Cluster Server 7.4.2 Agent Developer's Guide - AIX, Linux, Solaris and Windows

VERITAS™

Last updated: 2020-06-01

Legal Notice

Veritas Technologies LLC
2625 Augustine Drive
Santa Clara, CA 95054
http://www.veritas.com

Technical Support

Technical Support maintains support centers globally. All support services will be delivered in accordance with your support agreement and the then-current enterprise technical support policies. For information about our support offerings and how to contact Technical Support, visit our website:
https://www.veritas.com/support

You can manage your Veritas account information at the following URL:
https://my.veritas.com

If you have questions regarding an existing support agreement, please email the support agreement administration team for your region as follows:

Worldwide (except Japan)	CustomerCare@veritas.com
Japan	CustomerCare_Japan@veritas.com

Documentation

Make sure that you have the current version of the documentation. Each document displays the date of the last update on page 2. The latest documentation is available on the Veritas website:
https://sort.veritas.com/documents

Documentation feedback

Your feedback is important to us. Suggest improvements or report errors or omissions to the documentation. Include the document title, document version, chapter title, and section title of the text on which you are reporting. Send feedback to:
infoscaledocs@veritas.com

You can also see documentation information or ask a question on the Veritas community site:
http://www.veritas.com/community/

Veritas Services and Operations Readiness Tools (SORT)

Veritas Services and Operations Readiness Tools (SORT) is a website that provides information and tools to automate and simplify certain time-consuming administrative tasks. Depending on the product, SORT helps you prepare for installations and upgrades, identify risks in your datacenters, and improve operational efficiency. To see what services and tools SORT provides for your product, see the data sheet:
https://sort.veritas.com/data/support/SORT_Data_Sheet.pdf

Contents

Chapter 1

Introduction

This chapter includes the following topics:

About VCS agents

Agents are programs that manage resources, such as a disk group or a mount point, 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 them online, monitoring their state, or taking them offline.

VCS agents are classified as follows:

- Bundled agents
  These agents are part of the VCS infrastructure and are packaged along with VCS. Examples of bundled agents include the IP (Internet Protocol) and NIC (network interface card) agents. For more information on VCS bundled agents, including their attributes and modes of operation, see the *Cluster Server Bundled Agents Reference Guide*.
- Enterprise agents
  These agents manage enterprise databases including Oracle, Sybase, and DB2, and are packaged along with VCS.
- High availability agents
  High availability agents manage various applications and are available through a release vehicle called Agent Pack. The Agent Pack is released every quarter independent of the VCS release. The agents are classified based on the application type as follows:

- Application agents
  These agents manage enterprise applications, such as WebLogic, WebSphere, and SAP.
- Database agents
  These agents manage database applications, such as MySQL, SAP MaxDB, and Informix.
- Replication agents
  These agents manage hardware and software replication technologies, such as SRDF and HP EVACA.

The Agent Pack is available as a free download from the following locations:

- Veritas Services and Operations Readiness Tool (SORT)
  The agents are available as individual tarballs from:
  https://sort.veritas.com/agents

- Custom agents: These are agents that are developed outside of Veritas and are not supported by Veritas Technical Support.

How agents work

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

As part of the VCS processes, a high availability daemon (HAD) is responsible for making applications highly available on a system.

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

The VCS HAD process provides the agents with the specific configuration information for the resources that are configured under VCS.

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

About the agent framework

The agent framework is a set of predefined functions compiled into the agent for each resource type. These functions include the ability to connect to the VCS engine and to understand the common configuration attributes, such as RestartLimit and MonitorInterval. When an agent is built using C++, the agent framework is compiled in the agent with an include statement. When an agent is built using script languages,

such as shell, Perl, or Python, the user can use the different agent binaries on UNIX shipped with VCS that provides framework functions. The agent framework handles much of the complexity that need not concern the agent developer.

Note: Python scripts are supported only on the Linux and the Windows platforms.

See "Script based agent binaries" on page 138.

About intelligent monitoring framework (IMF)

With the IMF, VCS supports intelligent resource monitoring in addition to the poll-based monitoring. IMF is an extension to the VCS agent framework. Many VCS agents use IMF for monitoring resources. You can enable and disable the IMF functionality of the VCS agents as needed. However, IMF is enabled by default for all agents that support IMF.

The benefits of intelligent monitoring over poll-based monitoring are as follows:

- Provides faster notification of resource state changes.
- Reduces VCS system utilization due to reduced monitor function footprint, which enables VCS to effectively monitor a large number of resources.

For information on agents that support IMF in VCS, refer https://www.veritas.com/support/en_US/article.TECH223407

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 file can be considered similar to a header file in a C program. The type definition defines the attributes and their data types, and provides default values for certain attributes that affect all resources of that resource type.

For example, one of the attributes that is defined for the IP resource type is the Address attribute, which stores the IP address of a specific IP resource. This attribute is defined as a 'string-scalar'.

About agent functions (entry points)

An entry point is either a C++ function or a script (for example: shell, Perl, or Python) 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 25.

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

See "About agent entry points" on page 25.

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

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

- On-off resources
  Most resources are on-off, meaning agents start and stop them as required. For example, VCS assigns an IP address to a specified NIC when bringing a resource online and removes the assigned IP address when taking the resource offline. Another example is the DiskGroup resource. VCS imports a disk group when needed and deports it when it is no longer needed. For agents of on-off resources, all entry points can be implemented.
- On-only resources
  An on-only resource is brought online, but it is not taken offline when the associated service group is taken offline. For example, in the case of the FileOnOnly resource, the engine creates the specified file when required, but does not delete the file if the associated service group is taken offline. For agents of on-only resources, the offline entry point is not needed or invoked.
- Persistent resources
  Persistent resource has an operation value of None. It cannot be brought online or taken offline, yet the resource must be present in the configuration to allow the resource to be monitored. For example, a NIC resource cannot be started or stopped, but it is required to be operational in order for the associated IP address to function properly. The agent monitors persistent resources to ensure their status and operation. An agent for a persistent resource does not require or invoke the online or offline entry points. It uses only the monitor entry points.

About attributes

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

Resource type attributes

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

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

Attribute data types

VCS supports the following data types for attributes.

String

A string is a sequence of characters. If the string contains double quotes, the quotes must be immediately preceded by a backslash. A backslash is represented in a string as \\. Quotes are not required if a string begins with a letter, and contains only letters, numbers, dashes (-), and underscores (_). For example, a string defining a network interface such as hme0 or eth0 does not require quotes as it contains only letters and numbers. However a string defining an IP address contains periods and requires quotes, such as: "192.168.100.1"

Integer

Signed integer constants are a sequence of digits from 0 to 9. They may be preceded by a dash, and are interpreted in base 10. Integers cannot exceed the value of a 32-bit signed integer: 2147483647.

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

Attribute dimensions

VCS attributes have the following dimensions.

Scalar — A scalar has only one value.

For example:

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

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

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

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

Example snippet from the type definition file for an agent:

```
str BackupSys[]
```

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

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

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

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

Keylist

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

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

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

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

For example:

```
keylist BackupVols = {}
```

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

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

Association

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

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

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

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

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

For example:

```
int BackupSysList {}
```

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

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

Attribute scope across systems: global and local attributes

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

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

In the following example of the MultiNICA resource type, attributes applying locally are indicated by "@system" following the attribute name:

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

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

Attribute life: temporary attributes

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

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

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

You can modify attribute values only while VCS is running.

In the following example of RVGSnapshot resource type, FDFile is the temporary attribute.

```
type RVGSnapshot (
        static keylist RegList = { Prefix }
        static int NumThreads = 1
        static str ArgList[] = { RvgResourceName, CacheObj,
Prefix, DestroyOnOffline }
        str RvgResourceName
        str CacheObj
        str Prefix
        boolean DestroyOnOffline = 1
        temp str FDFile
)
```

About intentional offline of applications

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

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

This feature allows administrators to stop applications without causing failovers.

See "IntentionalOffline" on page 189.

