Veritas™ Cluster Server Agent Developer's Guide

Windows Server 2003, Windows Server 2008

5.1

Veritas Cluster Server
Agent Developer's Guide

Copyright © 2008 Symantec Corporation. All rights reserved.

Veritas Cluster Server 5.1

Symantec, the Symantec logo, Storage Foundation, are trademarks or registered trademarks of Symantec Corporation or its affiliates in the U.S. and other countries. Other names may be trademarks of their respective owners.

The product described in this document is distributed under licenses restricting its use, copying, distribution, and decompilation/reverse engineering. No part of this document may be reproduced in any form by any means without prior written authorization of Symantec Corporation and its licensors, if any.

THIS DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID, SYMANTEC CORPORATION SHALL NOT BE LIABLE FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING PERFORMANCE, OR USE OF THIS DOCUMENTATION. THE INFORMATION CONTAINED IN THIS DOCUMENTATION IS SUBJECT TO CHANGE WITHOUT NOTICE.

The Licensed Software and Documentation are deemed to be "commercial computer software" and "commercial computer software documentation" as defined in FAR Sections 12.212 and DFARS Section 227.7202.

Symantec Corporation
20330 Stevens Creek Blvd.
Cupertino, CA 95014
www.symantec.com

Third-party legal notices

Third-party software may be recommended, distributed, embedded, or bundled with this Symantec product. Such third-party software is licensed separately by its copyright holder. All third-party copyrights associated with this product are listed in the accompanying release notes.

Windows is a registered trademark of Microsoft Corporation.

Licensing and registration

See the *Veritas Storage Foundation and High Availability Solutions Installation and Upgrade* Guide for license installation instructions.

Technical support

For technical assistance, visit http://www.symantec.com/business/support/index.jsp and select phone or email support. Use the Knowledge Base search feature to access resources such as TechNotes, product alerts, software downloads, hardware compatibility lists, and our customer email notification service.

Contents

Chapter 1 Introduction

About VCS agents 12
How agents work 13
About the agent framework 13
Resource type definitions 13
Agent functions (entry points) 14
About on-off, on-only, and persistent resources 14
About attributes 15
Developing an agent: overview 19
Considerations for the application 19
Creating an agent: highlights 20

Chapter 2 Agent entry point overview

About agent entry points 22
Supported entry points 22
How the agent framework interacts with entry points 22
Agent entry points described 23
About the monitor entry point 23
About the info entry point 24
About the online entry point 26
About the offline entry point 27
About the clean entry point 28
About the action entry point 30
About the attr_changed entry point 31
About the open entry point 31
About the close entry point 32
About the shutdown entry point 32
Return values for entry points 33
Considerations for using C++ or script entry points 34
About the VCSAgStartup routine 34
Script agents 35
About the agent information file 36
Example agent information file (Windows) 36
Implementing the agent XML information file 39
About the ArgList Attribute 39

ArgList attribute for agents registered as V50 40
ArgList Attribute for agents registered as V40 and earlier 40

Chapter 3 Creating entry points in C++

About creating entry points in C++ 44
Entry point examples in this chapter 45
Data Structures 46
C++ entry point syntax 48
VCSAgStartup 48
monitor 49
info 50
online 56
offline 57
clean 58
action 59
attr_changed 61
open 63
close 64
shutdown 65
Agent framework primitives 66
VCSAgRegisterEPStruct 66
VCSAgSetCookie 67
VCSAgRegister 68
VCSAgUnregister 69
VCSAgGetCookie 70
VCSAgStrlcpy 71
VCSAgStrlcat 71
VCSAgSnprintf 71
VCSAgCloseFile 71
VCSAgDelString 71
VCSAgExec 72
VCSAgExecWithTimeout 73
VCSAgGenSnmpTrap 75
VCSAgSendTrap 75
VCSAgLockFile 75
VCSAgInitEntryPointStruct 76
VCSAgSetStackSize 76
VCSAgUnlockFile 77
VCSAgDisableCancellation 77
VCSAgRestoreCancellation 77
VCSAgSetEntryPoint 78
VCSAgSetLogCategory 78
VCSAgGetProductName 78

Agent framework primitives for Windows 79
C++ logging primitives for use by Windows agents 79
C++ primitives for ArgList conversion 81
C++ primitives for ArgList string manipulation 82
C++ primitives for allocating and freeing memory 83
Example: Windows-based primitives for the FileOnOff agent 83

Chapter 4 Creating entry points in scripts

About creating entry points in scripts 90
Rules for using script entry points 90
Parameters and values for script entry points 90
ArgList attributes 91
Examples 91
Script entry point syntax 92
monitor 92
online 92
offline 92
clean 93
action 93
attr_changed 94
info 94
open 95
close 95
shutdown 95
Example script entry points for the FileOnOff resource 96

Chapter 5 Logging agent messages

About logging agent messages 100
Logging in C++ and script-based entry points 100
Agent messages: format 101
C++ agent logging APIs 102
Agent application logging macros for C++ entry points 103
Agent debug logging macros for C++ entry points 104
Severity arguments for C++ macros 105
Initializing function_name using VCSAG_LOG_INIT 106
Log category 107
Examples of logging APIs used in a C++ agent 108
Script entry point logging functions 111
VCSAG_SET_ENVS 112
VCSAG_LOG_MSG 114
VCSAG_LOGDBG_MSG 115
Using the functions in scripts 116

Chapter 6 Building a custom agent

Files for use in agent development 118
Creating the type definition file for a custom agent 119
Example: FileOnOffTypes.cf 119
Requirements for creating the agentTypes.cf file 119
Adding the custom type definition to the configuration 119
Building a custom agent on Windows 120
Recommendations for building custom agents on Windows 120
Implementing entry points using C++ 121
Example: Using C++ entry points on Windows 121
Implementing entry points using scripts 124
Example: Using script entry points on Windows 124
Example: Using VCSAgStartup() and script entry points on Windows 125
Example: Using C++ and script entry points on Windows 126
Installing the custom agent 129
Defining resources for the custom resource type 130
Sample resource definition 130

Chapter 7 Testing agents

About testing agents 134
Using debug messages 134
Using the engine process to test agents 135
Test commands 135
Using the AgentServer utility to test agents 136

Chapter 8 Static type attributes

About static attributes 140
Overriding static type attributes 140
Static timeout attributes on Windows 141
Static type attribute definitions 142
AgentFailedOn 142
.ArgList 142
ConfInterval 143
FaultOnMonitorTimeouts 144
FireDrill 144
InfoInterval 144
InfoTimeout 145
LogDbg 145
LogFileSize 146
ManageFaults 147
MinCompatAgentVersion 147
MonitorInterval 148

MonitorStatsParam 148
NumThreads 149
OfflineMonitorInterval 149
OnlineRetryLimit 149
OnlineWaitLimit 150
Operations 150
RegList 151
RestartLimit 152
SupportedActions 153
ToleranceLimit 153
TypeDefinitionVersion 154

Chapter 9 State transition diagrams

State transitions 156
State transitions with respect to ManageFaults attribute 165

Chapter 10 Internationalized messages

Creating SMC files 174
SMC format 174
Example SMC file 174
Formatting SMC files 175
Naming SMC files, BMC files 175
Message examples 176
Using format specifiers 176
Converting SMC files to BMC files 177
Storing BMC files 177
Displaying the contents of BMC files 177
Using BMC Map Files 178
Location of BMC Map Files 178
Creating BMC Map Files 178
Example BMC Map File 178
Updating BMC Files 179

Appendix A Using pre-5.0 VCS agents

Using pre-5.0 VCS agents and registering them as V50 181
Outline of steps to change V40 agents V50 181
Overview of V50 name-value tuple format 182
Sourcing ag_i18n_inc modules in script entry points 184
Guidelines for using pre-VCS 4.0 Agents 185
Log messages in pre-VCS 4.0 agents 186
Mapping of log tags (pre-VCS 4.0) to log severities (VCS 4.0) 186

How Pre-VCS 4.0 Messages are Displayed by VCS 4.0 and Later 187
Comparing Pre-VCS 4.0 APIs and VCS 4.0 Logging Macros 187
Pre-VCS 4.0 Message APIs 188
VCSAgLogConsoleMsg 188
VCSAgLogI18NMsg 189
VCSAgLogI18NMsgEx 190
VCSAgLogI18NConsoleMsg 191
VCSAgLogI18NConsoleMsgEx 192

Index 193

Chapter 1

Introduction

- About VCS agents
- How agents work
- Developing an agent: overview

About VCS agents

Agents are programs that manage resources, such as a disk group or an IP address, within a cluster 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 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 bundled agents, including their attributes and modes of operation, see the *Bundled Agents Reference Guide*.

Agents packaged separately from the product for use with 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 configuration files.

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

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

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 ScriptAgent 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 variables (attributes) to provide error checking, and to provide default values for certain attributes for the entire type.

All resource types need definition. One of the attributes that is defined for the IP resource type is the IPAddress 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.

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 22.

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 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 22.

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, depending on the entry points required to control them.

- On-off resources

  Most resources are on-off, meaning agents start and stop them as required. For example, the engine assigns an IP address to a specified NIC when bringing a resource online and removes the assigned IP address when taking the service group offline. Another example is the DiskGroup resource. The engine 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 when required by the engine, 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. For example, a NIC resource cannot be started or stopped, but it is required to configure an IP address.

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 that define the 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.
- **Type-dependent**—Attributes that apply to a particular resource type. These attributes appear in the type definition file (.cf) for the agent. 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 file types.cf apply to all resources of the resource type. Defining these attributes in the main.cf file overrides the values in the .cf file for a specific resource.
  For example, setting StartVolumes = 1 for the DiskGroup resource type defaults StartVolumes to True for all DiskGroup resources. Setting the value in main.cf overrides the value on a per-resource basis.
- **Static**—These attributes apply for every resource of a particular type. These attributes are prefixed with the term static and are not included in the resource's argument list. You can override some static attributes and assign them resource-specific values.

Attribute data types

VCS supports the following data types for attributes.

String — A string is a sequence of characters enclosed by double quotes. A string may also contain double quotes, but the quotes must be immediately preceded by a backslash. 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”

VCS also supports UTF-8 encoded values for some attributes.

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. This is the default dimension.

For example:

```
str MountPoint
```

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

```
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.

For example:

```
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[] = { Address, SubNetMask,
MACAddress }
```

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 }
```

Localizable attributes

VCS supports UTF-8 encoded localized values for some attributes. These attributes are identified by the i18nstr keyword in the type definition file types.cf.

For example, in the FileOnOff agent, the attribute PathName is a localizable attribute.

```
type FileOnOff (
    static i18nstr ArgList[] = { PathName }
    i18nstr PathName
)
```

You can add a localizable string attribute using the `haattr -add -i18nstring` command.

Attribute scope across systems: global and local attributes

An attribute whose value is the same across all systems in the SystemList is *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.

Windows: An example of local attributes can be found in the IP resource type where Mac addresses and routing options are assigned per machine.

```
    IP IP_resource (
    Address = "192.168.1.201"
    SubNetMask = "255.255.254.0"
    MACAddress @ system1= "02-B1-D5-D1-88-0E"
    MACAddress @ system2= "04-B0-D0-D1-88-43"
    )
```

Attribute life: temporary attributes

You can define temporary attributes in the types.cf file. The values of temporary attributes remain in memory as long as the VCS engine (HAD) is running. Values of temporary attributes are not available when the engine 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 temporary attributes and their default values in the file types.cf.

You can define and modify these attributes only while VCS is running.

Developing an agent: overview

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 cluster environment. The following criteria describe an application that can successfully operate in a cluster:

- The application must be capable of being started by a defined procedure. Specific commands must be available to start 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. Monitoring an application becomes more effective when the monitoring procedure resembles the actual activity of the application's user.
- The application must be capable of storing data on shared disks rather than locally or in memory, and each system must be capable of accessing the data and all information required to run the application.
- The application must be crash-tolerant, that is, it must be capable of being run on a system that crashes and of being started on a failover node in a known state. This typically means that data is regularly written to shared storage rather than stored in memory.
- The application must be host-independent within a cluster; that is, there are no licensing requirements or host name dependencies that prevent successful failover.
- The application must run properly with other applications in the cluster.

Creating an agent: highlights

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

Developing the entry points

Decide whether to implement the agent entry points using C++ code, scripts, or a combination of the two. There are advantages and disadvantages implementing entry points in either method.

See "Considerations for using C++ or script entry points" on page 34.

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.cf` file contains definitions of resource types that come bundled with VCS. A custom resource type definition should be placed in a file called *Resourcetype*`Types.cf`, for example, MyResourceTypes..cf or OracleTypes.cf. This file is referenced as an "include" statement in the configuration file, `main.cf`.

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.

- On Windows, create the agent using sample files in %VCS_HOME%\src\agent\Sample. Build the agent binary and place it %VCS_HOME%\bin*resource_type*.
- For building a scripts agent on Windows, use DefaultAgent.dll or Default50Agent.dll.
  On Windows systems, copy the dll in the agent directory.
  %VCS_HOME%\bin*resource_type*
  If you use script entry point, install the files files in the directory %VCS_HOME\bin*resource_type*

Testing the agent

Test the agent 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 Attribute
- About the agent information file

About agent entry points

Developing an agent means 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 commonly used 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 for a resource being taken offline
- `action` - starts a defined action for a resource
- `info` - retrieves specific information about a resource

The agent framework also supports the following entry points:

- `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 23.

