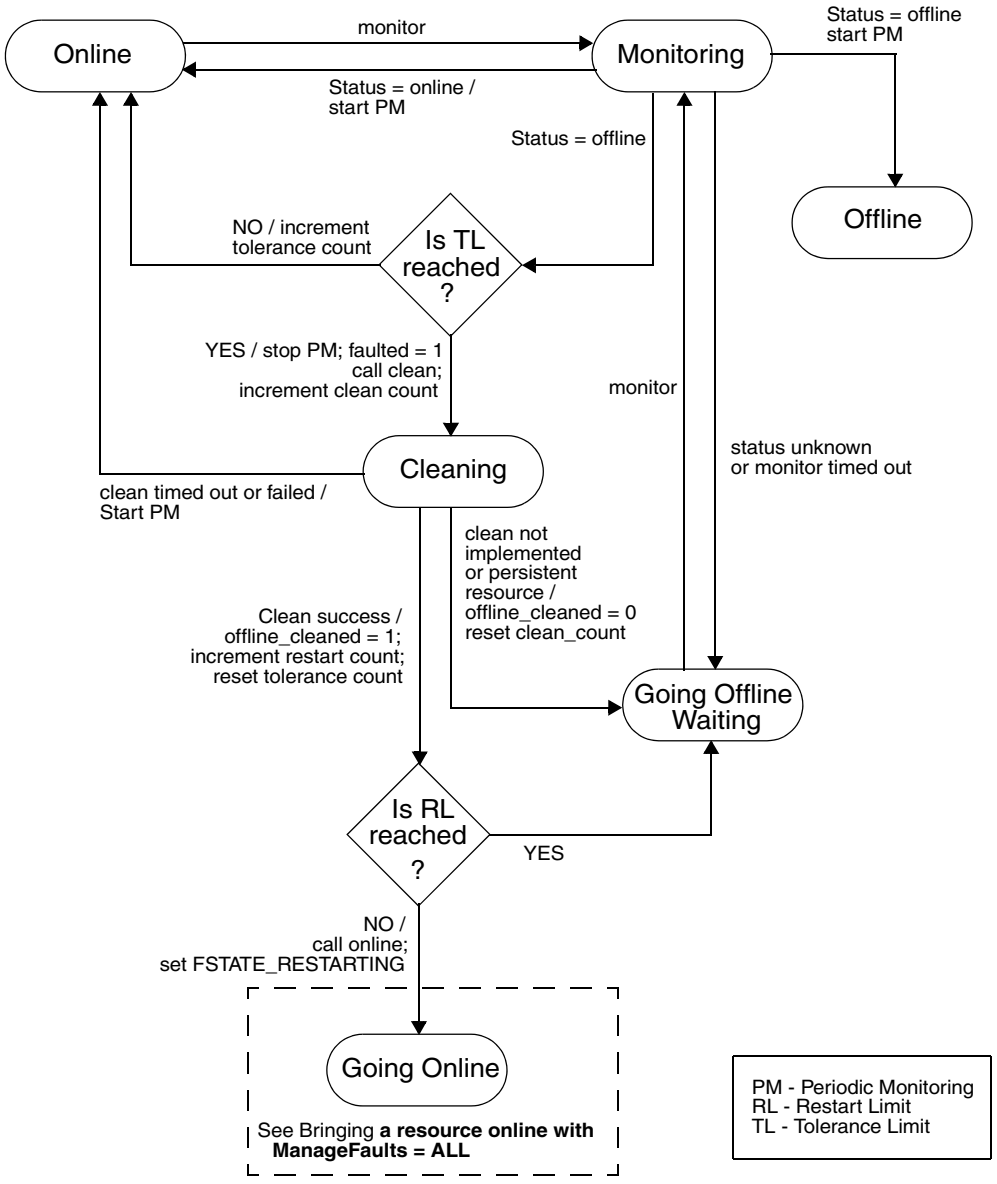
Bringing a resource online: ManageFaults attribute = NONE