About developing an agent

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

Considerations for the application

The application for which an agent for VCS is developed must lend itself to being controlled by the agent and be able to operate in a 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 if new agent is of type OnOff or OnOnly. There must be some means of starting the application's external resources such as file systems that store databases, or IP addresses used for listener processes, and so on.
- Each instance of an application must be capable of being stopped by a defined procedure if new agent is of type OnOff. Other instances of the application must not be affected.
- The application must be capable of being stopped cleanly, by forcible means if necessary.
- Each instance of an application must be capable of being monitored uniquely. Monitoring can be simple or in-depth so as to achieve a high level of confidence in the operation of the application. Monitoring an application becomes more effective when the monitoring procedure resembles the actual activity of the application's user.
- For failover capability, the application must be capable of storing data on shared disks rather than locally or in memory, and each system must be capable of accessing the data and all information required to run the application.
- The application must be crash-tolerant. It must be capable of being run on a system that crashes and of being started on a failover node in a known state.

This typically means that data is regularly written to shared storage rather than stored in memory.

- The application must be host-independent within a 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.
- The applications configured under VCS control must not write data on stdout and stderr stream. This may interfere with VCS agent functionality. For such applications to run under VCS control, you must redirect the application's stdout and stderr stream.

High-level overview of the agent development process

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

Creating the type definition file

The types definition file contains definitions of resource types. Place a custom resource type definition in a file that specifies the name of the custom resource; for example, MyResourceTypes.cf . This file is referenced as an "include" statement in the VCS configuration file, main.cf.

Decide about attributes, attribute types, and attribute dimension of this new agent. Based on these create the type definition file for this agent.

See “Creating the type definition file for a custom agent” on page 139.

Developing the entry points

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

See “Considerations for using C++ or script entry points” on page 39.

Create the entry points.

For more information on developing entry points, refer to the following links.

See “About creating entry points in C++” on page 49.

See “About creating entry points in scripts” on page 90.

Building the agent

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

See “Creating the type definition file for a custom agent” on page 139.

See “Files for use in agent development” on page 138.

For building an agent, sample files are provided.

Testing the agent

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

See “About testing agents” on page 171.

Chapter 2

Agent entry point overview

This chapter includes the following topics:

About agent entry points

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

Supported entry points

The agent framework supports the following entry points:

- `open` - initializes the environment for a resource before the agent starts to manage it
- `monitor` - determines the status of a resource
- `online` - brings a resource online
- `offline` - takes a resource offline
- `clean` - terminates ongoing tasks associated with an online or partially-online resource and then forcefully brings the resource offline

- `action` - starts a defined action for a resource
- `info` - provides information about an online resource
- `attr_changed` - responds to a resource's attribute's value change
- `close` - terminates the environment associated with a resource before the agent stops managing it
- `shutdown` - called when the agent shuts down
- `imf_init`- initializes the agent to interface with the IMF notification module
- `imf_register` - registers or unregisters resource entities with the IMF notification module
- `imf_getnotification` - gets notifications about the resource state changes from IMF notification module

  Note: IMF entry points are supported only on V51 or later versions of agent.

- `migrate` - migrates a resource.

  Note: The migration is supported only on V60 or later agent versions.

- `meter` - measures the system resource utilization of a VCS resource.

  Note: The migration is supported only on V60 or later agent versions.

See "Agent entry points described" on page 27.

How the agent framework interacts with entry points

The agent framework ensures that only one entry point is running for a given resource at one time. If multiple requests are received or multiple events are scheduled for the same resource, the agent queues them and processes them one at a time. An exception to this behavior is an optimization such that the agent framework discards internally generated periodic monitoring requests for a resource that is already being monitored or that has a monitor request as the last request in resource command queue.

The agent framework is multithreaded. This means a single agent process can run entry points for multiple resources simultaneously. However, if an agent receives a request to take a given resource offline and simultaneously receives a request

to close it, it calls the `offline` entry point first. The `close` entry point is called only after the `offline` request returns or times out. If the `offline` request is received for one resource, and the `close` request is received for another, the agent can call both simultaneously.

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

Agent entry points described

This section describes each entry point in detail.

About the open entry point

The status of the `open` entry point is passed as an argument to the next monitor entry point. The name of the argument is OpenStatus. The possible value for OpenStatus is 0 and 2. A value of 0 means that the open entry point completed successfully. A value of 2 means that the open entry point has timed out.

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

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

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

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`, `offline` and `migrate` entry points to determine if bringing the resource online, offline or migration operations were 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 60 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 183.).

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

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

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

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

See “Syntax for script entry points” on page 92.

About the online entry point

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

The `online` entry point receives a resource name and `ArgList` attribute values as input. It returns an integer indicating the number of seconds to wait for the online to take effect. The typical return value is 0. If the return value is not zero, the agent framework waits the number of seconds indicated by the return value before calling the `monitor` entry point for the resource.

About the offline entry point

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

The `offline` entry point receives a resource name and `ArgList` attribute values as input. It returns an integer indicating the number of seconds to wait for the offline to take effect. The typical return value is 0. If the return value is not zero, the agent framework waits the number of seconds indicated by the return value to call the `monitor` entry point for the resource.

About the clean entry point

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

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

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

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

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

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

- VCSAgCleanOfflineHung
  The `offline` entry point did not complete within the expected time.
  See "OfflineTimeout" on page 196.
- VCSAgCleanOfflineIneffective

The `offline` entry point was ineffective. The `monitor` entry point scheduled for the resource after the `offline` entry point invocation returned a status other than OFFLINE.

- VCSAgCleanOnlineHung
  The `online` entry point did not complete within the expected time.
  (See "OnlineTimeout" on page 197.)
- 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.
  (See "ToleranceLimit" on page 201.)
- VCSAgCleanMonitorHung
  The online resource faulted because the `monitor` entry point consistently failed to complete within the expected time.
  (See "FaultOnMonitorTimeouts" on page 186.)

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

- Automatically restarts a resource on the local system when the resource faults.
  See "RestartLimit" on page 199.
- Automatically retries the `online` entry point when the attempt to bring a resource online fails.
  See "OnlineRetryLimit" on page 197.
- Enables the engine to bring a resource online on another system when the resource faults 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 pre-specified action on a resource. Use the entry point to run non-periodic actions like suspending a database or resuming the suspended database.

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

See "SupportedActions" on page 200.

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

If all actions are script based, make sure the action scripts reside within an `actions` directory under the agent directory. Create a script for each action. Use the correct action_token as the script name.

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

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

See "Syntax for C++ action" on page 61.

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

```
hares -action <res> <token> [-actionargs <arg1> ...]
                  -sys <system> [-clus <cluster> | -localclus]
```

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

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

Also,

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

Return values for action entry point

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

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

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

About the info entry point

The `info` entry point enables agents to obtain information about an online resource. For example, the Mount agent's `info` entry point could be used to report on space available in the file system. All information the `info` entry point collects is stored in the "temp" attribute `ResourceInfo`.

See “About the ResourceInfo attribute” on page 33.

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

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

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

or

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

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

See “About the ResourceInfo attribute” on page 33.

See “Syntax for C++ entry points” on page 51.

Return values for info entry point

- If the `info` entry point exits with 0 (success), the output captured on stdout for the script entry point, or the contents of the `info_output` argument for C++ entry point, is dumped to the `Msg` key of the `ResourceInfo` attribute. The `Msg` key is updated only when the `info` entry point is successful. The `State` key is set to the value: `Valid`.
- If the entry point exits with a non-zero value, `ResourceInfo` is updated to indicate the error; the script's stdout or the C++ entry point's `info_output` is ignored. The `State` key is set to the value: `Invalid`. The error message is written to the agent's log file.
- If the `info` entry point times out, output from the entry point is ignored. The `State` key is set to the value: `Invalid`. The error message is written to the agent's log file.
- If a user kills the `info` entry point (for example, `kill -15 pid`), the `State` key is set to the value: `Invalid`. The error message is written to the agent's log file. See “About logging agent messages” on page 117.
- If the resource for which the entry point is invoked goes offline or faults, the `State` key is set to the value: `Stale`.
- If the `info` entry point is not implemented, the `State` key is set to the value: `Stale`. The error message is written to the agent's log file.

About the ResourceInfo attribute

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

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

These keys are updated only by the agent framework, not the entry point. The entry point can define and add other keys (name-value pairs) and update them.

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

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

The resource for which the `info` entry point is invoked must be online.

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

For more information on `hares -flushinfo` command, refer the `hares` manual page.

Invoking the info entry point

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

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

See “InfoInterval” on page 188.

About the attr_changed entry point

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

Registering can be accomplished either through VCSAgRegister api or by definition in the RegList. Script-based agents can register only through the RegList attribute definition.

See "VCSAgRegister" on page 71.

See "VCSAgUnregister" on page 72.

See "RegList" on page 198.