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`, all 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 142).

It returns the resource status (online, offline, or unknown), and the confidence level 0–100. 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.

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

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

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

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

The `ResourceInfo` attribute is a string association that stores name-value pairs. By default, there are three such name-value pairs: State, Msg, and TS. State indicates the status, valid or invalid, 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.

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.

For a description of the `ResourceInfo` attribute, see "ResourceInfo resource attribute used by info entry point" on page 26.

For input, the `info` entry point receives as arguments the resource name, the value of `resinfo_op`, and the `ArgList` attribute values.

In the case of C++ implementation, the output of the entry point is returned in `info_output`. The output buffer has a 2048-byte limit.

For any optional name-value pairs, which are returned in either `opt_add_args` or `opt_update_args` two-dimensional character arrays, the name and the value each have a limit of 4096 bytes.

See the C++ example, "Example: info entry point implementation in C++" on page 52. For the script example, see "info" on page 94.

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.
- 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.

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 144.

ResourceInfo resource attribute used by info entry point

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.

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

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

The resource for which the info entry point is invoked must be online. When a resource goes offline or faults, the State key is marked "Stale" because the information is not current. If the info entry point exits abnormally, the State key is marked "Invalid" because not all of the information is known to be valid. 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.

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

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

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

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.

Note: On Windows, VCSAgCleanOfflineHung, VCSAgCleanOnlineHung, and VCSAgCleanMonitorHung do not apply. Entry point timeouts are not configurable parameters on the Windows platform.

- VCSAgCleanOfflineHung
  The `offline` entry point did not complete within the expected time. This parameter is not used on Windows.
- 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. This parameter is not used on Windows.

- 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.
  This parameter is not used on Windows.

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

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

Each action is identified by an action_token. The action_token is a name for an action. The list of supported actions (described by the action_tokens) for a resource type is described in the SupportedActions keylist.

You must list the action in the SupportedActions attribute. You can define actions all C++ actions or all script actions.

Make sure the action scripts reside within an `action` 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 “action” on page 59.

The following shows the syntax for the `-action` option used with the `hares` command:

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

The 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 -actionsargs dbstart -sys Sys1
```

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 action entry point output to 2048 bytes.

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 is registered with the agent framework for notification.

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. Most agents do not require this functionality and will not implement this entry point.

See "VCSAgRegister" on page 68.

See "VCSAgUnregister" on page 69.

You can register resources automatically.

See "RegList" on page 151.

About the open entry point

When an agent starts, the `open` entry point of each resource defined in the configuration file 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 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. This entry point typically initializes the resource.

Note: The 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. 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.

About the close entry point

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

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 cluster. 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

Note: 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.

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 Script-Based Exit values: ■ 100 - Offline ■ 101-110 - Online ■ 99 - Unknown
Info	0 if successful; non-zero value if not successful
Online	Integer specifying number of seconds to wait before monitor can check the state of the resource; typically 0, 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. Refer to Chapter 9, "State transition diagrams" on page 155 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++, Perl, and shell in any combination to implement multiple entry points for a single agent. This allows you to implement each entry point in the most advantageous manner. For example, you may use scripts to implement most entry points while using C++ to implement the `monitor` entry point, which is called often. If the `monitor` entry point 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 43.

See Chapter 4, “Creating entry points in scripts” on page 89.

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 Windows, use the DefaultAgent.dll or the DefaultAgent50Agent.dll.

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.

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(V50);

    VCSAgSetEntryPoint(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:

Windows

`%VCS_HOME%\bin\`*`resource_type`*`\<entry_point>.extn`

Note: For Windows, the extension (.extn) assumes the value “exe” for executable programs, “sh” for shell scripts compatible with MKS Toolkit, “bat” or “cmd” for batch files, “pl” for Perl scripts, “js” for Javascript, or “vbs” for Visual Basic scripts.

Script agents

On Windows, script agents use DefaultAgent.dll and Default50Agent.dll.

See Chapter 4, “Creating entry points in scripts” on page 89.

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 (Windows)

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="5.0">
    <agent_description>Creates, removes, and monitors files.
    </agent_description>
    <!--Platform the agent runs on-->
    <platform>Windows</platform>
    <!--Type of agent : script-Binary-Mixed-->
    <agenttype>Binary</agenttype>
    <!--info entry point implemented or not-->
    <info_implemented>No</info_implemented>
    <!--The minimum VCS version needed
         for this agent-->
    <minvcsversion>4.01.0</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\FileOnOff.dll" />
</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="5.0"`
Agent description	`<agent_description>Creates, removes, and monitors files.</agent_description>`
Platform. For example, Windows 2000 i386, or Solaris Sparc, or HP-UX 11.11.	`<platform>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>`
VCS compatibility; the minimum version required to support the agent	`<minvcsversion>4.0</minvcsversion>`

Attribute argument details

The agent's attribute information is described by several arguments. 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 16.
dimension	Values for the attribute dimension, such as "Scalar;" See "Localizable attributes" on page 18.
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 cluster node.
3. Restart the GUI to have the agent's information shown in the GUI.

About the ArgList Attribute

The `ArgList` attribute is used to pass resource type attributes and their values to the `open`, `close`, `online`, `offline`, `action`, `info`, and `monitor` entry points.

The values of the `ArgList` attribute are passed through a parameter of type `void **`. For example, the signature of the `online` entry point is:

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

The ArgList attribute behavior varies depending on whether the agent is registered as a V51, V50, or V40.

ArgList attribute for agents registered as V50

For agents registered as V50, the ArgList attribute passes attributes and values to the entry points in tuple format through the parameter attr_val.

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

ArgList Attribute for agents registered as V40 and earlier

For agents registered as V40 and earlier, the ArgList attribute is a predefined static attribute that specifies the list of attributes whose values are passed to the entry points.

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

The values of scalar attributes (integer and string) are each contained in a single element of `attr_val`. The values of non-scalar attributes (vector, keylist, and association) are contained in one or more elements of `attr_val`.

The `open`, `close`, `online`, `offline`, `monitor`, `action`, `info`, and `clean` scripts receive the resource name and values of the `ArgList` attributes. The ArgList attribute behavior varies depending on whether the agent uses the

For agents registered as V40 and earlier:

- The values of scalar `ArgList` attributes (integer and string) are each contained in a single command-line argument.
- The values of complex `ArgList` attributes (vector, keylist, and association) are contained in one or more command-line arguments.

If a vector or association attribute contains *N* components, it is represented by *N*+1 command-line arguments. The first command-line argument is *N*, and the remaining *N* arguments correspond to the *N* components. (See "ArgList reference attributes" on page 142.)

See the chapter describing the configuration language in the *Administrator's Guide* for attribute definitions.

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

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

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

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

Then, for V50, 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
- C++ entry point syntax
- Agent framework primitives
- Agent framework primitives for Windows

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:

Windows: %VCS_HOME%\src\agent\Sample

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 type.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

```
// Values for the state of a resource - returned by the
// monitor entry point.

    enum VCSAgResState {
    VCSAgResOffline,          // Resource is offline.
    VCSAgResOnline,           // Resource is online.
    VCSAgResUnknown           // Resource is neither online nor
                              // offline.
    };

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

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

Note: On Windows, VCSAgCleanOfflineHung, VCSAgCleanOnlineHung, and VCSAgCleanMonitorHung do not apply. Entry point timeouts are not configurable parameters on Windows.

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

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

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

C++ entry point syntax

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

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

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V50);

    VCSAgSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgSetEntryPoint(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) {
...
}
```

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 33.

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(V50);

    VCSAgSetEntryPoint(VCSAgEPMonitor, res_monitor);
}
```

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 three keys State, Msg, and TS to the attribute 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 `VCSAgEPInfo` parameter of `VCSAgSetEntryPoint()` primitive 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 V50 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
```

The following example is for a V40 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

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

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

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

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

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

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

online

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

You may select any name for the function.

Set the VCSAgEPInfo parameter of VCSAgSetEntryPoint() primitive 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(V50);

    VCSAgSetEntryPoint(VCSAgEPOnline, res_online);
}
```

offline

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

You may select any name for the function.

Set the VCSAgEPInfo parameter of VCSAgSetEntryPoint() primitive 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(V50);

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

clean

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

You may select any name for the function.

Set the VCSAgEPInfo parameter of VCSAgSetEntryPoint() primitive 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(V50);

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

action

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

You may select any name for the function. The agent framework truncates the output for the `action` entry point to a maximum of 2048 bytes.

Set the VCSAgEPInfo parameter of VCSAgSetEntryPoint() primitive to the name of the entry point's function.

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;
}
```

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 Attribute" on page 39.

You may select any name for the function.

Set the VCSAgEPInfo parameter of VCSAgSetEntryPoint() primitive 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 68, or the agent parameter RegList (see "RegList" on page 151).

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(V50);

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

open

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

You may select any name for the function.

Set the VCSAgEPInfo parameter of VCSAgSetEntryPoint() primitive 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(V50);

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

close

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

You may select any name for the function.

Set the VCSAgEPInfo parameter of VCSAgSetEntryPoint() primitive 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(V50);

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

shutdown

```
void res_shutdown();
```

You may select any name for the function.

Set the VCSAgEPInfo parameter of VCSAgSetEntryPoint() primitive to the name of the entry point's function.

In the following example, the function res_shotdown 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(V50);

    VCSAgSetEntryPoint(VCSAgEPShutdown, res_shutdown);
    VCSAgSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgSetEntryPoint(VCSAgEPOffline, res_offline);
    VCSAgSetEntryPoint(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 Windows" on page 79

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.

For example:

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

#include "VCSAgApi.h"

void VCSAgStartup() {
        VCSAgV40EntryPointStruct ep;

        ep.open = NULL;
        ep.close = NULL;
        ep.monitor = res_monitor;
        ep.online = res_online;
        ep.offline = res_offline;
        ep.action = NULL;
        ep.info = NULL;
        ep.attr_changed = NULL;
        ep.clean = NULL;
        ep.shutdown = NULL;
        VCSAgRegisterEPStruct(V50, &ep);
}
```

VCSAgSetCookie

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

This primitive requests the agent framework to store a cookie. 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();
                            VCSAgSetCookie(res_name, key);
                }
}
VCSAgResState  res_monitor(const char *res_name, void
                **attr_val, int *conf_level_ptr) {
                VCSAgResState state = VCSAgResUnknown;
                *conf_level_ptr = 0;
                void *key = VCSAgGetCookie(res_name);
                if (key == NULL) {
                    // Take care of the rare cases when
                    // the open entry point failed to
                    //  obtain the key and set the the cookie.
                            key = get_key();
                            VCSAgSetCookie(res_name, key);
                }
                // Use the key for testing if the resource is
                // online, and set the state accordingly.
                ...
                return state;
}
```

VCSAgRegister

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

This primitive requests that the 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 `VCSAgSetCookie()`. It returns `NULL` if cookie was not previously set. This primitive can be called from any entry point. For example:

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

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

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 Get Exit Code Process() on Windows

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.

Note: This API is not available on Windows.

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.

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

Mt-safe; deferred cancel safe.

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. This primitive replaces VCSAgRegisterEPStruct().

Examples:

```
VCSAgInitEntryPointStruct(V40);
VCSAgInitEntryPointStruct(V50);
VCSAgInitEntryPointStruct(V51);
```

If the agent version is less than 4.0, the following entry points are supported:

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

If the agent version is >= 4.0 and < 5.0, the agent supports definitions for the following additional entry points:

- info
- action

If the version is >= 5.0, the agent supports the following additional entry points:

- amf_init
- amf_register
- amf_getnotification

VCSAgSetStackSize

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

Set the calling thread's stack size to the value specified (that is, 1).

Note: This API is not available on Windows.

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.

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

Mt-safe; deferred cancel safe.

VCSAgDisableCancellation

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

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

Note: This API is not available on Windows.

VCSAgRestoreCancellation

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

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

Note: This API is not available on Windows.

VCSAgSetEntryPoint

```
void VCSAgSetEntryPoint(VCSAgEntryPoint ep, void *f_ptr)
```

This primitive enables an agent developer to register any C++ entry point with the agent framework. This primitive replaces the VCSAgRegisterEPStruct() API.

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

```
enum VCSAgEntryPoint {
        VCSAgEPAction,
        VCSAgEPAttrChanged,
        VCSAgEPClean,
        VCSAgEPClose,
        VCSAgEPInfo,
        VCSAgEPMonitor,
        VCSAgEPOffline,
        VCSAgEPOnline,
        VCSAgEPOpen,
        VCSAgEPStartup,
        VCSAgEPShutdown,
        VCSAgEPAMFInit,
        VCSAgEPAMFGetNotification,
        VCSAgEPAMFRegister
};
```

For example:

```
VCSAgSetEntryPoint(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.

Note: This API is not available on Windows.