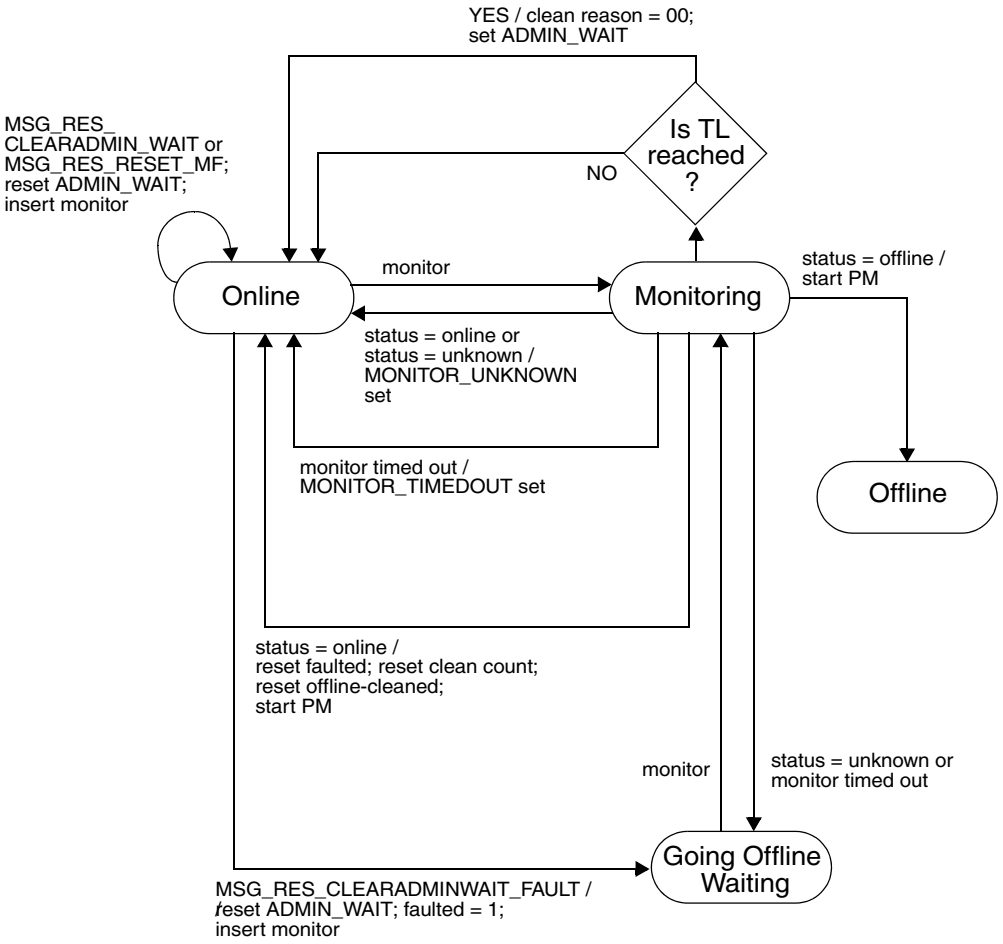
Taking a resource offline; ManageFaults = None