The `attr_changed` entry point receives as input the resource name registered with the agent framework for notification, the name of the changed resource, the name of the changed attribute, and the new attribute value. It does not return a value.

About the close entry point

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

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

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

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 clean the setup created by the open entry point.

About the shutdown entry point

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

About the imf_init entry point

This is a type-specific entry point. The agent framework invokes this entry point when the agent starts. Agent developers can use this entry point to initialize the agent to interface with the IMF notification module.

About the imf_register entry point

The agent framework invokes this entry point to either register or unregister a resource with IMF.

The agent framework schedules a command to register a resource with IMF after resource is in either steady ONLINE or steady OFFLINE state. In steady ONLINE state, the previous and current state of the resource is ONLINE as reported by the monitor entry point. In steady OFFLINE, the previous and current state of the resource is OFFLINE, as reported by the monitor entry point.

The agent framework schedules the command to unregister a resource from IMF under following circumstances:

- When MonitorFreq key of IMF attribute has non-zero values and traditional monitor entry point detects any of the following state changes of a resource:
  - ONLINE to OFFLINE
  - OFFLINE to ONLINE
  - ONLINE to UNKNOWN
  - OFFLINE to UNKNOWN
- When you modify Mode key of the IMF attribute.
- When the ContainerInfo attribute of a resource is changed.
- If IMFRegList attribute or any attribute defined in IMFRegList is changed.

- If IMFRegList is undefined and if ArgList attribute or any attribute defined in ArgList gets changed.

About the imf_getnotification entry point

The Agent framework invokes this entry point after agent is started and the imf_init entry point returns success. Agent framework expects this as a blocking entry point and remains blocked until an event is received. After processing the event, agent framework again blocks on this entry point. When agent framework receives a notification for some resource then it confirms the resource state changes of the scheduled monitor entry point.

This is a type-specific entry point like shutdown and imf_init entry points.

About the migrate entry point

The `migrate` entry point migrates a resource. For example, the migrate entry point for an LDom agent migrates the LDom resource from a source system to a target system. When the migration is complete, the monitor entry point is automatically called by the framework to verify whether the resource has migrated. The migrate entry point receives a resource name, ArgList attribute and VCSInfo values as input. VCSInfo is an internally-generated information that contains the TargetHost name.

About the meter entry point

The `meter` entry point measures the system resource utilization of a VCS resource based on the meters configured in the Meters attribute. For example, for a resource of a LDom type, the `meter` entry point measures the virtual CPU and memory requirement of that LDom configured under the resource.

The meter entry point can be called periodically as per the values configured in the MeterInterval key of the MeterControl attribute

See “MeterControl” on page 192.

See “Meters” on page 192.

Return values for entry points

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

Table 2-2 Return values for entry points

Entry Point	Return Values
Monitor	C++ Based Returns ResStateValues: ■ VCSAgResOnline ■ VCSAgResOffline ■ VCSAgResUnknown ■ VCSAgResIntentionalOffline Script-Based Exit values: ■ 99 - Unknown ■ 100 - Offline ■ 101-110 - Online ■ 200 - Intentional Offline ■ Other values - Unknown.
Info	0 if successful; non-zero value if not successful
Online	Integer specifying number of seconds to wait before monitor can check the state of the resource; typically 0, that is, check resource state immediately.
Offline	Integer specifying number of seconds to wait before monitor can check the state of the resource; typically 0, that is, check resource state immediately.
Clean	0 if successful; non-zero value if not successful If clean fails, the resource remains in a transition state awaiting the next periodic monitor. After the periodic monitor, clean is attempted again. The sequence of clean attempt followed by monitoring continues until clean succeeds or CleanRetryLimit is not reached if it is set to non-zero value. For detailed descriptions of internal transition states, See “State transitions” on page 220.
Action	0 if successful; non-zero value if not successful
Attr_changed	None
Open	None
Close	None
Shutdown	None
imf_init	0 if successful; 1 if unsuccessful

Table 2-2 Return values for entry points *(continued)*

Entry Point	Return Values
imf_register	0 if successful; 1 if unsuccessful
imf_getnotification	0 if successful; 1 if failure; 3 if interrupted (failure case); 4 if critical failure
migrate	■ An integer in the range of 0 to 100. The typical return value is 0. If the return value is not zero, the agent framework waits for the number of seconds indicated by the (return value * 10) to call the monitor entry point for the resource. For example, for a return value of 1, agent framework schedules monitor after 1*10=10 seconds. Similarly, for a return value of 5 monitor is scheduled after 50 seconds. ■ 255 indicating that migration verification has failed and there is no need to schedule a monitor to verify whether resource has migrated. The subsequent monitor is a scheduled based on the MonitorInterval value. All other values in the range of 101 to 254 are reserved for future use. Agent framework ignores any value returned between this range and returns to previous state to continue with rest of the operations. Refer to MigrateWaitLimit and MigrateTimeout, before implementing this entry point. See "MigrateTimeout" on page 195. See "MigrateWaitLimit" on page 195. See "SupportedOperations" on page 201.
meter	■ 0 - Indicates that meter entry point has completed successfully. ■ 255 - Indicates that meter entry point has failed. METER FAILED flags will be set if meter entry point fail or timeout. ■ 254 - If the meter entry point fails with this return value, then it is treated as critical fault and metering is disabled for the resource until the agent restarts All other values in the range of 1 to 253 are reserved for future use. The Agent framework considers those values as failure with unsupported value and sets the METER FAILED flag. These failure with unsupported value will not be counted against the MeterRetryLimit, so meter entry point should not use these values. See "MeterRetryLimit" on page 193.

Considerations for using C++ or script entry points

You may implement an entry point as a C++ function or a script.

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

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

See “About creating entry points in C++” on page 49.

See “About creating entry points in scripts” on page 90.

About the VCSAgStartup routine

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

If you implement entry points using scripts

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

On UNIX, the user can use one of the different agent binaries which are provided with VCS.

See “Script based agent binaries” on page 138.

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

See “About creating entry points in scripts” on page 90.

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 VCSAgValidateAndSetEntryPoint API and specify the entry point and the function name. In the following example, the function `my_shutdown` is defined as the Shutdown entry point.

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

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(v51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPShutdown, my_shutdown);
}
```

Note that the `monitor` entry point, which is mandatory, is not specified. This indicates that it is implemented using scripts. For an entry point whose field is not set, the agent automatically looks for the correct script to execute as per following path:

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

The path where agent searches the entry piont can be different, given that the AgentDirectory attribute is set. You can refer to the *Cluster Server Administrator's Guide* for information on AgentDirectory attribute.

See “AgentDirectory” on page 181.

About the agent information file

The graphical user interface (GUI), Cluster Manager, can display information about the attributes of a given resource type. For each custom agent, developers can create an XML file that contains the attribute information for use by the GUI. The

XML file also contains information to be used by the GUI to allow or disallow certain operations on resources managed by the agent.

Example agent information file (UNIX)

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