Agent framework primitives for Windows

This chapter describes agent primitives for use by agents in a Windows environment. These primitives are used for:

- Logging messages
- Converting ArgList values from UTF-8 encoding to UCS-2 encoding (wchar)
- Manipulating UCS-2 (wchar) arguments
- Allocating and freeing memory

C++ logging primitives for use by Windows agents

The four C++ logging primitives provided by the agent framework include the 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 two debugging macros:

```
VCSAG_LOGDBG_MSG (dbgsev,flags,fmt,variable_args … )
VCSAG_RES_LOG_MSG (dbgsev,flags,fmt,variable_args … )
```

See “C++ agent logging APIs” on page 102.

Using logging primitives with localized Windows agents

The C++ logging primitives can be used to log Windows agent messages that use `wchar` arguments. The agents can simply pass in a format string in `wchar` as well as arguments in `wchar` to the log macros. The agent framework converts the wchar message to UTF-8 encoding before writing them to log files.The log macros accept a variable number of arguments as input. The arguments can be `wchar` string, `wchar`, integer, or floating point values.

Defining UCS-2 encoding when compiling agents

To enable the logging primitives to log messages using UCS-2 encoding (wchar), compile the agent with the “UNICODE” flag defined. Otherwise, compilation fails with errors that indicate a mismatch between the expected argument types and the actual UCS-2 encoded messages sent as arguments.

A sample rule to use to build an agent named MyAgent

```
.
.
MyAgent.o:   MyAgent.C
       $(CCPLUS) /FoMyAgent.o -DUNICODE -D_UNICODE -c MyAgent.C
.
.
MyAgent:     MyAgent.o
       $(CCLINK) /dll /OUT:MyAgent MyAgent.o $(NEED_LIBS)
.
.
```

where CCPLUS, CCLINK, NEED_LIBS are set as required for the compilation and linking of the agent code.

C++ primitives for ArgList conversion

The two primitives provided for converting the UCS-2 encoded `ArgList` attribute are:

- `VCSAgGetEncodedArgList`
- `VCSAgDelEncodedArgList`.

VCS on Windows allows users to use localized values for attributes. Therefore, Windows agents must handle UCS-2 encoded or OS encoded values. All arguments in the `ArgList` attribute are passed to the Windows agents in UTF-8 encoding, which the agent must convert to UCS-2 encoding or OS encoding. The `VCSAgGetEncodedArgList` is provided for this purpose.

`VCSAgGetEncodedArgList` allocates memory as needed to the encoded `ArgList` array output. This memory can be freed by the caller using the primitive `VCSAgDelEncodedArgList`.

See "VCSAgDelEncodedArgList" on page 81).

VCSAgGetEncodedArgList

```
int VCSAgGetEncodedArgList (VCSAgEncodingType from_encoding,
void
    **input_arglist_values, VCSAgEncodingType to_encoding, void
    ***output_arglist_values)
```

This primitive converts the ArgList values in UTF-8 encoding to UCS-2 or the OS encoding. `VCSAgGetEncodedArgList` stores a pointer to the encoded array in the "`output_arglist_values`" and returns 0 on success and 1 on failure.

The input `ArgList` array must be terminated with a NULL string to indicate the end of the input `ArgList`.

Note: This primitive only supports conversions to UCS-2 and OS encoding.

VCSAgDelEncodedArgList

```
void VCSAgDelEncodedArgList (void **enc_arg_list)
```

This primitive must be used to free the memory allocated by the agent in the primitive `VCSAgGetEncodedArgList`. This primitive expects the input, `enc_arg_list`, to be terminated by a NULL string.
`VCSAgDelEncodedArgList` can be called from any entry point that is passed the `ArgList` values.

C++ primitives for ArgList string manipulation

Three primitives provided for manipulating wchar strings in UCS-2 encoding are:

- VCSAgStrlcpyW
- VCSAgStrlcatW
- VCSAgSnprintfW

VCSAgStrlcpyW

```
void VCSAgStrlcpyW (wchar *dst, const wchar *src, int size)
```

This primitive must be used by agents to perform string copy operations on wchar arguments. The argument "size" refers to the number of characters in the destination buffer "dst." This helps prevent buffer overflow errors. The output contained in the buffer "dst" may be truncated if the buffer is not large enough.

VCSAgStrlcatW

```
void VCSAgStrlcatW (wchar *dst, const wchar *src, int size)
```

This primitive must be used in the agents to perform a string concatenation operation of wchar arguments. The argument "size" refers to the number of characters in the destination buffer "dst." This helps prevent buffer overflow errors. The output contained in the buffer "dst" may be truncated if the buffer is not large enough.

VCSAgSnprintfW

```
int VCSAgSnprintfW (wchar *buffer, int size, const wchar
    *format, … )
```

This primitive accepts a variable number of arguments and works like the C library function "vswprintf," except that VCSAgSnprintfW takes in the size of the output buffer "dst" as an argument. 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 in the buffer is not large enough.

C++ primitives for allocating and freeing memory

Two primitives are provided for use by Windows agents to allocate and free memory:

- VCSAgAllocMem
- VCSAgDeleteMem

VCSAgAllocMem

```
void *VCSAgAllocMem (size_t size)
```

This primitive can be used to allocate "size" bytes of memory. The primitive VCSAgDeleteMem must be used to free the memory that is allocated using this API.

VCSAgDeleteMem

```
void VCSAgDeleteMem (void *mp)
```

This primitive can be used to free memory allocated to "mp." The memory for "mp" should have been allocated in a previous call to the API VCSAgAllocMem.

Example: Windows-based primitives for the FileOnOff agent

The following example illustrates the use of the Windows-based primitives within an agent for FileOnOff.

```
/* @(#)src/agent/w2k/FileOnOff/FileOnOff.C      2.4 01/04/03
11:57:11 -   */
static const char sccsid[] =
"@(#)VCS:src/agent/w2k/FileOnOff/FileOnOff.C 2.4";
/*
 * START COPYRIGHT-NOTICE: 1998, 2003
 *
 * Copyright 2008 Symantec Corporation.
 * All rights reserved.
 *
*/
#include <stdio.h>
#include <windows.h>
#include "VCSAgApi.h"

WCHAR *ConvertToUnicodeString(const char *attr_val)
{
        int len = 0,outLen ;
        WCHAR * szOutStr = NULL;
        int num_bytes = 0, num_written = 0;
        len = strlen(attr_val);
```

```
        // Call the API to convert the UTF-8 encoded string into
a
        // UCS-2 encoded string
        VCSAgEncodeString(VCSAgUTF8, /* convert from UTF8 */
                        (void *)attr_val,
                        len,
                        VCSAgUCS2, /* to the UCS2 format */
                        (void **)&szOutStr,
                        &outLen);

        return szOutStr;
}
////////////////////////////////////////////////////////////////
////////////////
// Function:    res_monitor
//
// Description: Determine if the given file is online (file
exists)
//              or offline (file does not exist).
//
// Notes:       None
////////////////////////////////////////////////////////////////
////////////////

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

        VCSAgResState state;
        DWORD attrs ;

        // Temporary variables for sample code
        WCHAR * szPath;
        WCHAR *mymem = NULL;
        WCHAR **mymem2 = NULL;
        WCHAR **new_args = NULL;
        int ret = 0;
        int indx = 0;
        int temp_number = 0;

        // Dummy structure defined for the purpose of example
code
        struct mystruct{
                int a;
                WCHAR b[32];
        } *inst1;
```

```
/*************************************************************
***
         * Sample code for using the new AGFW APIs

*************************************************************
**/

        // Allocate 32 bytes of type WCHAR for mymem
        mymem = VCSAgNew(WCHAR, 32);

        if (mymem) {
                // Memory allocation succeeded
                VCSAgStrlcpyW(mymem, L"copy ", 32);

                VCSAgStrlcatW(mymem, L"concatenate", 32);

                VCSAG_LOGDBG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
                        L"Sample string is %s", mymem);

                temp_number = 10;
                VCSAgSnprintfW(mymem, 32,
                       L"Testing VCSAgSnprintfW...temp_number is
                        %d\n", temp_number);

                VCSAG_LOGDBG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
                        L"Sample string is %s", mymem);

                // Delete memory allocated to mymem
                VCSAgDelete((void *)mymem);
                mymem = NULL;

        } else {
                // Memory allocation failed
                VCSAG_LOG_MSG(VCS_NOTICE, 1005,
VCS_DEFAULT_FLAGS,
                        L"Failed to allocate memory for mymem");
        }

        // Allocating memory for a struct now

        inst1 = VCSAgNew(struct mystruct, 1);

        if (inst1) {
                // Memory allocation succeeded
                VCSAgStrlcpyW(inst1->b,
                        L"Copying dummy string", 32);

                VCSAG_LOGDBG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
                        L"Sample string is %s", inst1->b);

                // Delete memory allocated to inst1
```

```
                VCSAgDelete(inst1);

        } else {
                // Memory allocation failed
                VCSAG_LOG_MSG(VCS_NOTICE, 1006,
VCS_DEFAULT_FLAGS,
                        L"Failed to allocate memory for inst1");
        }

        // Allocating memory for an array of pointers
        // Here, there will be four entries in the array
        myptr = VCSAgNew(WCHAR *, 4);
        if (myptr) {
                // Memory allocation succeeded
                myptr[0] = VCSAgNew(WCHAR, 16);
                VCSAgStrlcpyW(myptr[0], L"One", 16);

                myptr[1] = VCSAgNew(WCHAR, 16);
                VCSAgStrlcpyW(myptr[1], L"Two", 16);

                VCSAG_LOGDBG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
                        L"Contents of myptr: %s and %s",
                        myptr[0], myptr[1]);

                // Free the memory allocated
                VCSAgDelete(myptr[0]);
                VCSAgDelete(myptr[1]);
                VCSAgDelete(myptr);
        }

        // Call the function that uses the VCSAgEncodeString API
        szPath = ConvertToUnicodeString((char *)attr_val[0]);
        VCSAG_LOGDBG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
                L"Encoded string is %s", szPath);

        // Sample usage of VCSAgGetEncodedArgList API
        ret = VCSAgGetEncodedArgList(VCSAgUTF8, attr_val,
                VCSAgUCS2, (void ***)&new_args);

        if (ret) {
                // Encode API failed
                VCSAG_LOG_MSG(VCS_NOTICE, 1007,
VCS_DEFAULT_FLAGS,
                        L"Failed to encode the ArgList entries");
        } else {
                // Encode succeeded
                VCSAG_LOGDBG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
                        L"The encoded arg list values are:");
```

```
                while (new_args[indx] != NULL) {
                     VCSAG_LOGDBG_MSG(VCS_DBG1,
VCS_DEFAULT_FLAGS,
                    L"arg %d is %s", indx, new_args[indx]);
                    indx++;
                }

                // Sample usage of API to delete the encoded
                //ArgList entries

                VCSAgDelEncodedArgList((void **)new_args);

        }


/*************************************************************
***
         * End of sample code for using the new AGFW APIs

*************************************************************
**/

        :
        // Actual code here
        :
        :

        VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
                L"Monitor is successful");

        if(szPath)
                VCSAgDelString((void *)szPath);

        return state;
}

        :
        // Rest of the agent code
        :
        :
```

Chapter 4

Creating entry points in scripts

- About creating entry points in scripts
- Script entry point syntax
- Example script entry points for the FileOnOff resource

About creating entry points in scripts

On Windows you may use the VCSdefault.dll.

`%VCS_HOMEVAD_HOME%\bin\VCSdefault.dll`

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, you may use the ScriptAgent or the Script50Agent binary, wherein all fields in VCSAgV40EntryPointStuct are set to NULL (that is, all entry points are implemented as scripts).

On Windows, use VCSdefault.dll.

See "Implementing entry points using scripts" on page 124.

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:

- You can use (.bat) files too. To use Shell scripts, you must first install MKSToolkit or a similar software application.
- The name of the script file must be the same as the entry point, followed by a period and the appropriate extension. The extension assumes the value exe for executable programs, sh for shell scripts compatible with MKS Toolkit, bat or cmd for batch files, pl for Perl scripts, js for Javascript, or vbs for Visual Basic scripts.
- Place the file in the directory %VCS_HOME%\bin\resource_type.
  For example, if the `online` entry point for Oracle is implemented using Perl, the online script must be:
  `%VCS_HOME%\bin\Oracle\online.pl`

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.

ArgList attributes

See “About the ArgList Attribute” on page 39.

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 V50 agent, when invoked for Bar, resemlbes:

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

The online script for a V40 agent, when invoked for Bar, resemlbes:

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

Script entry point syntax

The following paragraphs describe the syntax for script entry points.

monitor

```
monitor resource_name  ArgList_attribute_values
```

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

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

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

online

```
online resource_name  ArgList_attribute_values
```

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

offline

```
offline resource_name  ArgList_attribute_values
```

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

clean

`clean` *`resource_name clean_reasonrgList_attribute_values`*

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

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

1 - The `offline` entry point was ineffective.

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

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.

The exit value is 0 (successful) or 1.

Note: Because entry point timeouts cannot be set for agents running on Windows, clean_reason values of 0, 2, and 5 do not apply.

---

action

`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.

attr_changed

```
attr_changed resource_name changed_resource_name
        changed_attribute_name new_attribute_value
```

The exit value is ignored.

Note: This entry point is called only if you register for change notification using the primitive `VCSAgRegister()` (see "VCSAgRegister" on page 68), or the agent parameter `RegList` (see "RegList" on page 151).