Resource fault: ManageFaults attribute = ALL

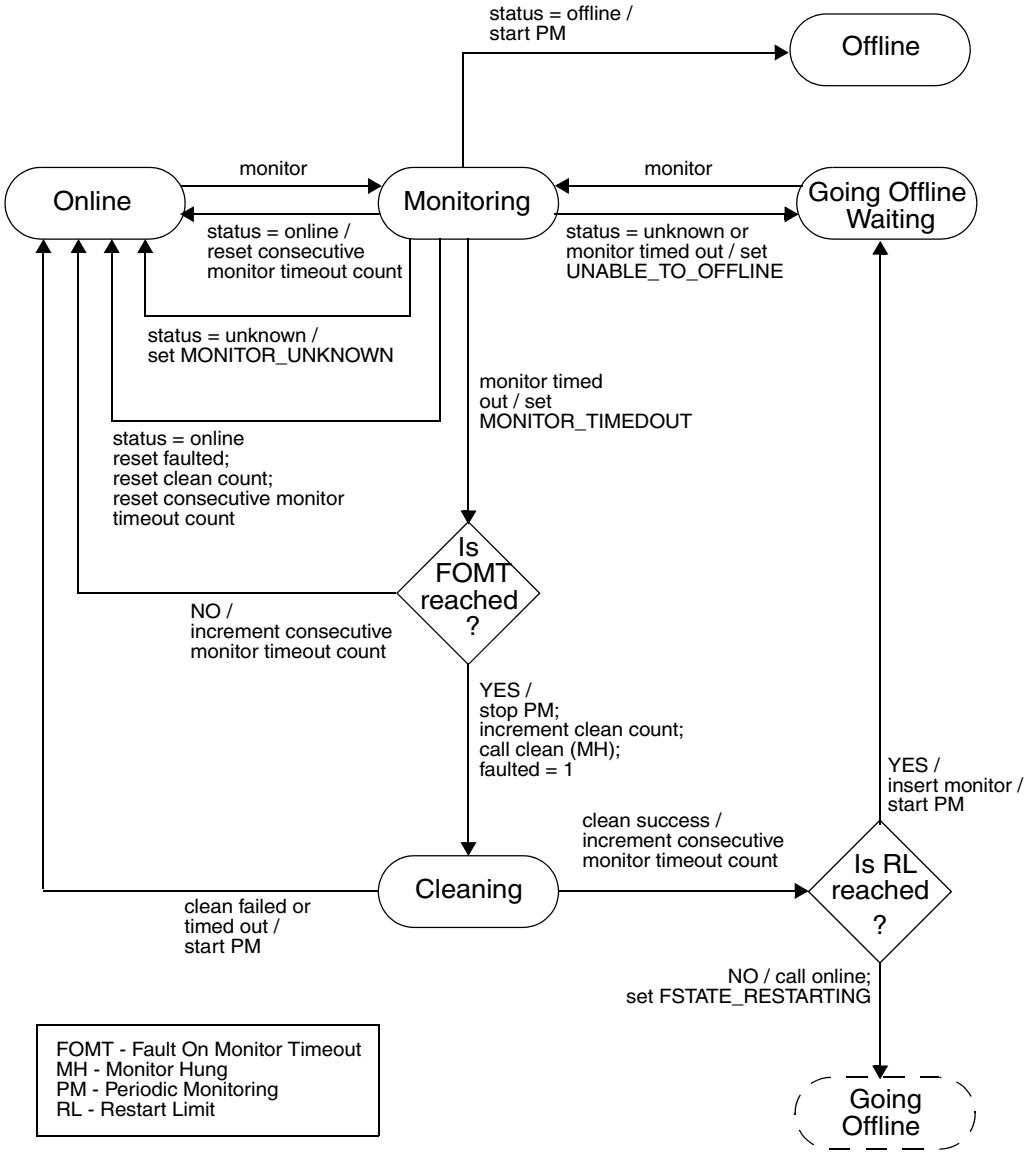


Resource fault (unexpected offline): ManageFaults attribute = NONE

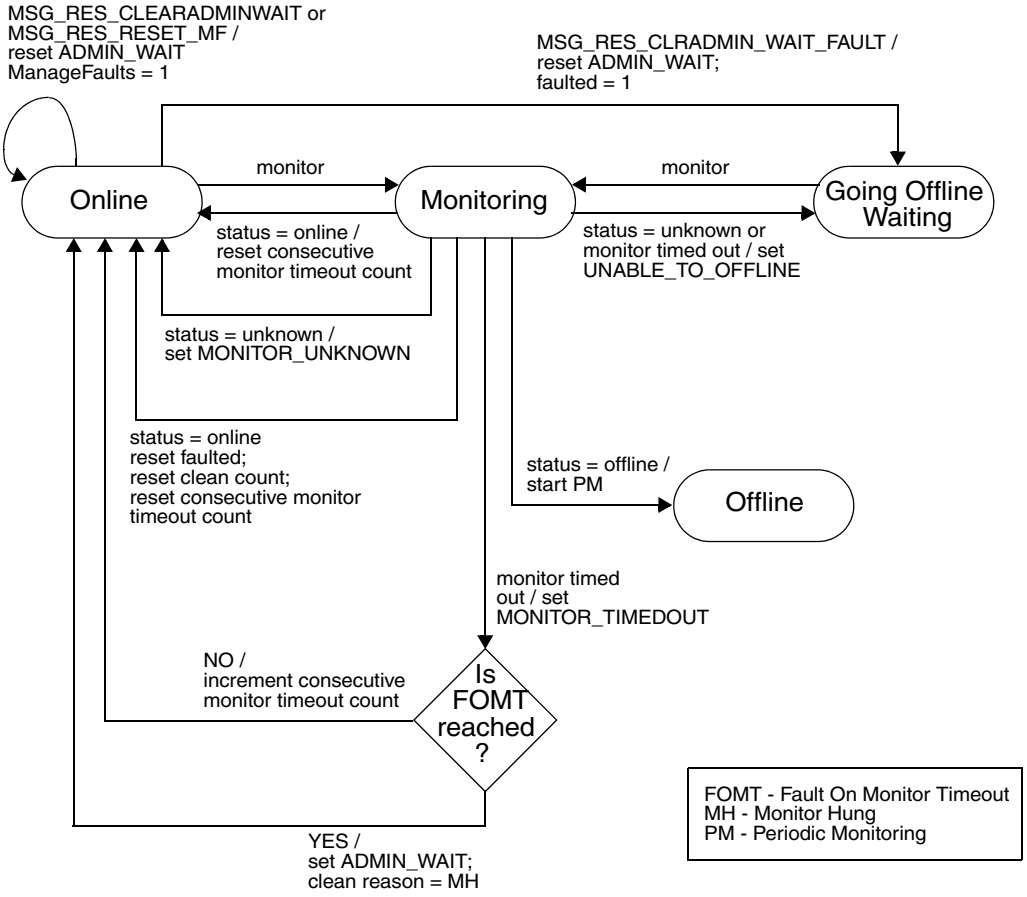


PM - Periodic Monitoring
TL - Tolerance Limit

Resource fault (monitor hung): ManageFaults attribute = ALL



Resource fault (monitor hung): ManageFaults attribute = NONE



Chapter 10

Internationalized messages

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 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:

`VRTSvcsSunAgent.smc`

The file is 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 177 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:

```
VRTSvcsLnxOracle.bmc
```

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 VRTSvcsLnxAgent.smc
```

The file `VRTSvcsLnxAgent.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/VRTS/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/VRTS/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/VRTS/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 VRTSvcsLnxAgent.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/VRTS/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 VRTSvcsSunLnxOracle.bmc /opt/VRTS/messages/en
   ```
2. Change to the directory containing the BMC file and run the `bmcmap` utility. For example:
   ```
   cd /opt/VRTS/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.bmc map 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) Symantec Corporation.
# ALL RIGHTS RESERVED.
#

Language=en

HAD=VRTSvcsHad VRTSvcsHPAgent VRTSvcsHPNewCustomAgent
```

```
# VCS
VRTSvcsHad.version=5.1
VRTSvcsHad.Category=1
VRTSvcsHad.IDstart=0
VRTSvcsHad.IDend=53502

# 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.
   - Make your changes, such as adding, deleting, or changing messages.
   - Change the minor number of the version number in the header. For example, change the version from 2.0 to 2.1.
   - 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.

Appendix A

Using pre-5.0 VCS agents

- Using pre-5.0 VCS agents and registering them as V51 or later
- Guidelines for using pre-VCS 4.0 Agents
- Log messages in pre-VCS 4.0 agents
- Pre-VCS 4.0 Message APIs

Using pre-5.0 VCS agents and registering them as V51 or later

With VCS 5.0, the agent framework has been enhanced. Agents you develop to use this framework are registered as V50 or V51 agents. The framework enables you to use pre-5.0 agents and register them as V40 agents. The following sections describe how to use pre-5.0 agents with the VCS 5.0 agent framework.