```
<?xml version="1.0" encoding="us-ascii"?>
<agent name="FileOnOff" version="version">
    <agent_description>Creates, removes,
     and monitors files.</agent_description>
    <!--Platform the agent runs on-->
    <platform>Cross-Platform</platform>
    <!--Type of agent : script-Binary-Mixed-->
    <agenttype>Binary</agenttype>
    <!--The minimum VCS version needed for this agent-->
    <minvcsversion>5.0</minvcsversion>
    <!--The agent vendor name-->
    <vendor>VendorName</vendor>
    <!--Is Info Entry Point Implemented-->
    <info_implemented>No</info_implemented>
    <!--Attributes list for this agent-->
    <attributes>
        <PathName type="str" dimension="Scalar" editable="True"
important="True" mustconfigure="True" unique="True"
persistent="True" range="" default="" displayname="File Name">
            <attr_description>Specifies the complete pathname,
starting with the slash (/) preceding the file name.
</attr_description>
        </PathName>
    </attributes>
    <!--List of files installed by this agent-->
    <agentfiles>
        <file name="$VCS_HOME/bin/FileOnOff/FileOnOffAgent" />
    </agentfiles>
</agent>
```

Agent information

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

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

Agent Information	Example
Agent name	`name="FileOnOff"`
Version	`version="x.y"`
Agent description	`<agent_description>Creates, removes, and monitors files.</agent_description>`
AIX, HP-UX, Linux, Solaris, or Cross-Platform. Platform, for example: ■ AIX ■ Solaris ■ Cross-platform	`<platform>Cross-Platform</platform>`
Agent vendor	`<vendor>VendorName</vendor>`
`info` entry point implemented or not; Yes, or No; if not indicated, `info` entry point is assumed not implemented	`<info_implemented>No</info_implemented>`
Agent type, for example, Binary, Script or Mixed	`<agenttype>Binary</agenttype>`
Compatibility with Cluster Server; the minimum version required to support the agent	`<minvcsversion>5.0</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 18.
dimension	Values for the attribute dimension, such as "Scalar;" See “About attributes” on page 17.
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.
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.

Table 2-4 Description of attribute argument details in XML file *(continued)*

Argument	Description
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 `$VCS_HOME/bin/resource_type` directory or in the directory mentioned in the AgentDirectory attribute.
2. Make sure that the command server is running on each node in the cluster.
3. Restart the GUI to have the agent's information shown in the GUI.

About the ArgList and ArgListValues attributes

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

In C++ agents, the value of the ArgListValues attribute is passed through a parameter of type `void **`. For example, the signature of the online entry point is:

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

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

The number of values in the ArgListValues should not exceed more than 425. This requirement becomes a consideration if an attribute in the ArgList is a keylist, a vector, or an association. Such type of non-scalar attributes can typically take any number of values, and when they appear in the ArgList, the agent has to compute ArgListValues from the value of such attributes. If the non-scalar attribute contains many values, it will increase the size of ArgListValues. Hence when developing an agent, this consideration should be kept in mind when adding a non-scalar attribute in the ArgList. Users of the agent need to be notified that the attribute should not be configured to be so large that it pushes that number of values in the ArgListValues attribute to be more than 425.

ArgListValues attribute for agents registered as V50 and later

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

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

Overview of the name-value tuple format

For agents registered with agent version V40 and earlier, it's 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 in the resource type definition.

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

The general tuple format for attributes in the ArgList is:

<name> <number_of_elements_in_value> <value>

Scalar attribute format

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

<attribute_name> 1 <value>

Example is:

DiskGroupName 1 mydg

Vector attribute format

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

<attribute_name> <number_of_values_in_vector> <values_in_vector>

Examples are:

MyVector 3 aa cc dd

MyEmptyVector 0

Keylist attribute format

For string keylist attributes, the formatting is:

<attribute_name> <number_of_keys_in_keylist> <keys>

Examples are:

DiskAttr 4 hdisk3 hdisk4 hdisk5 hdisk6

DiskAttr 0

Association attribute format

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

<attribute_name> <number_of_keys_and_values> <key_value_pair>

Examples are:

MyAssoc 4 key1 val1 key2 val2

MyAssoc 0

ArgListValues attribute for different agents versions

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

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

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

```
            VectorAttr, AssocAttr }
)
```

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

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

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

```
attr_val[0] = "IntAttr"
attr_val[1] = "1"      // Number of components in
                       // IntAttr attr value
attr_val[2] = "100"    // Value of IntAttr
attr_val[3] = "StringAttr"
attr_val[4] = "1"      // Number of components in
                       // StringAttr attr value
attr_val[5] = "Oracle" // Value of StringAttr
attr_val[6] = "VectorAttr"
attr_val[7] = "3"      // Number of components in
                       // VectorAttr attr value
attr_val[8] = "vol1"
attr_val[9] = "vol2"
attr_val[10] = "vol3"
attr_val[11] = "AssocAttr"
attr_val[12] = "4"     // Number of components in
                       // AssocAttr attr value
attr_val[13] = "disk1"
attr_val[14] = "1024"
attr_val[15] = "disk2"
attr_val[16] = "512"
attr_val[17] = NULL    // Last element
```

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

```
attr_val[0] ===>  "100"   // Value of IntAttr, the first
                          // ArgList attribute.
attr_val[1] ===> "Oracle" // Value of StringAttr.
attr_val[2] ===> "3"      // Number of components in
                          // VectorAttr.
```

```
attr_val[3] ===> "vol1"
attr_val[4] ===> "vol2"
attr_val[5] ===> "vol3"
attr_val[6] ===> "4"      // Number of components in
                          // AssocAttr.
attr_val[7] ===> "disk1"
attr_val[8] ===> "1024"
attr_val[9] ===> "disk2"
attr_val[10]===> "512"
attr_val[11]===> NULL     // Last element.
```

About the entry point timeouts

Use the AEPTimeout attribute to append the timeout value for a particular entry point.

This feature does not apply to pre-V50 agents.

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

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

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

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

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

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

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

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

See “AEPTimeout” on page 180.

Chapter 3

Creating entry points in C++

This chapter includes the following topics:

About creating entry points in C++

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

The following guidelines also apply:

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

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

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

UNIX: $VCS_HOME/src/agent/Sample

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

See "Agent Framework primitives for container support" on page 84.

Entry point examples in this chapter

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

In the types.cf format:

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

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

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 pre-specified action
info	Populates the ResourceInfo attribute with the values of the attributes specified by the PathName attribute

Data Structures

This section describes the various enumerations in relation to the entry points.

- VCSAgResState:
  The VCSAgResState enumeration describes what state the monitor entry point can return.

```
enum VCSAgResState {
VCSAgResOffline, // Resource is OFFLINE
VCSAgResOnline, // Resource is ONLINE
VCSAgResUnknown,// Resource state is UNKNOWN
VCSAgResIntentionalOffline// Resource state is OFFLINE, but is
intentionally done. Only in V51 and later agents)
};
```

- VCSAgWhyClean
  This VCSAgWhyClean enumeration describes 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.
};
```

- VCSAgResInfoOp
  The VCSAgResInfoOp enumeration indicates whether to initialize or update the data in the ResourceInfo attribute.

```
enum VCSAgResInfoOp {
        VCSAgResInfoAdd = 1,// Add non-default keys to the
ResourceInfo attribute.
        VCSAgResInfoUpdate// Update only the non-default
key-value data pairs in the ResourceInfo attribute.
};
```

Syntax for C++ entry points

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

Syntax for C++ VCSAgStartup

```
void VCSAgStartup();
```

Note that the name of the C++ function must be `VCSAgStartup()`.

For example:

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

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

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


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

unsigned int res_online(const char *res_name,
                                   void **attr_val) {
...
}
unsigned int res_offline(const char *res_name,
                                    void **attr_val) {
...
}
unsigned int res_clean(const char *res_name,
```

```
VCSAgWhyClean reason, void **attr_val) {
...
}
```

Syntax for C++ monitor

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

You may select any name for the function.

The parameter `conf_level` is an output parameter. The return value, which indicates the resource status, must be a defined `VCSAgResState` value.

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

For example:

```
#include "VCSAgApi.h"

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

    // Code to determine the state of a resource.
    VCSAgResState res_state = ...
    if (res_state == VCSAgResOnline) {
    // Determine the confidence level (0 to 100).
        *conf_level = ...
    }
    else {
        *conf_level = 0;
    }
    return res_state;
}

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);
```

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

Syntax for C++ info

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

You may select any name for the function.

resinfo_op

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

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

info_output

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

The `info` entry point is responsible for allocating memory for `info_output`. The agent framework handles the deletion of any memory allocated to this argument.

Since memory is allocated in the entry point and deleted in the agent framework, the entry point needs to pass the address of the allocated memory to the agent framework.

opt_update_args

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

For example:

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

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

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

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

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

opt_add_args

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

For example:

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

A valid `opt_add_args` array for this would be:

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

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

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

Example: info entry point implementation in C++

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

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

Example V51 entry point:

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

    *info_output = out;

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

    // Use the stat system call on the file to get its
    // information The attr_val array will look like "PathName"
    // "1" "<pathname value>" ... Assuming that PathName is the
    // first attribute in the attr_val array, the value
    // of this attribute will be in index 2 of this attr_val
    // array

    if (attr_val[2]) {
```

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

            if (resinfo_op == VCSAgResInfoAdd) {
                // Add and initialize all the static and
                // dynamic keys in the ResourceInfo attribute
                args = new char * [7];
                for (I = 0; I < 6; I++) {
                    args[i] = new char [15];
                }

                // All the static information - file owner
                // and group
                VCSAgSnprintf(args[0], 15, "%s", "Owner");
                VCSAgSnprintf(args[1], 15, "%d",
                stat_buf.st_uid);
                VCSAgSnprintf(args[2], 15, "%s", "Group");
                VCSAgSnprintf(args[3], 15, "%d",
stat_buf.st_gid);