---

info

```
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.

open

```
open resource_name  ArgList_attribute_values
```

The exit value is ignored.

close

```
close resource_name  ArgList_attribute_values
```

The exit value is ignored.

shutdown

```
shutdown
```

The exit value is ignored.

Example script entry points for the FileOnOff resource

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

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

Online entry point for FileOnOff

The FileOnOff example entry point is simple. When the agent's `online` entry point is called by the agent, the entry point expects the name of the resource as the first argument, 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.

In this example of an agent earlier than V50, the value of `PathName` attribute is the second argument. The agent creates the file `test` in the specified path.

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. 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.

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.

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 173.
- For information on APIs used by VCS 3.5 and earlier, see "Log messages in pre-VCS 4.0 agents" on page 186.

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 when attempting 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 (VCS, in this case), the *severity*, and the *UMI* (unique message ID). The subsequent lines contain the *message text*.

Timestamp

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

Mnemonic

The mnemonic field is used to indicate the product.

The mnemonic, 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-" assigned by Symantec.
- The category ID is a number in the range of 0 to 65536 assigned by Symantec. For each custom agent, Symantec must be contacted so that a unique category ID can be registered for the agent.

- For C++ messages, the category ID is defined in the `VCSAgStartup` entry point.
  See “Log category” on page 107.
- For script-based entry points, the category is set within the VCSAG_SET_ENVS function
  See “VCSAG_SET_ENVS” on page 112.
- 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.

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 112).

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 engine log, and, in the case of messages of CRITICAL and ERROR severity, 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 105.
`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 145). 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 105.
`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 106.

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 108 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 2001:

```
VCSEXPORT void VCSDECL VCSAgStartup()
{
        VCSAG_LOG_INIT("VCSAgStartup");
       VCSAgInitEntryPointStruct(V50);


       VCSAgSetEntryPoint(VCSAgEPMonitor, res_monitor);
       VCSAgSetEntryPoint(VCSAgEPOnline, res_online);
       VCSAgSetEntryPoint(VCSAgEPOffline, res_offline);
       VCSAgSetEntryPoint(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(V50);

       VCSAgSetEntryPoint(VCSAgEPMonitor, res_monitor);
       VCSAgSetEntryPoint(VCSAgEPOnline, res_online);
       VCSAgSetEntryPoint(VCSAgEPOffline, res_offline);
       VCSAgSetEntryPoint(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, functions, or *wrappers*, can call the `halog` command for message logging purposes. While the `halog` command can be called directly within the script to log messages, the following entry point logging functions are easier to use and less error-prone:

VCSAG_SET_ENVS - sets and exports entry point environment variables

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

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

VCSAG_SET_ENVS

The VCSAG_SET_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, VERITAS assigns the category ID. See the category ID description in "UMI" on page 101. 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. Windows: %VCS_HOME%\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. 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.
Message (*msg*)	A text message in quotes; for example: “One file copied”
Encoding *Format*	UTF-8, ASCII, or UCS-2 in the form: “-encoding *format*”
Parameters	Parameters (up to six), each within quotes

VCSAG_LOGDBG_MSG examples, Shell script entry points

- Calling a function without encoding or parameters:

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

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

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

- Calling a function with a parameter and encoding format:

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

VCSAG_LOGDBG_MSG examples, Perl script entry points

- Calling a function:

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

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

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

- Calling a function with a parameter and encoding format:

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

Using the functions in scripts

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

- Shell Script include file

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

- Perl Script include file

```
use ag_i18n_inc;
```

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 Windows
- 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.	Windows: %VCS_HOME%\bin\VCSdefault.dll VCSdefault.dll cannot be used with C++ entry points.

Table 6-2 C++ Agents

Description	Pathname
Directory containing a sample C++ agent and `Makefile`.	Windows: VCS_HOME%\src\agent\Sample
Sample `Makefile` for building a C++ agent.	Windows: %VCS_HOME%\src\agent\Sample\Makefile
Entry point templates for C++ agents.	Windows:%VCS_HOME%\src\agent\Sample\Sample.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.

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`.

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 };
)
```

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

Adding the custom type definition to the configuration

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

Windows: %VCS_HOME%\conf\config\

Building a custom agent on Windows

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 use the keyword VCS_HOME to denote the VCS installation directory. Substitute VCS_HOME with the actual directory name when entering commands.

The examples assume:

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.

Recommendations for building custom agents on Windows

- For multi-system configurations, you must install the agent on each system.
- Compiling is not required if all entry points are implemented using scripts. On Windows, a copy of VCSdefault.dll is sufficient.
  Compiling is required to build the agent if any entry points are implemented using C++.

Implementing entry points using C++

1. Edit sample.C to customize the implementation; sample.C is located in the directory %VCS_HOME%\src\agent\Sample.
2. After completing the changes to `sample.C`, invoke the `nmake` command to build the agent. The command is invoked from %VCS_HOME%\src\agent\Sample..
3. Name the agent DLL *`resource_type`*`Agent.`
4. Place the agent in `%VCS_HOME%\bin\`*`resource_type`*.
5. We also recommend naming the agent DLL resource_type.dll. The agent DLL for Oracle would be %VCS_HOME%\bin\Oracle\Oracle.dll.
   For example, the agent DLL for Oracle would be %VCS_HOME%\bin\Oracle\Oracle.dll.

Example: Using C++ entry points on Windows

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.

Note: You must first install Visual C++ on the system on which the agent will be built.

Example: VCSAgStartup and C++ entry points

Create the following directory for the agent sources: %VCS_HOME%\src\agent\FileOnOff:

```
C:\> md %VCS_HOME%\src\agent\FileOnOff
```

6. Copy the contents of the sample agnet directory to the directory you created in the previous step:

```
C:\> cp %VCS_HOME%\src\agent\Sample\* %VCS_HOME%\src\
        agent\FileOnOff
```

7. Change to the new directory:

```
C:\> cd %VCS_HOME%\src\agent\FileOnOff
```

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

```
void VCSAgStartup()
    {
    // This example implements online, offline, and
```

```
    // monitor entry points using C++.

    VCSAG_LOG_INIT("VCSAgStartup");
    VCSAgSetLogCategory(10051);

    VCSAgInitEntryPointStruct(V50);
    VCSAgSetEntryPoint(VCSAgEPMonitor, res_monitor);
    VCSAgSetEntryPoint(VCSAgEPOnline, res_online);
    VCSAgSetEntryPoint(VCSAgEPOffline, res_offline);
}
```

9 Modify `res_online()` and `res_offline()`:

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

    unsigned int res_online(const char *res_name,
        void **attr_val) {
        WCHAR                   **new_args = NULL;
        DWORD                   ret;
        HANDLE                  hFile = NULL;

VCSAG_LOG_INIT("res_online");
VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
                  L"Inside online.");

ret = VCSAgGetEncodedArgList(VCSAgUTF8, attr_val, VCSAgUCS2,
 (void ***)&new_args);

if (ret)
{
        VCSAG_LOG_MSG(VCS_NOTICE, 1001, VCS_DEFAULT_FLAGS,
          L"Unable to get the arguments");
        goto exit;
}
hFile = CreateFile((WCHAR *)new_args[0], 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,
  L"Unable to create the file" );
}
else
{
VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
  L"Online successful."); CloseHandle(hFile);
```

```
    }

exit:
    if(new_args)
            VCSAgDelEncodedArgList((void**)new_args);
    return 0;
}
       // This is a C++ implementation of the offline
       // entry point for the FileOnOff resource type.
       // This function takes offline aFileOnOff
       // resource by deleting the corresponding file. It
       // is assumed that the complete pathname of the
       // file will be passed as the first
       // ArgList attribute.
unsigned int res_offline(const char *res_name,
    void **attr_val)
{
    WCHAR                   **new_args = NULL;
    DWORD                   ret;

    VCSAG_LOG_INIT("res_offline");
    VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
          L"Inside offline.");

    ret = VCSAgGetEncodedArgList(VCSAgUTF8, attr_val,
          VCSAgUCS2,(void ***)&new_args);

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

    if (!DeleteFile((WCHAR *)new_args[0]))
{
            VCSAG_LOG_MSG(VCS_ERROR, 1003, VCS_DEFAULT_FLAGS,
                 L"Unable to delete the file" );
}
    else
    {
            VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
                 L"offline successful.");
    }

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

10 Modify the `res_monitor()` function.

See "Example: Using VCSAgStartup() and script entry points on Windows" on page 125.

11 Compile `sample.C` and build the agent by invoking `nmake`. (`Makefile` is provided.)

```
C:\> nmake
```

12 Create a directory for the agent binaries:
%VCS_HOME%\bin\FileOnOff:

```
C:\> md %VCS_HOME%\bin\FileOnOff
```

13 Place the sample.dll as the FileOnOff agent library.

```
C:\> copy sample.dll %VCS_HOME%\bin\FileOnOff\FileOnOff.dll
```

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 in the following manner:

%VCS_HOME%\bin\resource_type\entry_point.extn

The extension (.extn) assumes the value exe for executable programs, sh for shell scripts compatible with MKS Toolkit, bat or cmd for batch files, pl for Perl scripts, js for Javascript, or vbs for Visual Basic scripts.

For example, if the online entry point for Oracle is implemented using Perl, the online script must be: %VCS_HOME%\bin\Oracle\online.pl.

We also recommend naming the agent DLL resource_type.dll. Place the agent in %VCS_HOME%\bin\resource_type.

Example: Using script entry points on Windows

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 without C++

1 Create the directory %VCS_HOME%\bin\FileOnOff:

```
C:\> md %VCS_HOME%\bin\FileOnOff
```

2 Use %VCS_HOME%\bin\VCSdefault.dll as the agent DLL.
Copy the file to %VCS_HOME%\bin\FileOnOff.dll, as shown below:

```
C:\> copy %VCS_HOME%\bin\VCSdefault.dll
                %VCS_HOME%\bin\FileOnOff\FileOnOff.dll
```

3 Implement the `online`, `offline`, and `monitor` entry points using scripts. The examples below use shell scripts. MKS Toolkit or equivalent software is

required for running shell scripts. Entry points can also be implemented as batch files.

a Using any editor, create the file%VCS_HOME%\FileOnOff\online.sh with the following contents:

```
# Create the file specified by the PathName
# attribute.
    touch $2
```

b Create the file %VCS_HOME%\bin\FileOnOff\offline.sh with the following contents:

```
# Remove the file specified by the PathName
# attribute.
rm $2
```

c Create the file %VCS_HOME%\bin\FileOnOff\monitor.sh with the following contents:

```
# Verify that file specified by PathName
# attribute exists.
if test -f $2
then exit 110;
else exit 100;
fi
```

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 Windows

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.

Note: You must first install Visual C++ on the system on which the agent will be built.

Example: implementing agent using VCSAgStartup and script entry points

1 Create a directory for the agent source files VCS_HOME%\src\agent\FileOnOff:

```
C:\> md %VCS_HOME%\src\agent\FileOnOff
```

2 Copy the contents of the sample agent directory to the directory you created in the previous step:

```
C:\> copy %VCS_HOME%\src\agent\Sample\*
          %VCS_HOME%\src\agent\FileOnOff
```

3 Change to the new directory:

```
C:\> cd %VCS_HOME%\src\agent\FileOnOff
```

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

```
void VCSAgStartup() {
    VCSAgInitEntryPointStruct(V50);

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

    VCSAgSetLogCategory(10041);
    }
```

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

```
C:\> nmake
```

6 Create the following directory %VCS_HOME%\bin\FileOnOff:

```
C:\> md %VCS_HOME%\bin\FileOnOff
```

7 Place the sample.dll as the FileOnOff agent library.

```
C:\> copy sample.dll
        %VCS_HOME%\bin\FileOnOff\FileOnOff.dll
```

8 Implement the `online`, `offline`, and `monitor` entry points.
See "Example: Using C++ entry points on Windows" on page 121.

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

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

Note: To build an agent, you must first install Visual C++ on the system on which the agent will be built.

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

1 Create a directory for the agent:

```
C:\> md %VCS_HOME%\src\agent\FileOnOff
```

2 Copy the contents of the sample agent directory to the directory you created in the previous step:

```
C:\> copy %VCS_HOME%\src\agent\Sample\* %VCS_HOME%\src\
        agent\FileOnOff
```

3 Change to the new directory:

```
C:\> cd %VCS_HOME%\src\agent\FileOnOff
```

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

```
void VCSAgStartup()
   {
       VCSAG_LOG_INIT("VCSAgStartup");
       VCSAgSetLogCategory(10051);
       VCSAgInitEntryPointStruct(V50);

       VCSAgSetEntryPoint(VCSAgEPMonitor, res_monitor);
}
```

5 Modify the `res_monitor()` function:

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

       VCSAgResState res_monitor(const char *res_name, void
                  **attr_val,int *conf_level) {
           // Initialize the OUT parameters.
           VCSAgResState state;
           WCHAR **new_args = NULL;
           DWORD ret, attrs;
           VCSAG_LOG_INIT("res_monitor");
           VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,
               L"Inside monitor.";
           ret = VCSAgGetEncodedArgList(VCSAgUTF8, attr_val,
           VCSAgUCS2,
               (void ***)&new_args);
    if (ret)
    {
            VCSAG_LOG_MSG(VCS_NOTICE, 1001, VCS_DEFAULT_FLAGS,
                          L"Unable to get the arguments");
            state = VCSAgResUnknown;
            goto exit;
    }

    attrs = GetFileAttributes(new_args[0]);
    if (attrs == 0xffffffff)
    {
            state = VCSAgResOffline;
```

```
                *conf_level = 0;
        }
        else
        {
                state = VCSAgResOnline;
                *conf_level = 100;
        }
        VCSAG_RES_LOG_MSG(VCS_DBG1, VCS_DEFAULT_FLAGS,  L"Monitor
              successful.");

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

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

```
C:\> nmake
```

7 Create the following directory:
C:\Program Files\VERITAS\cluster server\bin\FileOnOff:

```
C:\> md %VCS_HOME%\bin\FileOnOff
```

8 Place the sample.dll as the FileOnOff agent library.