When you use pre-5.0 agents with VCS, you may register them as V51 agents after making necessary modifications. Making this conversion affords you advantages, which include:

- You can use different versions of an agent on different systems in VCS.
- You can make changes to the resource type definition used on some systems without affecting how older versions of the agents function

Outline of steps to change V40 agents V51

- Modifications to PATH variables and links to the VCS Script51Agent binary (on UNIX) may be necessary.
- Change the way attributes and their values are passed to the entry points from the V40 format to V51 name-value tuple format.
- Include /opt/VRTSvcs/lib in path for Perl and shell to source them.

- Set necessary environment variables.

Name-value tuple format for agents registered as V51 or later

VCS pre-5.0 agents required that the arguments passed to the entry point to be in the order indicated by the ArgList attribute as it was defined in the resource type. The order of parsing the arguments was determined by their position the definition.

With the V51 agent framework, agents can use entry points that can be passed attributes and their values i a format of name-value tuples. Such a format means that attributes and their values are parsed by the name of the attribute and not by their position in the ArgList Attribute.

The general tuple format for V51 attributes in the ArgList is:

<name> <number_of_elements_in_value> <value>

Scalar attribute format

For *scalar* attributes, whether string, integer, or boolean, the formatting is:

<attribute_name> 1 <value>

Example is:

DiskGroupName 1 mydg

Vector attribute format

For *vector* attributes, whether string or integer, the formatting is:

<attribute_name> <number_of_values_in_vector> <values_in_vector>

Examples are:

MyVector 3 aa cc dd

MyEmptyVector 0

Keylist attribute format

For string *keylist* attributes, the formatting is:

<attribute_name> <number_of_keys_in_keylist> <keys>

Examples are:

DiskAttr 4 hdisk3 hdisk4 hdisk5 hdisk6

DiskAttr 0

Association attribute format

For association attributes, whether string or integer, the formatting is:

<attribute_name> <number_of_keys_and_values> <values_of_keylist>

Examples are:

MyAssoc 4 key1 val1 key2 val2

MyAssoc 0

Example script in V40 and V51

Note the following comparison.

V40