               // Initialize the dynamic information for the file
                VCSAgSnprintf(args[4], 15, "%s", "FileSize");
                VCSAgSnprintf(args[5], 15, "%d",
                stat_buf.st_size);
                args[6] = NULL;
                *opt_add_args = args;
            }
            else {

                // Simply update the dynamic keys in the
                // ResourceInfo attribute. In this case, the
                // dynamic info on the file
                args = new char * [3];
                for (I = 0; I < 2; I++) {
                    args[i] = new char [15];
                }
                VCSAgSnprintf(args[0], 15, "%s", "FileSize");
                VCSAgSnprintf(args[1], 15, "%d",
                stat_buf.st_size);
                args[2] = NULL;
                *opt_update_args = args;
            }
```

```
        }
        else {
            // Set the output to indicate the error
            VCSAgSnprintf(out, 80, "Stat on the file %s failed",
            attr_val[2]);
            return 1;
        }
    }
    else {
        // Set the output to indicate the error
       VCSAgSnprintf(out, 80, "Error in arglist values passed to
        the info entry point");
        return 1;
    }

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

} // End of entry point definition
```

Syntax for C++ online

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

You may select any name for the function.

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

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

For example:

```
#include "VCSAgApi.h"

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

```
    return 0;
}

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPOnline, res_online);
}
```

Syntax for C++ offline

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

You may select any name for the function.

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

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

For example:

```
#include "VCSAgApi.h"

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

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

```
    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

   VCSAgValidateAndSetEntryPoint(VCSAgEPOffline, res_offline);}
```

Syntax for C++ clean

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

You may select any name for the function.

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

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

For example:

```
    #include "VCSAgApi.h"

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


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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPClean, res_clean);}
```

Syntax for C++ action

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

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

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

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

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

  If the user ran

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

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

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

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

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

For example:

```
extern "C"
unsigned int res_action (const char *res_name, const char
    *token,void **attr_val, char **args, char
    *action_output)
    {
const int output_buffer_size = 2048;
    //
    // checks on the attr_val entry point arg list
    // perform an action based on the action token passed in

if (!strcmp(token, "token1")) {
    //
    // Perform action corresponding to token1
    //
} else if (!strcmp(token, "token2") {
    //
    // Perform action corresponding to token2
    //
}
:
:
:
} else {
    //
    // a token for which no action is implemented yet
    //
VCSAgSnprintf(action_output, output_buffer_size, "No implementation
provided for token(%s)", token);
}
    //
    // Any other checks to be done
    //
    //
    // return value should indicate whether the ep succeeded or
    // not:
    // return 0 on success
    // any other value on failure
    //
if (success) {
return 0;
}
else {
return 1;
```

```
}
}
void VCSAgStartup()
{
VCSAG_LOG_INIT("VCSAgStartup");
VCSAgSetLogCategory(10051);
VCSAgInitEntryPointStruct(V51);
VCSAgValidateAndSetEntryPoint(VCSAgEPAction, res_action);
}
```

Syntax for C++ attr_changed

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

The parameter `new_val` contains the attribute's new value. The encoding of `new_val` is similar to the encoding described under *About the ArgList and ArgListValues attributes*.

See "About the ArgList and ArgListValues attributes" on page 44.

You may select any name for the function.

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

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

Note: This entry point is called only if you register for change notification using the primitive VCSAgRegister or the agent parameter RegList.

See "RegList" on page 198.

For example:

```
#include "VCSAgApi.h"

void
 res_attr_changed(const char *res_name,
                        const char *changed_res_name,
                        const char *changed_attr_name,
```

```
                        void **new_val) {
    // When the value of attribute Foo changes, take some action.
    if ((strcmp(res_name, changed_res_name) == 0) &&
            (strcmp(changed_attr_name, "Foo") == 0)) {
        // Extract the new value of Foo. Here, it is assumed
        // to be a string.
        const char *foo_val = (char *)new_val[0];
        // Implement the action.
            ...
    }
    // Resource Ora1 managed by this agent needs to
    // take some action when the Size attribute of
    // the resource Disk1 is changed.
    if ((strcmp(res_name, "Ora1") == 0) &&
            (strcmp(changed_attr_name, "Size") == 0) &&
            (strcmp(changed_res_name, "Disk1") == 0)) {

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

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPAttrChanged,
res_attr_changed);}
```

Syntax for C++ open

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

You may select any name for the function.

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

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

For example:

```
#include "VCSAgApi.h"

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

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPOpen, res_open);
}
```

Syntax for C++ close

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

You may select any name for the function.

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

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

For example:

```
#include "VCSAgApi.h"

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

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

```
    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPClose, res_close);
}
```

Syntax for C++ shutdown

```
void shutdown();
```

You may select any name for the function.

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

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

For example:

```
#include "VCSAgApi.h"

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

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPShutdown, shutdown);
}
```

Syntax for C++ migrate

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

You can assign any name to the function.

Set the `VCSAgValidateAndSetEntryPoint()` parameter as the name of function of the entry point.

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

In the following example, the function res_migrate is defined as the migrate entry point.

For example:

```
#include "VCSAgApi.h"
unsigned int
res_migrate(const char *res_name, void **attr_val) {
// Implement the code to migrate a resource here.

}
void VCSAgStartup()
{
VCSAG_LOG_INIT("VCSAgStartup");
VCSAgSetLogCategory(10051);
VCSAgInitEntryPointStruct(V60);
VCSAgValidateAndSetEntryPoint(VCSAgEPMigrate, res_migrate);
}
```

Syntax for C++ meter

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

You can assign any name to the function.

Set the `VCSAgValidateAndSetEntryPoint()` parameter as the name of function of the entry point.

Refer to See "Return values for entry points " on page 36.

In the following example, the function `res_meter` is defined as the meter entry point.

For example:

```
#include "VCSAgApi.h"
unsigned int
res_meter(const char *res_name, void **attr_val) {
// Implement the code to meter a resource here.
}
void VCSAgStartup()
{
VCSAG_LOG_INIT("VCSAgStartup");
VCSAgSetLogCategory(10051);
VCSAgInitEntryPointStruct(V60);
VCSAgValidateAndSetEntryPoint(VCSAgEPMeter, res_meter);
}
```

Along with attribute that is specified in ArgList, the meter entry point also gets the value of Meters and MeterUnit attributes.

You can refer to the *Cluster Server Administrator's Guide* for information on MeterUnit attribute.

See "Meters" on page 192.

Agent framework primitives

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

See "Agent Framework primitives for container support" on page 84.

VCSAgGetMonitorLevel

```
int VCSAgGetMonitorLevel(int *level_one, int *level_two);
```

The agent developer can use this primitive to query if the LevelOne (Basic) monitoring or the LevelTwo (Detail) monitoring or both need to be scheduled.

- Output parameters:
  - level_one - This parameter will be updated to 1 or 0. The value of 0 indicates that the basic monitoring should not be scheduled. And the value of 1 indicates that the basic monitoring should be scheduled.
    See "IMF" on page 187.
  - level_two - This parameter will be updated to 0, 1, or 2 . The value of 0 indicates that the detail monitoring should not be scheduled, and the value of 1 indicates that the detail monitoring should be scheduled. And the value of 2 indicates that the detail monitoring should be scheduled if basic monitoring (level_ one) reports the state as online in current running monitor.
- Return values: It can be set to VCSAgSuccess or VCSAgFailure based on whether the api passes or fails.
  The following example outlines the process of setting the output parameters:
  For example, if you set LevelTwoMonitorFrequency to 5 and the resource state is ONLINE, then every fifth monitor cycle, level_two will have the value as 1. If the resource state is OFFLINE, then every monitor cycle level_two will have the value as 2.
  See "LevelTwoMonitorFreq" on page 189.
  If you set MonitorFreq to 5 and the resource is registered with IMF, then every fifth monitor cycle level_one parameter will have the value of 1.
  See "IMF" on page 187.

- Usage:

```
int ret = VCSAgFailure;
ret = VCSAgGetMonitorLevel(&level_one, &level_two);
```

Note: This API can only be used in monitor entry point. It does not reflect correct monitor levels when you call this API in other entry points.

See "VCSAG_GET_MONITOR_LEVEL" on page 97.

VCSAgGetFwVersion

```
int VCSAgGetFwVersion();
```

This primitive will return the latest agent framework version.