```
C:\> copy sample.dll
        %VCS_HOME%\bin\FileOnOff\FileOnOff.dll
```

9 Implement the `online` and `offline` entry points.
See “Example: Using C++ entry points on Windows” on page 121.

Installing the custom agent

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

On Windows:

- %VCS_HOME%\bin\custom_type\
- A user-defined directory. For example \myagents\custom_type\. Note that you must configure the AgentDirectory attribute for this option.

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.

VCS looks for the agent binary and script entry points in these directories in the following order:

On Windows:

- Directory specified in the AgentDirectory attribute
- %VCS_HOME%\bin\custom_type\

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 Windows:

```
include types.cf
.
.
.
    FileOnOff temp_file01 (
    PathName = "C:\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_file01`
- The name of the attribute, `PathName`
- The value for the `PathName` attribute:
  `On Windows: "C:\tmp\test"`

When the resource temp_file01 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 system. 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, 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

You can test agents using:

- The engine
  (See "Using the engine process to test agents" on page 135)
- The `AgentServer` utility
  (See "Using the AgentServer utility to test agents" on page 136)

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

- Build the agent DLL and place it in the directory %VCS_HOME%\bin*resource_type*.
- Install script entry points in the directory %VCS_HOME%\bin*resource_type*.

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

- Defined the resource type in *agent*Types.cf, defined the resources in `main.cf`, and restarted the engine. You may define the new type and resources using commands from the command line.

Using debug messages

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

%VCS_HOME%\log*resource_type*_A.txt

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

Using the engine process to test agents

When the engine process (HAD) 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 engine process is active, type the following command at the system prompt to verify that the agent has been started and is running:

```
haagent -display <resource_type>
```

For example, to test the Oracle agent, type:

```
haagent -display Oracle
```

If the Oracle agent is running, the output resembles:

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

Test commands

The following examples show how 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:
  ```
  haatype -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 VCS engine process. The utility is part of the product package and is installed in the directory

Windows: %VCS_HOME%\bin

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

To start AgentServer and test the FileOnOff agent (Windows)

1. Type the following command to start AgentServer:
   `%VCS_HOME%\bin\AgentServer`
   AgentServer must monitor a TCP port for messages from the agents. This port number can be configured by setting `vcstest` to the selected port number in the file C:\WINNT\system32\drivers\etc\services. If `vcstest` is not specified, AgentServer uses the default value 14142.
2. Start the agent for the resource type:
   `>startagent FileOnOff %VCS_HOME%\bin\VCSAgDriver.exe`
   . You will receive the following messages:
   ```
   Agent (FileOnOff) has connected.
   Agent (FileOnOff) is ready to accept commands.
   ```
3. Review the sample configuration:
   ```
   types.cf:

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

   main.cf:
       ...
       group ga (
       ...
       )
       FileOnOff file1 (
           Enabled = 1
           PathName = "C:\\VRTSvcsfile001"
       )
   ```
4. Refer to the types.xml and main.xml configuration files when testing the FileOnOff agent. In the next step, the sample configuration is set up using AgentServer commands.

5 Complete step n through step n 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
    C:\\VRTSvcsfile001

>modifyres FileOnOff file1 Enabled int 1
```

6 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. A status message similar to the example in step n is displayed when file1 is taken offline.

7 View the list of agents started by the AgentServer process:

```
>print
```

Output resembles:

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

8 Stop the agent:

```
>stopagent FileOnOff
```

9 Exit from the AgentServer:

```
>quit
```

Chapter 8

Static type attributes

- About static attributes
- Static timeout attributes on Windows
- Static type attribute definitions

About static attributes

Predefined static resource type attributes apply to all resource types.

See “Static type attribute definitions” on page 142

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.

Static timeout attributes on Windows

Windows does not support a safe method for one thread to terminate another thread running arbitrary code. Therefore, agents on Windows wait until the entry points complete. The entry point timeouts listed in this chapter *do not* apply to Windows.

The following entry point timeouts *do not* apply for agents on Windows:

- ActionTimeout
- AttrChangedTimeout
- CloseTimeout
- CleanTimeout
- FaultOnMonitorTimeout
- InfoTimeout
- MonitorTimeout,
- OfflineTimeout
- OnlineTimeout
- OpenTimeout.

On the Windows platform, agents do not execute entry point timeouts, but wait until the entry points complete. For information concerning the timeouts listed above:

See the *Veritas Cluster Server Administrator's Guide*.

Static type attribute definitions

The following sections describe the static attributes for agents.

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.

.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).

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 the engine from restarting the resource indefinitely. The engine 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.

LogDbg

LogDbg is a type-level attribute that specifies which debug messages originating from the agent for that type are to be logged.

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.

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 engine 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]
```

You can override the LogDbg attribute. For example, for a specific critical resource, this attribute's value can be set to obtain more debug messages for the resource by adding more debug severities than those already set for the resource's type. From the command line, this can be done using the `hares` command. For example:

```
# hares -override f1 LogDbg
# hares -modify f1 LogDbg -add DBG_5
# hares -display f1 -attribute LogDbg
Resource      Attribute        System         Value
f1            LogDbg           global         DBG_3 DBG_4 DBG_5
```

Note that once LogDbg is overridden, you have to use the 'hares' command to display the value of the LogDbg attribute for resource f1. The 'hatype' command will display the value for the entire type.

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

The LogDbg attribute value can be overridden.

Note: Values of LogDbg overridden for a resource are effective only if the agent uses VCSAG_RES_LOG_MSG, the only API that checks if a particular debug severity is enabled for the resource before writing the message to the log file. Refer to Chapter 5, "Logging agent messages" on page 99 for more information.

LogFileSize

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

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

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

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

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

ManageFaults

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

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

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

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

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

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

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

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

MinCompatAgentVersion

Specifies the lowest compatible version of the agent that is supported work with the current version of the type definition. The version specified in this attribute must be lower than or equal to the version specified in the TypeDefinitionVersion attribute. All versions of the agent binary starting from the version mentioned in this attribute upto the version given in the TypeDefinitionVersion attribute are assumed to be compatible with the type-definition.

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 the engine. 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*.

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 the engine 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).

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.

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 the engine 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.

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 144 and

“ToleranceLimit” on page 153.

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 engine 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.

SupportedActions

The SupportedActions (string-keylist) attribute lists all possible actions defined for an agent, including those defined by the agent developer. The engine validates the *action_token* value specified in the `hares -action` *`resource action_token`* command against the SupportedActions attribute. For example, if action_token is not present in SupportedActions, the engine 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 30.

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.

TypeDefinitionVersion

Specifies the version of the type-definition for the custom type. If the agent code is modified or if the type-definition is changed in any way, then this attribute needs to be updated to a newer version. This attribute allows users to keep a track of the version of the type-definition for the type.

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

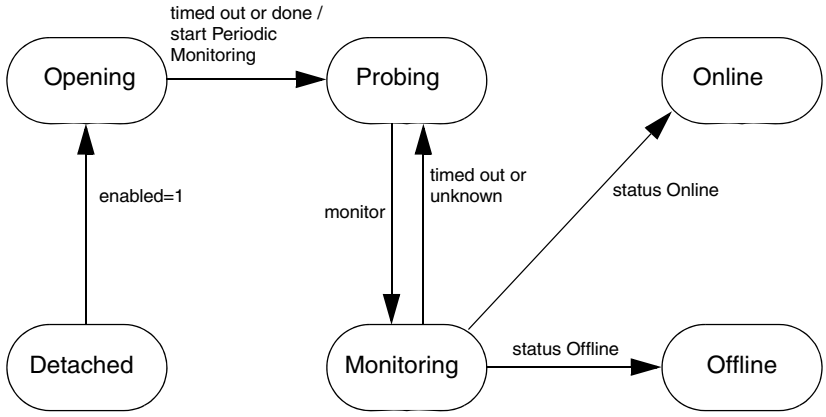
In addition, state transitions are shown for the handling of resources with respect to the `ManageFaults` service group attribute.

See "The next set of diagrams illustrate the following state transitions:" on page 164.