```
ResName=$1
Attr1=$2
Attr2=$3
VCSHOME="${VCS_HOME:-/opt/VRTSvcs}"
. $VCSHOME/bin/ag_i18n_inc.sh;
VCSAG_SET_ENVS $ResName;
```

V51

```
ResName=$1; shift;
."../ag_i18n_inc.sh";
VCSAG_SET_ENVS $ResName;

VCSAG_GET_ATTR_VALUE "Attr1" -1 1 "$@";
attr1_value=${VCSAG_ATTR_VALUE};
VCSAG_GET_ATTR_VALUE "Attr2" -1 1 "$@";
attr2_value=${VCSAG_ATTR_VALUE};
```

Sourcing ag_i18n_inc modules in script entry points

In entry points, you need to source the ag_i18n_inc modules. The following examples assume that the agent is installed in the directory /opt/VRTSvcs/bin/*type*.

For entry points in Perl:

```
...
$ResName = shitf;
use ag_i18n_inc;
VCSAG_SET_ENVS ($ResName);
...
```

For entry points in Shell:

```
...
ResName = $1; shift;
. "../ag_i18n_inc.sh";
VCSAG_SET_ENVS $ ResName;
```

Guidelines for using pre-VCS 4.0 Agents

The agent framework 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.

Changes made to the agent framework with VCS 4.0 and VCS 5.0 releases affect how agents developed using the pre-VCS 4.0 agent framework can be used. While not necessary, all pre-VCS 4.0 agents may 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 4.0 and later ScriptAgent can be used on UNIX. 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 you *want* to implement the `action` or the `info` entry point:
  - Add the `action` or `info` entry point, C++ or script-based, to the agent.
  - Use the API VCSAgInitEntryPointStruct with the parameter V40 to register the agent as a VCS 4.0 agent. Use the VCSAgValidateAndSetEntryPoint API to register your C++ entry points.
  - 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 pre-4.0 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.

See Chapter 5, "Logging agent messages" on page 97.

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

This guide describes the logging macros for C++ agents and script-based agents.

See Chapter 5, “Logging agent messages” on page 97.

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

UNIX: $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

UNIX: $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

UNIX: $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

UNIX: $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

UNIX: $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
  script entry point 91
  supported actions 156
agent framework
  described 17
  library, C++ 37
  logging APIs 100
  multithreaded 50
  working with pre-4.0 agents 187
agent messages
  formatting 99
  normal in VCSAG_LOG_MSG 113
AgentClass parameter 143
AgentPriority parameter 143
AgentReplyTimeout parameter 144
ArgList parameter 144
ArgList reference attributes 144
association attribute dimension 21
attr_changed entry point 34
  C++ syntax 64
  script syntax 91
AttrChangedTimeout parameter 145
attribute dimensions
  association 21
  keylist 21
  scalar 20
  vector 20
attribute types
  boolean 20
  integer 20
  string 20
attributes
  local and global 21

B

binary message catalog (BMC) files
  converting from SMC files 179
  displaying contents 179
  updating 181
bmcgen utility 179
bmcread utility 179
boolean attribute type 20

C

category ID for messages 110
clean entry point 31
  C++ syntax 61
  script syntax 91
CleanTimeout parameter 145
close entry point 35
  C++ syntax 66
  script syntax 92
CloseTimeout parameter 145
configuration language
  local and global attributes 21
ConfInterval parameter 146

D

debug message severity level 102
debug messages
  C++ entry points 102
  Perl script entry points 114
  Shell script entry points 114

E

entry points
  attr_changed 34
  clean 31
  close 35
  definition 18
  info 28
  monitor 27
  offline 30
  online 30
  open 34
  sample structure 38
  shutdown 35
enum types for clean
  VCSAgCleanMonitorHung 32
  VCSAgCleanOfflineIneffective 31

VCSAgCleanOnlineHung 31