See "VCSAG_GET_AGFW_VERSION" on page 98.

VCSAgGetRegVersion

```
int VCSAgGetRegVersion();
```

This primitive will return the currently registered agent framework version.

See "VCSAG_GET_REG_VERSION" on page 98.

VCSAgRegisterEPStruct

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

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

VCSAgSetCookie2

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

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

This value, which is transparent to the agent framework, can be obtained by calling the primitive `VCSAgGetCookie()`. A cookie is not stored permanently. It is lost when the agent process exits. This primitive can be called from any entry point. For example:

```
#include "VCSAgApi.h"
...
// Assume that the online, offline, and monitor
// operations on resource require a certain key. Also
// assume that obtaining this key is time consuming, but
// that it can be reused until this process is
// terminated.
//
// In this example, the open entry point obtains the key
// and stores it as a cookie. Subsequent online,
// offline, and monitor entry points get the cookie and
// use the key.
//
// Note that the cookie name can be any unique string.
// This example uses the resource name as the cookie
// name.
//
void *get_key() {
               ...
}
void res_open(const char *res_name, void **attr_val) {
              if (VCSAgGetCookie(res_name) == NULL) {
                         void *key = get_key();
                         VCSAgSetCookie2(res_name, key);
              }
}
VCSAgResState res_monitor(const char *res_name, void
              **attr_val, int *conf_level_ptr) {
              VCSAgResState state = VCSAgResUnknown;
              *conf_level_ptr = 0;
              void *key = VCSAgGetCookie(res_name);
              if (key == NULL) {
                  // Take care of the rare cases when
                  // the open entry point failed to
                  //  obtain the key and set the the cookie.
                         key = get_key();
                         VCSAgSetCookie2(res_name, key);
              }
```

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

VCSAgRegister

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

This primitive requests that the agent framework notify the resource `notify_res_name` when the value of the attribute `attr_name` of the resource `res_name` is modified. The notification is made by calling the `attr_changed` entry point for `notify_res_name`.

Note that `notify_res_name` can be the same as `res_name`.

This primitive can be called from any entry point, but it is useful only when the `attr_changed` entry point is implemented. For example:

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

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

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

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

      // Register to get notified when the
```

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

VCSAgUnregister

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

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

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

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

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

VCSAgGetCookie

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

This primitive requests that the agent framework get the cookie set by an earlier call to `VCSAgSetCookie2()`. It returns `NULL` if cookie was not previously set. This primitive can be called from any entry point. For example:

```
#include "VCSAgApi.h"
...
```

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

void *get_key() {
               ...
}
void res_open(const char *res_name, void **attr_val) {
              if (VCSAgGetCookie(res_name) == NULL) {
                        void *key = get_key();
                        VCSAgSetCookie2(res_name, key);
              }
}
VCSAgResState res_monitor(const char *res_name, void
                      **attr_val, int *conf_level_ptr) {
              VCSAgResState state = VCSAgResUnknown;
              *conf_level_ptr = 0;
              void *key = VCSAgGetCookie(res_name);
              if (key == NULL) {
              // Take care of the rare cases when the open
              // entry point failed to obtain the key and
              // set the the cookie.
                        key = get_key();
                        VCSAgSetCookie2(res_name, key);
              }
              // Use the key for testing if the resource is
              // online, and set the state accordingly.
              ...
              return state;
}
```

VCSAgStrlcpy

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

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

VCSAgStrlcat

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

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

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

VCSAgSnprintf

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

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

VCSAgCloseFile

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

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

VCSAgDelString

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

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

VCSAgExec

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

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

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

Explanation of arguments to the function:

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

Return value: VCSAgSuccess if the execution was successful.

Example:

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

args[0] = "/usr/bin/ls";
```

```
args[1] = "/tmp";
args[2] = NULL;
int result = VCSAgExec(args[0], args, buf, 100, &status);

if (result == VCSAgSuccess) {


// 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 canceled inside VCSAgExecWithTimeout. So the entry point needs to make sure that it pushes the appropriate cancellation cleanup handlers before calling VCSAgExecWithTimeout. The forced cancellation ensures that a service thread running a timed out entry point does not keep running or waiting for the child process created by this API to exit but instead honors a cancellation request when it receives one.

Explanation of arguments to the function:

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

Return value: VCSAgSuccess if the execution is successful.

VCSAgGenSnmpTrap

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

This API is used to send a notification via SNMP and/or SMTP. The clusterOutOfBandTrap 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.

Explanation of arguments to the function:

`fname` — File name

`ltype` — Lock type

`VCSAgLockType enum` describes the type of lock.

For example:

```
enum VCSAgLockType {
 VCSAgExclusiveLock, //for write operation
 VCSAgSharedLock //for read operation
 }
```

`btype` — Blocking type

`VCSAgBlockingType enum` describes the type of blocking which the user can require.

For example:

```
enum VCSAgBlockingType{
  VCSAgBlocking,
  VCSAgNonBlocking
 }
```

`errp` — Output parameter to return the error value.

Warning: Do not do any operations on the file (ex, open, or close) within this process, except through acquiring the read operation (shared lock) or write operation (exclusive lock) or VCSAgUnlock() interface.

VCSAgInitEntryPointStruct

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

This primitive enables agents to initialize the entry point struct depending on the agent framework version passed to this API. It must be called only from the VCSAgStartup function.

Examples:

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

V40 and V50	■ open ■ monitor ■ online ■ offline ■ clean ■ action ■ info ■ attr_changed ■ close ■ shutdown
V51	■ imf_init ■ imf_register ■ imf_getnotification
V60	■ migrate ■ meter

For information on available registration version numbers, check the VCSAgApiDefs.h header file availabe in the following location:

```
/opt/VRTSvcs/include/VCSAgApiDefs.h
```

VCSAgSetStackSize

```
void VCSAgSetStackSize(int I)
```

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

VCSAgUnlockFile

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

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

Mt-safe; deferred cancel safe.

Explanation of arguments to the function:

`fname`	File name
`errp`	Output parameter to return the error value.

Warning: Do not do any operations on the file (ex, open, or close) within this process, except through acquiring the read operation (shared lock) or write operation (exclusive lock) or VCSAgUnlock() interface.

VCSAgValidateAndSetEntryPoint

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

This primitive enables an agent developer to register any C++ entry point with the agent framework. And also performs the signature check for the entry point function at compile time.

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

Usage:

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

VCSAgSetLogCategory

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

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

VCSAgGetProductName

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

This API fetches the name of the product for logging purposes

VCSAgMonitorReturn

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

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

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

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

Note: You can also call this API using the macro `VCSAG_MONITOR_RETURN` with the same arguments as passed to `VCSAgMonitorReturn` API.

VCSAgSetResEPTimeout

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

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

Usage:

```
VCSAgSetResEPTimeout(tmo);
```

See "VCSAG_SET_RES_EP_TIMEOUT" on page 98.

VCSAgDecryptKey

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

This API lets you decrypt an encrypted string passed in the ArgListValues by the user. Typically users encrypt string attribute values for passwords using the

encryption commands provided by VCS. An entry point can use this API to decrypt the encrypted string and get the original string.

VCSAgGetConfDir

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

Returns the name of the VCS configuration directory.

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

Caller must supply the buffer

VCSAgGetHomeDir

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

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

Caller must supply the buffer

VCSAgGetLogDir

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

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

Caller must supply the buffer

VCSAgGetSystemName

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

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

Caller must supply the buffer

VCSAG_CONSOLE_LOG_MSG

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

Use the `VCSAG_CONSOLE_LOG_MSG` macro to send messages to the HAD log. If the messages are of CRITICAL or ERROR severity, then the messages are also logged to the console.

Usage:

```
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
```

See "C++ agent logging APIs" on page 120.

VCSAG_LOG_MSG

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

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.

Usage:

```
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);
```

See "C++ agent logging APIs" on page 120.

VCSAG_LOGDBG_MSG

Use the macros `VCSAG_LOGDBG_MSG` within agent entry points to log debug messages of a specific severity level to the agent log. The `VCSAG_LOGDBG_MSG` macro controls logging at the level of the resource type level.

```
VCSAG_LOGDBG_MSG(dbgsev, flags, fmt, variable_args);
```

The VCSAG_LOGDBG_MSG macro controls logging at the level of the resource type level.

Usage:

```
VCSAG_LOGDBG_MSG(VCS_DBG5, VCS_DEFAULT_FLAGS,
 "Received AMF monitor stop. Unregistering the group");
```

See "C++ agent logging APIs" on page 120.