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.

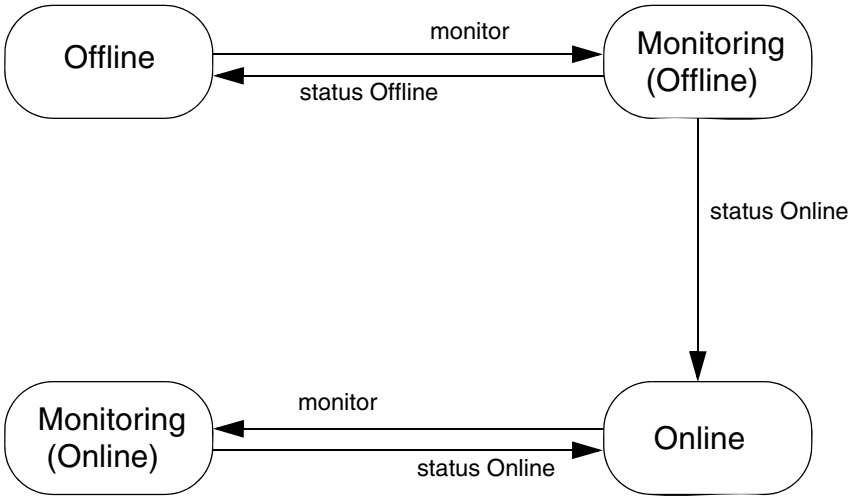
Opening a resource



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.

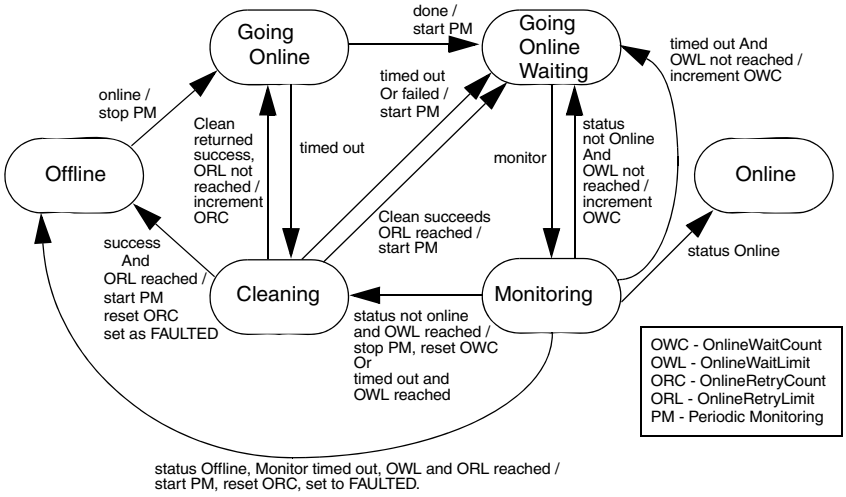
When the resource is enabled (Enabled=1), the `open` entry point is invoked. Periodic Monitoring begins after *`open`* times out or succeeds. Depending on the return value of *`monitor`*, the resource transitions to either the Online or the Offline state. In the unlikely event that *`monitor`* times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

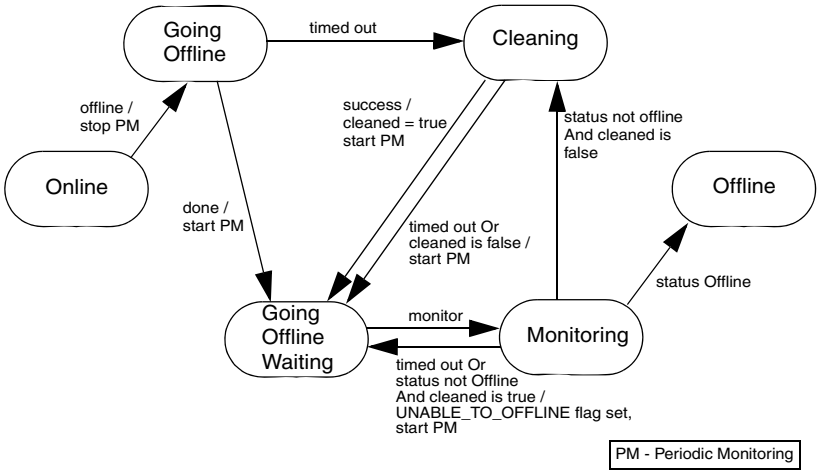
If *monitor* returns a status of online, the resource moves to the Online state.

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.

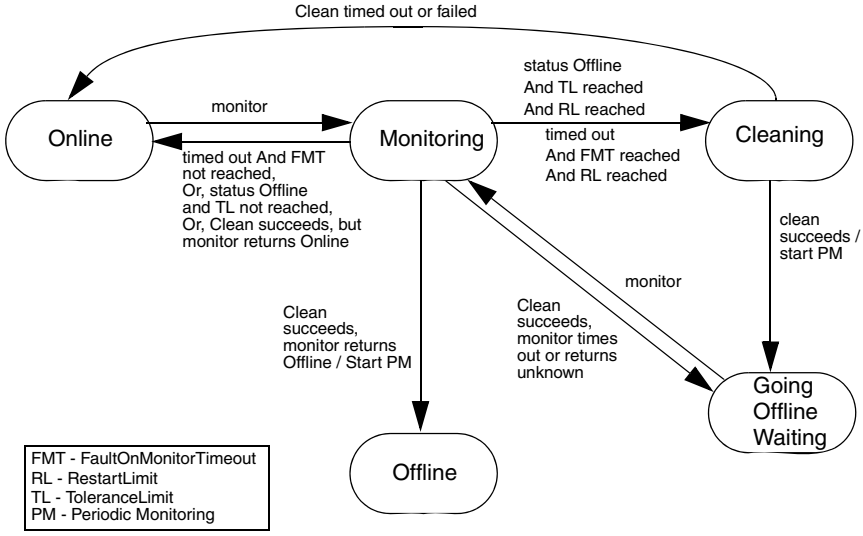
If *offline* succeeds, the resource enters the Going Offline Waiting state where it waits for the next monitor.

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.

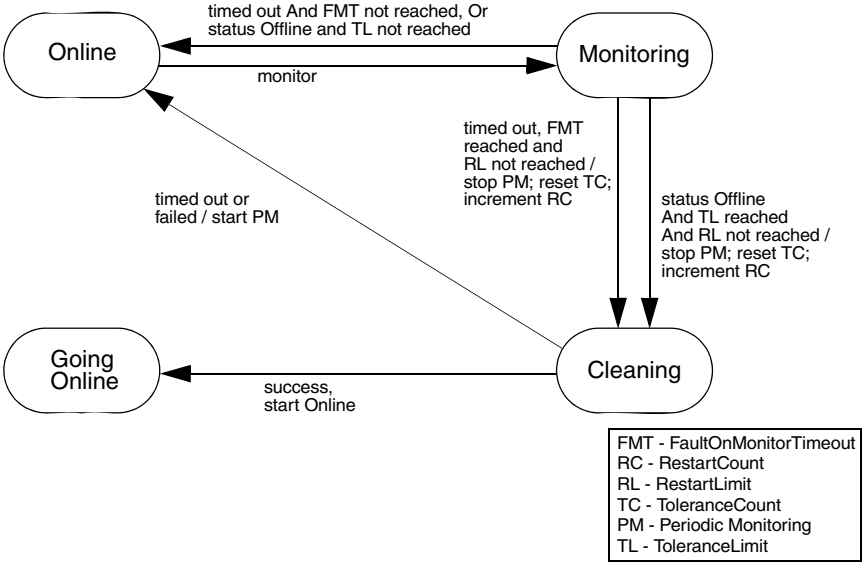
Note: For the Windows platform, the `FaultOnMonitorTimeout` attribute has no significance. Instead, the `monitor` entry point is allowed to continue to completion.

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

If *`clean`* succeeds, the resource is placed in the Going Offline Waiting state, where the agent waits for the next monitor.

- If *`monitor`* reports Online, the resource is placed back Online as if no fault occurred. If *`monitor`* reports Offline, the resource is placed in an Offline state and marked as FAULTED.
- If *`monitor`* reports unknown or times out, the agent places the resource back into the Going Offline Waiting state, and sets the UNABLE_TO_OFFLINE flag in the engine.

Resource fault with automatic restart



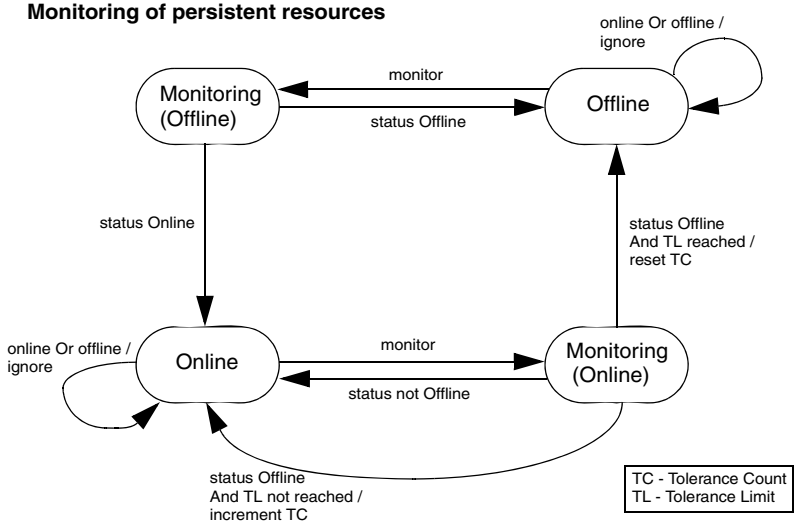
This diagram describes the activity that occurs when a resource faults and the `RestartLimit` is not reached. When the *`monitor`* entry point times out successively and `FaultOnMonitorTimeout` is reached, or `monitor` returns offline and the `ToleranceLimit` is reached, the agent invokes the *`clean`*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

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

- If *`clean`* succeeds, the resource is placed in the Going Online state and the *`online`* entry point is invoked to restart the resource; refer to the diagram, "Bringing a resource online."
- If *`clean`* fails or times out, the agent places the resource in the Online state as if no fault occurred.

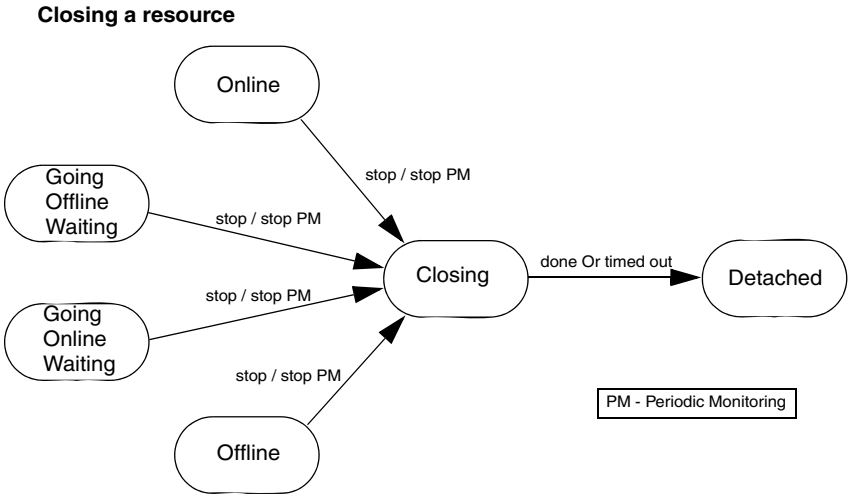
Refer to the diagram "Resource fault without automatic restart," for a discussion of activity when a resource faults and the `RestartLimit` is reached.

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:

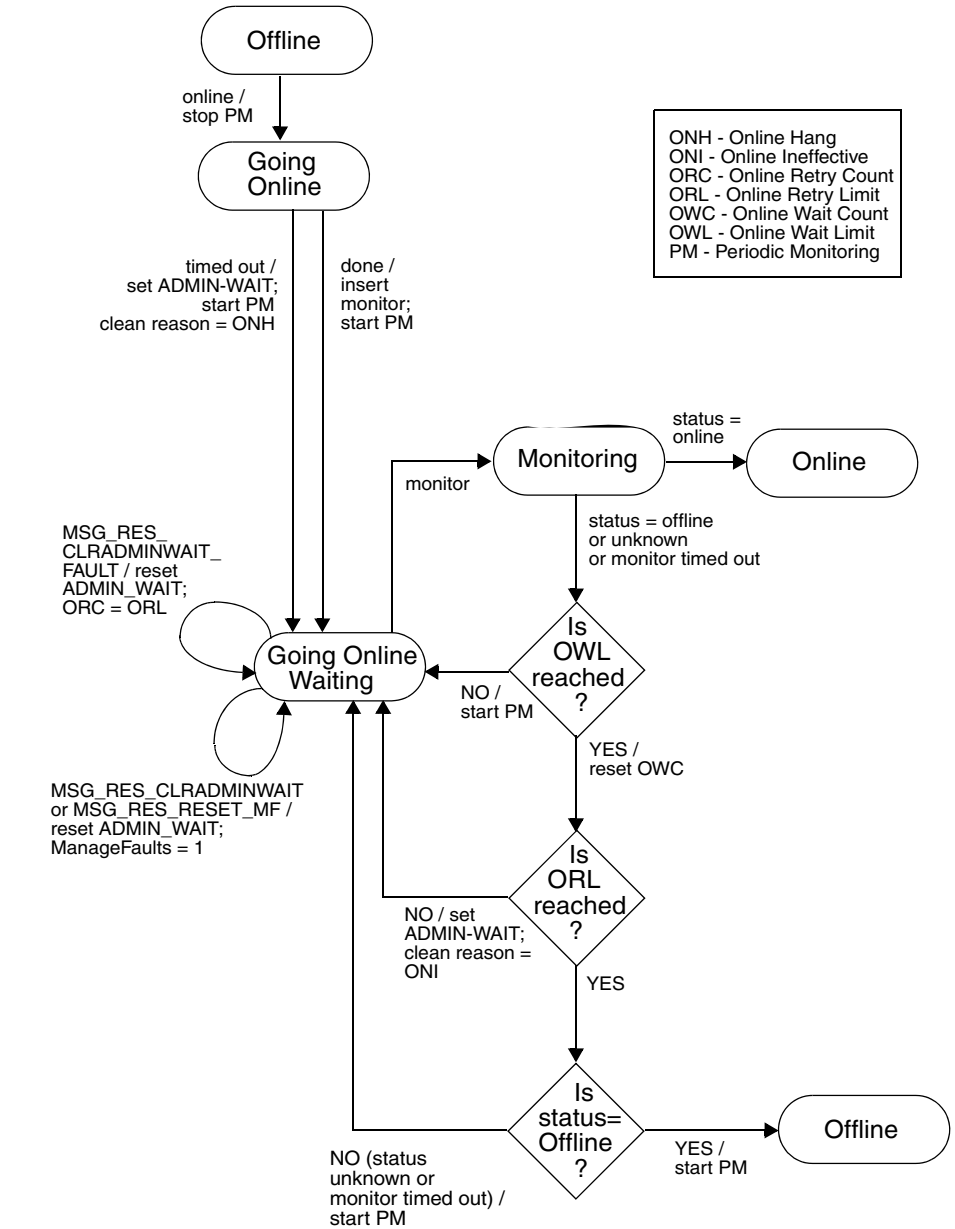
State transitions with respect to ManageFaults attribute

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

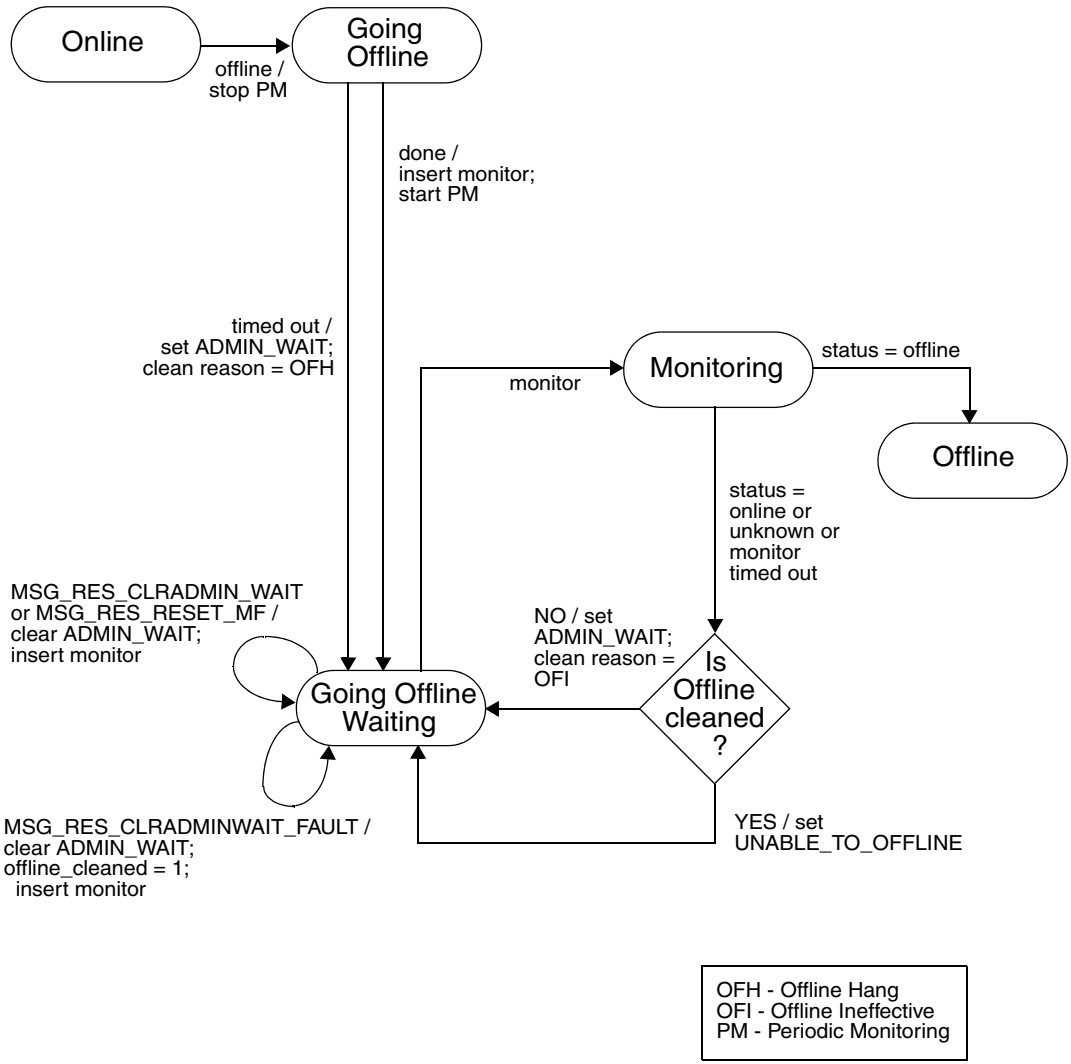
By default, `ManageFaults` is set to ALL, in which case the clean entry point is called by VCS. See "ManageFaults" on page 147.The diagrams cover the following conditions:

- 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

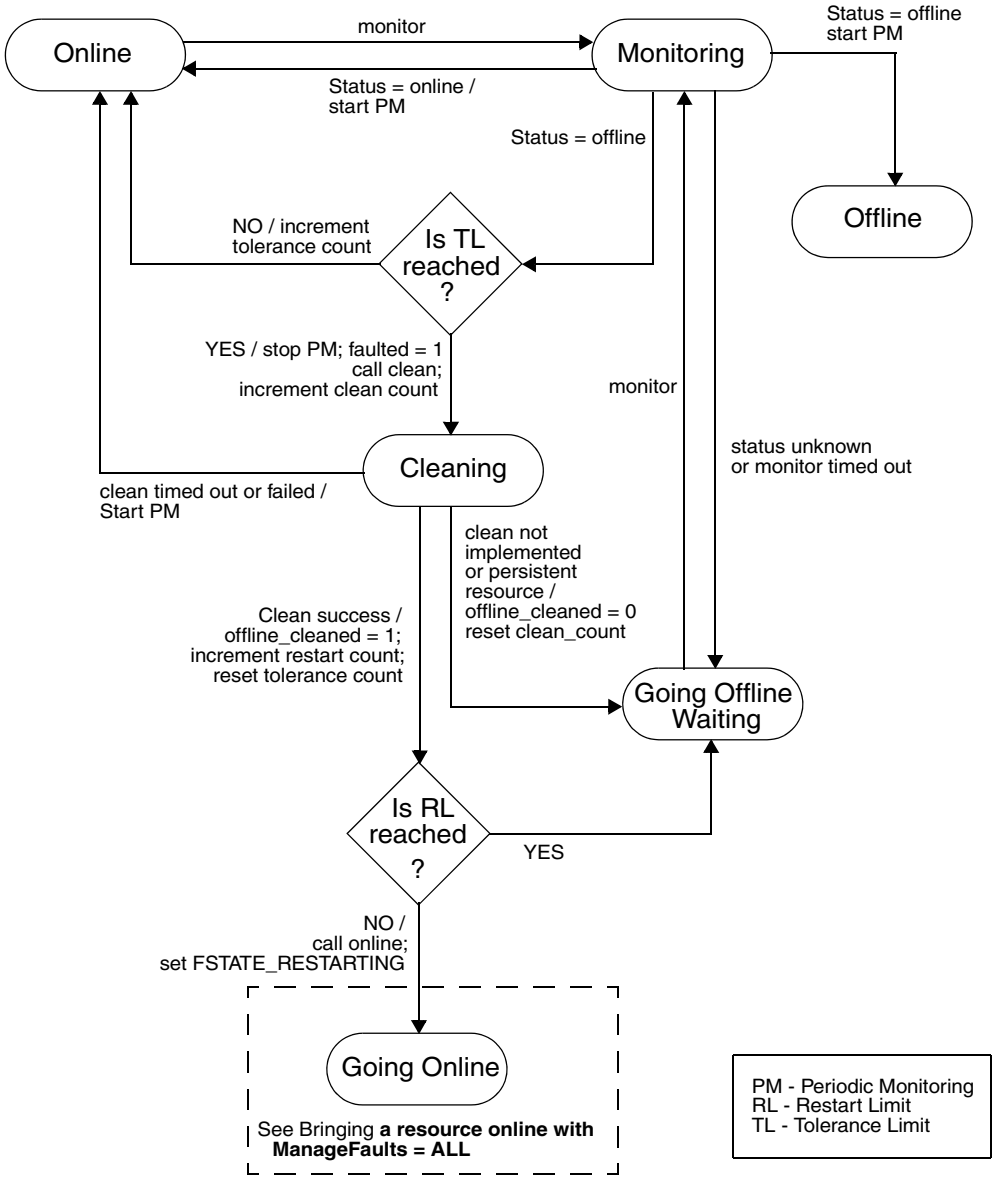
Bringing a resource online: ManageFaults attribute = NONE



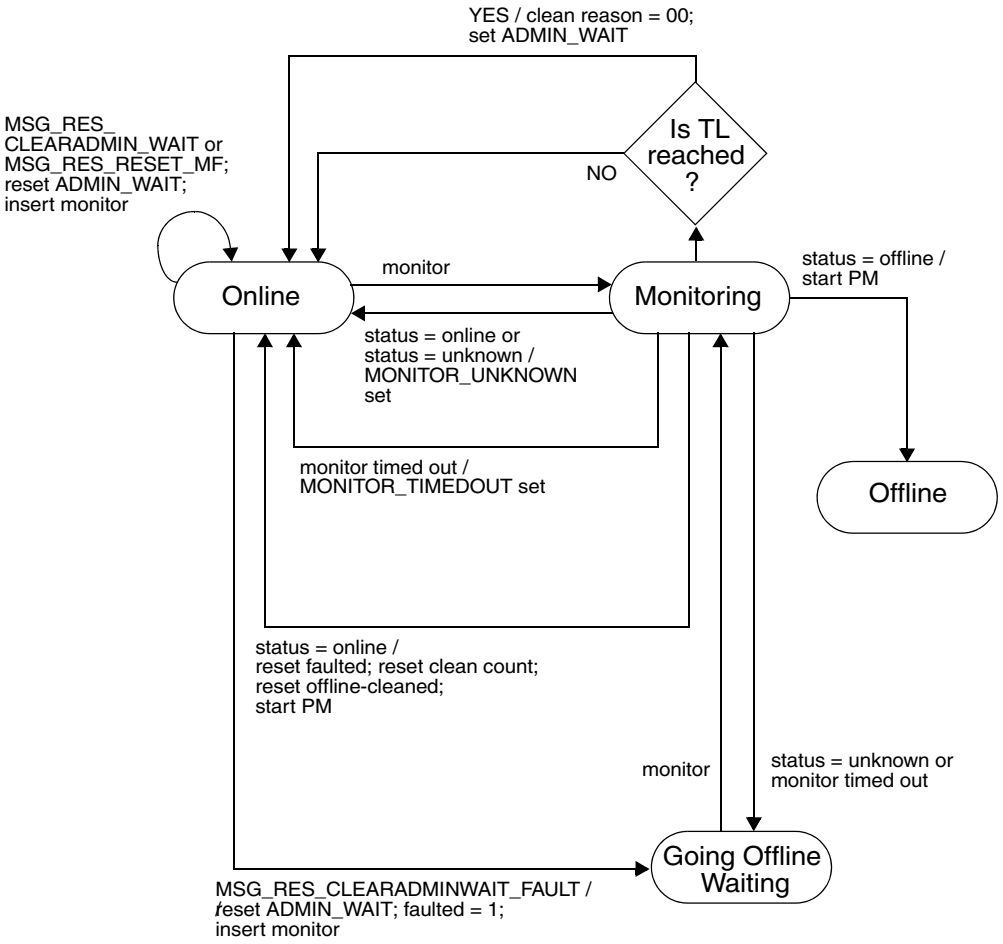
Taking a resource offline; ManageFaults = None



Resource fault: ManageFaults attribute = ALL

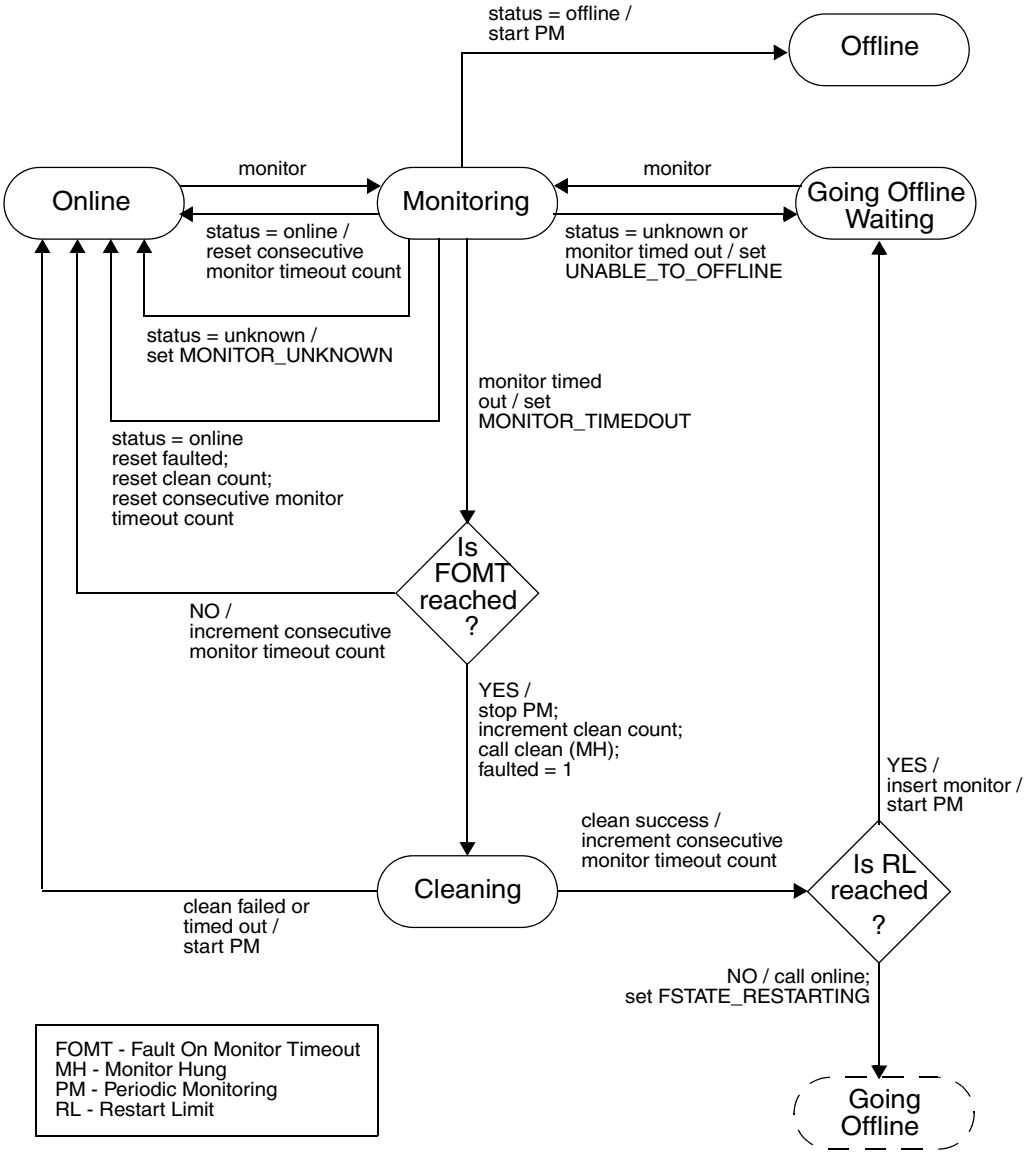


Resource fault (unexpected offline): ManageFaults attribute = NONE

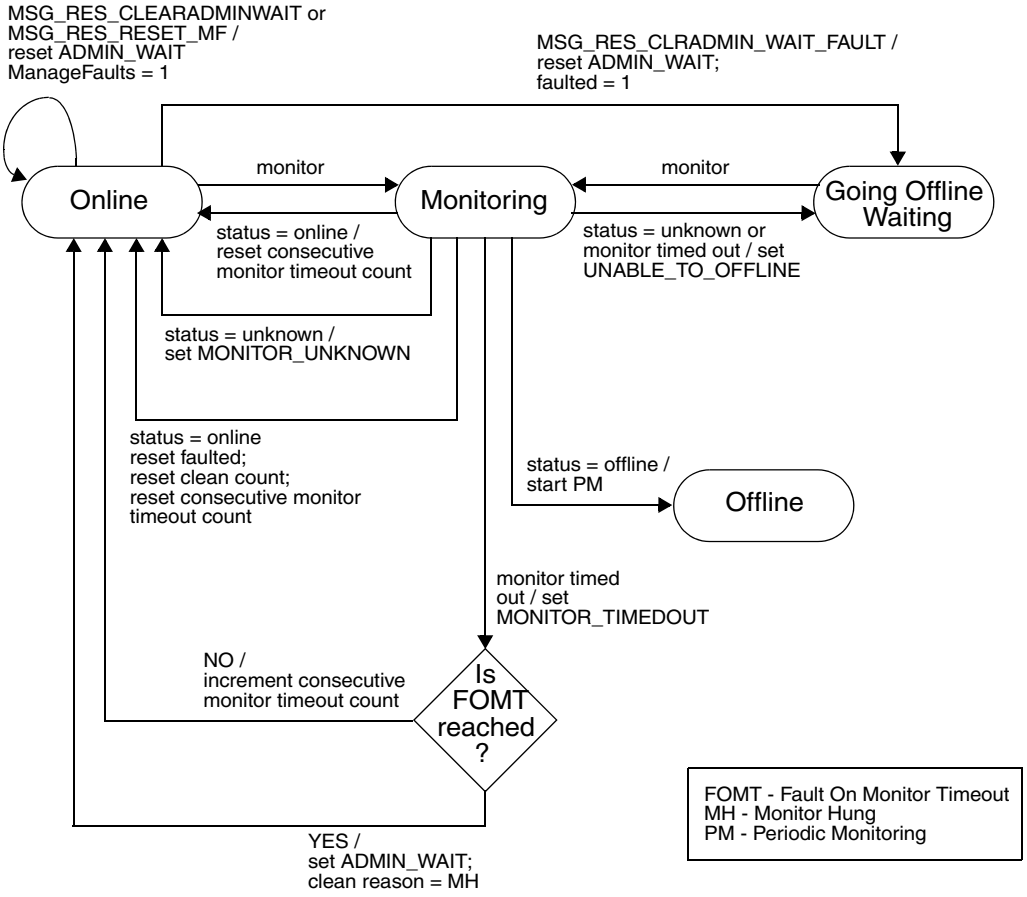


PM - Periodic Monitoring
TL - Tolerance Limit

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:

`VRTSvcsW2KAgent.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 175 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:

```
VRTSvcsW2KOracle.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 VRTSvcsW2KAgent.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 VRTSvcsW2KAgent.smc
```

Storing BMC files

By default, BMC files must be installed in C:\program files\VERITAS\cluster server\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 resembles:

```
%VCS_HOME%\messages\en\VRTSvcsW2KAgent.bmc.
```

VCS languages

The languages supported by VCS are listed as subdirectories in the directory %VCS_HOME%\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 VRTSvcsW2KAgent.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: %VCS_HOME%\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:

   ```
   copy VRTSvcsW2KOracle.bmc %VCS_HOME%\messages\en
   ```

2. Change to the directory containing the BMC file and run the `bmcmap` utility. For example:

   ```
   bmcmap -create en HAD
   cd %VCS_HOME%\messages\en
   bmcmap -create en HAD
   ```

The `bmcmap` utility scans the contents of the directory and dynamically generates the BMC map. In this case, HAD.bmcmap is created.

Example BMC Map File

In the following example, a BMC named `VRTSvcsHPW2KNewCustomAgent.bmc` is included in the BMC map file for the HAD module and the English language.

```
#
# Copyright(C) 2008 Symantec Corporation.
# ALL RIGHTS RESERVED.
#

Language=en

HAD=VRTSvcsHad VRTSvcsHPW2KAgent VRTSvcsHPW2KNewCustomAgent
```

```
# VCS
VRTSvcsHad.version=1.0
VRTSvcsHad.IDstart=0
VRTSvcsHad.IDend=16501
VRTSvcsHad.Category=__________

# HPW2K Bundled Agents
VRTSvcsHPW2KAgent.version=1.0
VRTSvcsHPW2KAgent.IDstart=100001
VRTSvcsHPW2KAgent.IDend=113501
VRTSvcsHPW2KAgent.Category=__________


# HPW2K NewCustomAgent
VRTSvcsHPW2KNewCustomAgent.version=1.0
VRTSvcsHPW2KNewCustomAgent.IDstart=2017001
VRTSvcsHPW2KNewCustomAgent.IDend=2017040
VRTSvcsHPW2KNewCustomAgent.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 V50
- 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 V50

With VCS 5.0, the agent framework has been enhanced. Agents you develop to use this framework are registered as V50 agent. 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 V50 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 the cluster.
- 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 V50

- Modifications to PATH variables and links to the VCS Script50Agent binary (on UNIX) may be necessary.

- Change the way attributes and their values are passed to the entry points from the V40 format to V50 name-value tuple format.
- On Windows. include the VCS libraries in path for Perl and shell to source them.
- Include /opt/VRTSvcs/lib in path for Perl and shell to source them.
- Set necessary environment variables.

Overview of V50 name-value tuple format

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 V50 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 V50 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 V50

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;
```

V50

```
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.
  - Define the entry points in the VCS primitive, `VCSAg40EntryPointStruct`, and use the primitive `VCSAgRegisterEPStruct` to register the agent as a VCS 4.0 agent.
  - Recompile the agent.

Note: Agents developed on the 4.0 and later agent framework are not compatible with the 2.0 or the 3.5pre-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 99.

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 99.

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

Windows: %VCS_HOME%\log\resource_type_A.txt.

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

Windows: %VCS_HOME%\log\resource_type_A.txt.

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

Windows: %VCS_HOME%\log\resource_type_A.txt.

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

Windows: %VCS_HOME%\log\resource_type_A.txt.

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

Windows: %VCS_HOME%\log\resource_type_A.txt.

The `message` must not exceed 4096 bytes. A message greater that 4096 bytes is truncated. The size of all argument strings combined must not exceed 4096 bytes. If the argument string total exceeds 4096 bytes, then each argument is allowed an equal portion of 4096 bytes and truncated if it exceeds the allowed portion.

`tag` can be any value from TAG_A to TAG_Z. Tags A through E are enabled by default. To enable other tags, use the `halog` command. `flags` can be zero or more of `LOG_NONE`, `LOG_TIMESTAMP` (prints date and time), `LOG_NEWLINE` (prints a new line), and `LOG_TAG` (prints tag). This primitive can be called from any entry point.

For example:

```
#include "VCSAgApi.h"
...
...
char buffer[256];
sprintf(buffer, "VCS:2015003:Oracle:%s:%s:During scan for
    process %s ioctl failed with return code %s, errno = %s",
res_name, ep_name, proc_name, ret_buf, err_buf);

VCSAgLogI18NConsoleMsgEx(TAG_A, buffer, 2015003, res_name,
ep_name, proc_name, ret_buf, err_buf, NULL, flags);
```

Index

A

action entry point
  C++ syntax 59
  script entry point 93
  supported actions 153
agent framework
  described 13
  library, C++ 34
  logging APIs 102
  multithreaded 44
  Windows primitives 79
  working with pre-4.0 agents 185
agent messages
  for Windows 79
  formatting 101
  normal in VCSAG_LOG_MSG 114
ArgList parameter 142
ArgList reference attributes 142
association attribute dimension 17
attr_changed entry point 31
  C++ syntax 61
  script syntax 94
attribute dimensions
  association 17
  keylist 17
  scalar 16
  vector 17
attribute types
  boolean 16
  integer 16
  string 16
attributes
  local and global 18

B

binary message catalog (BMC) files
  converting from SMC files 177
  displaying contents 177
  updating 179
bmcgen utility 177
bmcread utility 177
boolean attribute type 16

C

category ID for messages 112
clean entry point 28
  C++ syntax 58
  enum types 28
  script syntax 93
close entry point 32
  C++ syntax 64
  script syntax 95
configuration language
  local and global attributes 18
ConfInterval parameter 143

D

debug message severity level 104
debug messages
  C++ entry points 104
  Perl script entry points 115
  Shell script entry points 115

E

entry points
  attr_changed 31
  clean 28
  close 32
  definition 14
  info 24
  monitor 23
  offline 27
  online 26
  open 31
  sample structure 35
  shutdown 32
enum types for clean
  VCSAgCleanMonitorHung 29
  VCSAgCleanOfflineHung 28
  VCSAgCleanOfflineIneffective 28
  VCSAgCleanOnlineHung 28

VCSAgCleanOnlineIneffective 29
VCSAgCleanUnexpectedOffline 29

F

FaultOnMonitorTimeouts parameter 144
FireDrill parameter 144
formatting agent messages 101

G

global attributes 18

H

had 13
high-availability daemon (had) 13

I

info entry point 24
C++ syntax 50
script example 94
InfoTimeout parameter 145
initializing functions with VCSAG_LOG_INIT 106
integer attribute type 16

K

keylist attribute dimension 17

L

local attributes 18
log category 107
LogDbg parameter 145
LogFileSize parameter 146
logging APIs
C++ 102
script entry points 112

M

ManageFaults parameter 147
message text format 101, 102
mnemonic message field 101
monitor entry point 23
C++ syntax 49, 50
script syntax 92
MonitorLevel parameter 148
MonitorStatsParam parameter 148

O

offline entry point 27
C++ syntax 57
script syntax 92
OfflineMonitorInterval parameter 149
online entry point 26
C++ syntax 56
script syntax 92
OnlineRetryLimit parameter 149
OnlineWaitLimit parameter 150
OnOff resource type 14
OnOnly resource type 14
open entry point 31
C++ syntax 63
script syntax 95

P

parameters
ArgList 142
ConfInterval 143
FaultOnMonitorTimeouts 144
FireDrill 144
InfoTimeout 145
LogDbg 145
LogFileSize 146
ManageFaults 147
MonitorLevel 148
MonitorStatsParam 148
OfflineMonitorInterval 149
OnlineRetryLimit 149
OnlineWaitLimit 150
RegList 151
RestartLimit 152
ToleranceLimit 153
persistent resource type 14
primitives
definition 66
logging primitives for Windows 79
VCSAgAllocMem 83
VCSAgDelEncodedArgList 81
VCSAGDeleteMem 83
VCSAgGetCookie 70
VCSAgGetEncodedArglist 81
VCSAgLogI18NMsg 189, 191, 192
VCSAgLogI18NMsgEx 190
VCSAgLogMsg 188
VCSAgRegister 68
VCSAgRegisterEPStruct 66

VCSAgSetCookie 67
VCSAgSnprintf 71
VCSAgSnprintfW 82
VCSAgStrlcat 71
VCSAgStrlcatW 82
VCSAgStrlcpyW 81
VCSAgUnregister 69

R

RegList parameter 151
resource
closing (state transition diagram) 164
fault (state transition diagram) 161, 162
monitoring (state transition diagram) 163
offlining (state transition diagram) 160
onlining (state transition diagram) 159
OnOff type 14
OnOnly type 14
opening (state transition diagram) 157
persistent type 14
steady state (state transition diagram) 158
RestartLimit parameter 152

S

scalar attribute dimension 16
ScriptAgent 185
script-based logging functions 111
severity macros 105
severity message field 101
Shell scripts
definition 90
using on Windows 90
shutdown entry point 32
C++ syntax 65
script syntax 95
source message catalog (SMC) files
converting to BMC files 174
creating 174
state transition diagram
closing a resource 164
monitoring persistent resources 163
offlining a resource 160
onlining a resource 159
opening a resource 157
resource fault with auto restart 162
resource fault, no auto restart 161
resource in steady state 158
string attribute type 16

T

timestamp message field 101
ToleranceLimit parameter 153

U

UCS-2 encoding (wchar)
enabling agents with 80
primitives for converting to 81
UMI (unique message identifier) 101

V

VCSAG_CONSOLE_LOG_MSG logging macro 102
VCSAG_LOG_INIT initializing function 106
VCSAG_LOG_MSG logging macro 102
VCSAG_LOG_MSG script logging function 111
VCSAG_LOGDBG_MSG logging macro 102
VCSAG_LOGDBG_MSG script logging function 111
VCSAG_RES_LOG_MSG logging macro 102
VCSAG_SET_ENVS script logging function 111
VCSAgAllocMem primitive 83
VCSAgDelEncodedArgList primitive 81
VCSAgDeleteMem primitive 83
VCSAgGetCookie primitive 70
VCSAgGetEncodedArglist primitive 81
VCSAgLogI18NMsg primitive 189, 191, 192
VCSAgLogI18NMsgEx primitive 190
VCSAgLogMsg primitive 188
VCSAgRegister primitive 68
VCSAgRegisterEPStruct primitive 66
VCSAgSetCookie primitive 67
VCSAgSnprintf primitive 71
VCSAgSnprintfW primitive 82
VCSAgStartup entry point, C++ syntax 48
VCSAgStrlcat primitive 71
VCSAgStrlcatW primitive 82
VCSAgStrlcpyW primitive 81
VCSAgUnregister primitive 69
vector attribute dimension 17