VCSAgCleanOnlineIneffective 31
VCSAgCleanUnexpectedOffline 31

F

FaultOnMonitorTimeouts parameter 147
FireDrill parameter 147
formatting agent messages 99

G

global attributes 21

I

info entry point 28
C++ syntax 55
script example 92
InfoTimeout parameter 148
initializing functions with VCSAG_LOG_INIT 104
integer attribute type 20
intentional offline 18

K

keylist attribute dimension 21

L

local attributes 21
log category 105
LogDbg parameter 148
LogFileSize parameter 149
logging APIs
C++ 100
script entry points 110

M

ManageFaults parameter 149
message text format 99, 100
mnemonic message field 99
monitor entry point 27
C++ syntax 54, 55
script syntax 90
MonitorLevel parameter 150
MonitorStatsParam parameter 150
MonitorTimeout parameter 151

O

offline entry point 30
C++ syntax 60
script syntax 90
OfflineMonitorInterval parameter 152
OfflineTimeout parameter 152
online entry point 30
C++ syntax 59
script syntax 90
OnlineRetryLimit parameter 153
OnlineTimeout parameter 153
OnlineWaitLimit parameter 153
OnOff resource type 18
OnOnly resource type 18
open entry point 34
C++ syntax 66
script syntax 92
OpenTimeout parameter 153
Operations parameter 153

P

parameters
AgentClass 143
AgentPriority 143
AgentReplyTimeout 144
ArgList 144
AttrChangedTimeout 145
CleanTimeout 145
CloseTimeout 145
ConfInterval 146
FaultOnMonitorTimeouts 147
FireDrill 147
InfoTimeout 148
LogDbg 148
LogFileSize 149
ManageFaults 149
MonitorLevel 150
MonitorStatsParam 150
MonitorTimeout 151
OfflineMonitorInterval 152
OfflineTimeout 152
OnlineRetryLimit 153
OnlineTimeout 153
OnlineWaitLimit 153
OpenTimeout 153
Operations 153
RegList 154
RestartLimit 155

ScriptClass 155
ScriptPriority 155
ToleranceLimit 156
persistent resource type 18
primitives
definition 69
VCSAgGetCookie 73
VCSAgLogI18NMsg 191, 193, 194
VCSAgLogI18NMsgEx 192
VCSAgLogMsg 190
VCSAgRegister 71
VCSAgRegisterEPStruct 69
VCSAgSetCookie 69
VCSAgSetCookie2 69
VCSAgSnprintf 74
VCSAgStrlcat 74
VCSAgUnregister 72

R

RegList parameter 154
resource
closing (state transition diagram) 166
fault (state transition diagram) 163, 164
monitoring (state transition diagram) 165
offlining (state transition diagram) 162
onlining (state transition diagram) 161
OnOff type 18
OnOnly type 18
opening (state transition diagram) 159
persistent type 18
steady state (state transition diagram) 160
RestartLimit parameter 155

S

scalar attribute dimension 20
ScriptAgent 120, 187
script-based logging functions 109
ScriptClass parameter 155
ScriptPriority parameter 155
severity macros 103
severity message field 99
shutdown entry point 35
C++ syntax 68
script syntax 92
source message catalog (SMC) files
converting to BMC files 176
creating 176
state transition diagram
closing a resource 166
monitoring persistent resources 165
offlining a resource 162
onlining a resource 161
opening a resource 159
resource fault with auto restart 164
resource fault, no auto restart 163
resource in steady state 160
string attribute type 20

T

timestamp message field 99
ToleranceLimit parameter 156

U

UMI (unique message identifier) 99

V

VCSAG_CONSOLE_LOG_MSG logging macro 100
VCSAG_LOG_INIT initializing function 104
VCSAG_LOG_MSG logging macro 100
VCSAG_LOG_MSG script logging function 109
VCSAG_LOGDBG_MSG logging macro 100
VCSAG_LOGDBG_MSG script logging function 109
VCSAG_RES_LOG_MSG logging macro 100
VCSAG_SET_ENVS script logging function 109
VCSAgGetCookie primitive 73
VCSAgLogI18NMsg primitive 191, 193, 194
VCSAgLogI18NMsgEx primitive 192
VCSAgLogMsg primitive 190
VCSAgRegister primitive 71
VCSAgRegisterEPStruct primitive 69
VCSAgSetCookie primitive 69
VCSAgSetCookie2 primitive 69
VCSAgSnprintf primitive 74
VCSAgStartup entry point, C++ syntax 53
VCSAgStrlcat primitive 74
VCSAgUnregister primitive 72
VCSAgValidateAndSetEntryPoint 81
vector attribute dimension 20