VCSAG_RES_LOG_MSG

The macro `VCSAG_RES_LOG_MSG` can be used to print debug log message at resource level for a specific resource by enabling debugging at resource level by overriding LogDbg attribute. It only accepts debug severities i.e. DBG_1 to DBG_21.

```
VCSAG_RES_LOG_MSG(dbgsev, flags, fmt, variable_args);
```

The `VCSAG_RES_LOG_MSG` macro controls logging at the level of the resource type level.

Usage:

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

See "C++ agent logging APIs" on page 120.

Agent Framework primitives for container support

The following APIs are for use in agents that run in AIX WPARs, XRM containers, Solaris zones and Linux Docker containers.

Note that:

- Zones are supported by Solaris version 10 and above.
- Docker containers are supported on RHEL7 and SLES12.
- Container support is available only with agent version V50 or later.

VCSAgIsContainerUp

```
int VCSAgIsContainerUp();
```

This API returns either True or False. If the container configured under Service Group is up and running, this API returns True, else it returns False.

VCSAgGetContainerTypeEnum

```
VCSAgContainerType VCSAgGetContainerTypeEnum(const char *ctype);
```

This primitive takes a Container type name and returns a corresponding VCSAgContainerType enum value.

VCSAgExecInContainer2

```
int VCSAgExecInContainer2(const CHAR *path, CHAR *const argv[], char
*buf, long buf_size, unsigned long *exit_codep);
```

This API is the same as VCSAgExec; however, this API should be used by an agent only to execute a particular command or script in a specific container on the system. If the container is not enabled or invalid container is specified or if OS does not support container, then the API executes the command or script in the global container. If there are no containers configured on the system, or if the agent has no need to execute a script in a specific container, use the VCSAgExec API.

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

See "VCSAgExec" on page 75.

See "VCSAG_EXEC_IN_CONTAINER " on page 110.

VCSAgIsContainerCapable

```
VCSAgBool VCSAgIsContainerCapable();
```

This API returns either True or False.

- For Solaris zones
  If the agent is running on a Solaris 11 system, the API returns True; otherwise it returns False.
  Agents can use this API to decide whether or not to perform zone-specific operations like comparing the zone_id field in the psinfo structure with the ID of the zone name specified in the resource configuration to confirm whether the found process is indeed the process the agent is looking for.
- For XRM
  If the agent is running on a system that has xrm available, the API returns True; otherwise it returns False.
- For WPARs
  If the agent is running on a system that has WPARs available, the API returns True; otherwise it returns False.

VCSAgExecInContainerWithTimeout

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

This API is similar to the VCSAgExecWithTimeout API. This API can be used by an agent only to execute a particular command or script in a specific container on the system. If the container is not enabled or invalid container is specified or if OS does not support container, then the API executes the command or script in the global container. If there are no containers configured on the system, or if the agent has no need to exec a script in a specific container, the VCSAgExecWithTimeout API should be used.

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

VCSAgGetUID

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

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

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

The API returns 0 on success and 1 on failure

VCSAgIsPidInContainer

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

This API checks if the given pid is running inside the container as specified in the resource object. If the container is not enabled then the API checks that the pid is running in the global container.

Return values

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

VCSAgIsProcInContainer

Note: This API is not supported for Linux Docker containers.

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

This API checks if the process corresponding to the given psinfo structure is running inside the container as specified in the resource object. If the container is not enabled then the API checks that the process is running in the global container.

Return values

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

See "VCSAG_IS_PROC_IN_CONTAINER" on page 110.

VCSAgGetContainerID2

For Linux:

```
int VCSAgGetContainerID2(char*containerid)
```

This API returns 0 if container ID is successfully retrieved and -1 if not.

For other platforms:

```
int VCSAgGetContainerID2()
```

This API retrieves the ID of the container.

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

Return Values

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

VCSAgGetContainerName2

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

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

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

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

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

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

VCSAgGetContainerBasePath

Note: This API is not supported for Linux Docker containers.

```
int VCSAgGetContainerBasePath (char *buf, int bufsize, int *exit_info)
```

This API returns the base path of the container mentioned under the ContainerInfo attribute at group level. This API must be called from the global zone or WPAR.

For Solaris zones:

- If the agent is running on a Solaris machine, the API returns the base path of the zone where zone is installed.

For WPARs

- If the agent is running on an AIX machine, the API returns the base path of the WPAR where WPAR is installed.

Input parameters:

`buf`	Buffer to store the base path of the container. Caller must make the provision to reserve and release the memory for the buffer.
`bufsize`	Size of the buffer passed.

Output parameters:

`buf`	Buffer to store the base path of the container at the end of its execution.
`exit_info`	Provides extended information to the caller in certain cases as described under Return values.

Return values:

0	If ContainerInfo attribute is set properly, which means: ■ Name is set to <valid_container_name> ■ Type is set to <valid_container_type> ■ Enabled is set to 1 Container's base path is returned in the buf parameter.
1	If `buf` is passed as null.
2	If the buffer size is smaller than the size of the container's base path. The exit_info parameter is updated to reflect the correct value of the buffer size needed to be passed.
3	If the Enable key of the ContainerInfo attribute is set to 0 or 2. Container's base path is returned in buf parameter only if Name key is set to <valid_container_name>.
4	If the ContainerInfo attribute is not set for the resource. For example, Name key of ContainerInfo is "" or Type key of ContainerInfo is invalid.
5	If command to obtain the base path of the container fails. The exit_info parameter is updated accordingly with the exit status of the command.
6	If OS is not container capable.

See "VCSAG_GET_CONTAINER_BASE_PATH" on page 107.

VCSAgGetContainerEnabled

This API returns the Enabled key of ContainerInfo attribute.

Return values:

0	If ContainerInfo is not defined at group level then it returns the default value of the enabled key.
0, 1, or 2	If ContainerInfo is defined at group level then it returns the current value of the enabled key.

Refer to the *Veritas InfoScale 7.4.2 Virtualization Guide* for more information on the values of the enabled key.

Chapter 4

Creating entry points in scripts

This chapter includes the following topics:

About creating entry points in scripts

On UNIX, script agents use one of the different agent binaries that are shipped with VCS. The agent binaries are located at:

$VCS_HOME`/bin/`

See "Script based agent binaries" on page 138.

You can implement entry points using C++ or scripts. If you are implementing even one entry point in c++ then you must implement the `VCSAgStartup` function. If you do not implement any entry points in C++, then you do not need to implement the `VCSAgStartup` function since the default implementation of `VCSAgStartup` is present in the script agent binary provided by VCS as mentioned above.

See "About the VCSAgStartup routine" on page 39.

You can use script-based entry points to develop agents for monitoring applications that run in containers, including non-global zones. VCS provides APIs for container support. You can use Perl, shell, or Python scripts to develop entry points.

See "Agent Framework primitives with container support" on page 107.

Rules for using script entry points

Script entry points can be executables or scripts, such as shell, Perl, or Python (the product includes Perl and Python distributions).

Note: Python scripts are supported only on the Linux and the Windows platforms.

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

On UNIX platforms

- In the `VCSAgStartup` function, if you do not set a C++ function for an entry point using the VCSAgValidateAndSetEntryPoint() API, then the agent framework assumes the entry point is script-based.
  See "About the VCSAgStartup routine" on page 39.
- Verify the name of the script file is the same as the entry point name.
- Place the file in the $VCS_HOME`/bin/`*`resource_type`* directory or in the directory mentioned in the AgentDirectory attribute. If for example, the `online` entry point for Oracle were implemented using Perl, the `online` script must be:

  ```
  $VCS_HOME/bin/Oracle/online
  ```

- If you write scripts in shell, verify the PATH environment variable includes the directory where sh is installed.

Parameters and values for script entry points

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

Some entry points have an output parameter that is returned through the program exit value. See the entry point description for more information.

See "Syntax for script entry points" on page 92.

ArgList attributes

Specifies the attributes that must be passed to the agent entry points.

See "About the ArgList and ArgListValues attributes" on page 44.

Examples

If Type "Foo" is defined in types.cf as:

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

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

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

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

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

See "About the ArgList and ArgListValues attributes" on page 44.

Syntax for script entry points

The following paragraphs describe the syntax for script entry points.

Syntax for the monitor script

```
monitor resource_name ArgList_attribute_values
```

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

- 99 indicates unknown.
- 100 indicates offline.
- 101 indicates online and a confidence level of 10.

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

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

Syntax for the online script

```
online resource_name  ArgList_attribute_values
```

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

Syntax for the offline script

```
offline resource_name  ArgList_attribute_values
```

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

It also means the time (in seconds) that must pass before executing the monitor entry point to validate proper operation.

Syntax for the clean script

```
clean resource_name clean_reason argList_attribute_values
```

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

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

(See "OfflineTimeout" on page 196.)

1 - The `offline` entry point was ineffective.

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

(See "OnlineTimeout" on page 197.)

3 - The `online` entry point was ineffective.

4 - The resource was taken offline unexpectedly.

5 - The `monitor` entry point consistently failed to complete within the expected time.

(See "FaultOnMonitorTimeouts" on page 186.)

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

Syntax for the action script

```
action resource_name
ArgList_attribute_values_AND_action_arguments
```

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

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

Syntax for the attr_changed script

```
attr_changed resource_name changed_resource_name
        changed_attribute_name new_attribute_value
```

The exit value is ignored.

Note: This entry point is called only if you register for change notification using the primitive `VCSAgRegister`() (See "VCSAgRegister" on page 71.), or the agent parameter `RegList` (See "RegList" on page 198.).

Syntax for the info script

```
info resource_name resinfo_op ArgList_attribute_values
```

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

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

This entry point can add and update static and dynamic name-value pairs to the `ResourceInfo` attribute. The `info` entry point has no specific output, but rather, it updates the `ResourceInfo` attribute.

Syntax for the open script

```
open resource_name  ArgList_attribute_values
```

The exit value is ignored.

Syntax for the close script

```
close resource_name  ArgList_attribute_values
```

The exit value is ignored.

Syntax for the shutdown script

```
shutdown
```

The exit value is ignored.

Syntax for the imf_init script

```
imf_init   type_name
```

where `type_name` is the type of agent. For example, Mount, Process, Application and so on.

The exit value is 0 (zero) if successful and 1 (one) if unsuccessful.

Syntax for the imf_register script

```
imf_register  res_name mswitch rstate ArgList_attribute_values
```

`res_name`	Name of the resource that is required to be registered.
`mswitch`	Possible value of this parameter is either VCSAgIMFMonitorStop or VCSAgIMFMonitorStart. If its value is VCSAgIMFMonitorStart, then it registers a resource with underlying module. If its value is VCSAgIMFMonitorStop, then it unregisters a resource from underlying module. This is passed by the agent framework.
`rstate`	Possible value for this parameter is either VCSAgIMFResOffline or VCSAgIMFResOnline. If its value is VCSAgIMFResOffline, then it registers a resource with underlying module for OFFLINE monitoring. If its value is VCSAgIMFResOnline, then it registers underlying module for ONLINE monitoring.

The exit value is 0 (zero) if successful or non-zero if unsuccessful.

Note: The imf_register entry point also returns the resource ID to agent framework by writing the resource ID to the exposed PIPE FD.

Syntax for the imf_getnotification script

```
imf_getnotification type_name
```

`type_name`	Type of the agent. For example, Mount, Process, Application and so on.

The exit value is 0 (zero) if successful; 1 if failure; 3 if interrupted (failure case); 4 if critical failure.

Note: The imf_getnotification entry point also returns the resource event notification to the agent framework by writing the event information to the exposed PIPE FD.

Syntax for migrate script

```
migrate resource_name attribute_values
```

The exit value is interpreted as the expected time (in seconds) for the migrate procedure to be effective. The exit values is an integer in the range of 0 to 100. The agent framework waits for the number of seconds as indicated by the value (return value * 10) to call the monitor entry point for the resource to validate proper operation. The exit value is typically 0. For more information refer to return code of migrate entry point.

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

Syntax for meter script

```
meter resource_name attribute_values
```

Along with the attribute that is specified in ArgList, meter entry point also gets the value of Meters and MeterUnit attribute.

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

Agent framework primitives

The agent framework implements Perl-, Shell-, or Python-based methods, which are called primitives. The following sections describe the primitives.

VCSAG_GET_MONITOR_LEVEL

The agent developer can use this primitive to query if the LevelOne (Basic) monitoring or the LevelTwo (Detail) monitoring or both need to be scheduled.

Output parameters:

- level_one: This parameter will be updated to 1 or 0 if basic monitoring needs to be scheduled or not. A value of 0 means that basic monitoring should not be scheduled while a value of 1 means that basic monitoring should be scheduled.
  See "IMF" on page 187.
- level_two: This parameter will be updated to 0, 1, or 2, based on the present state of the resource, and if detail monitoring needs to be scheduled. A value of 0 means that detail monitoring should not be scheduled, a value of 1 means that detail monitoring should be scheduled, and a value of 2 means that detail monitoring should be scheduled if basic monitoring (level_ one) reports state as online

  Following example describes setting of output parameters,

  If you set `LevelTwoMonitorFrequency` to 5 and the resource state is ONLINE, then every fifth monitor cycle, level_two will have the value as 1. If the resource state is OFFLINE, then every monitor cycle level_two will have the value as 2.
  See "LevelTwoMonitorFreq" on page 189.

  If you set `MonitorFreq` to 5 and the resource is registered with IMF, then every fifth monitor cycle level_one parameter will have the value of 1.
  See "IMF" on page 187.

This API is typically used as Perl-based or Shell-based script.

Perl-based:

This API return the value of level_one and level__two and status as return value.

Usage: `($ret, $level_one, $level_two) = VCSAG_GET_MONITOR_LEVEL();`

- $ret : Checks whether the API passed or failed.
- $level_one : Holds the value of level one monitor flag if API is passed.
- $level_two : Holds the value of level two monitor flag if API is passed.

Shell-based:

This API return the value of level_one and level__two as environment variable `VCSAG_MONITOR_LEVEL_ONE` and `VCSAG_MONITOR_LEVEL_TWO`, and status as return value.

Usage: `VCSAG_GET_MONITOR_LEVEL`

Fetches the value of the `LevelOne` and `LevelTwo` monitoring flag as below if API passes,

- `level_one=${VCSAG_MONITOR_LEVEL_ONE}`
- `level_two=${VCSAG_MONITOR_LEVEL_TWO}`

VCSAG_GET_AGFW_VERSION

This API can be used to get the latest agent version.

- **Perl-based:** Returns the version information as return value.
  Usage:
  ```
  my $agfw_ver = VCSAG_GET_AGFW_VERSION();
  ```
- **Shell-based:** Returns the version information in environment variable `VCSAG_AGFW_VERSION_VALUE`, and provides success or failure as return value.
  Usage:
  ```
  VCSAG_GET_AGFW_VERSION agfw_ver=${VCSAG_AGFW_VERSION_VALUE}
  ```

VCSAG_GET_REG_VERSION

This API can be used to get the registered agent version.

- **Perl-based:** Returns the version information as return value.
  Usage:
  ```
  my $agfw_reg_ver = VCSAG_GET_REG_VERSION();
  ```
- **Shell-based:** Returns th version information in environment variable `VCSAG_REG_VERSION_VALUE` , and provides success or failure as return value.
  Usage:
  ```
  VCSAG_GET_REG_VERSION agfw_reg_ver=${VCSAG_REG_VERSION_VALUE}
  ```

VCSAG_SET_RES_EP_TIMEOUT

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

- **Perl-based usage:**

  ```
  VCSAG_SET_RES_EP_TIMEOUT($time);
  ```

- **Shell-based usage:**

  ```
  VCSAG_SET_RES_EP_TIMEOUT${time}
  ```

- **Python-based usage:**

```
VCSAG_SET_RES_EP_TIMEOUT(time)
```

VCSAG_GET_ATTR_VALUE

This API can be used to get the values of attribute. The attribute can be scalar type, key list type, and association type.

Input parameters:

- attribute name: The first argument holds the name of the attribute whose value and index needs to be founded.
- index of attribute: It is optional argument.
  - Should be specified as -1 for getting values of scalar attribute
  - Do not specify this argument, if you need to fetch only the number of keys in key list, vector, association attribute and the index of the attribute.
  - Should be the index of attribute if you need to fetch any particular key from the key list, vector and association attribute.
- index of value required.
  - Should be specified as 1 for fetching the values of scalar attribute.
  - Do not specify this argument, if you need to fetch only the number of keys in key list, vector, association attribute and the index of the attribute.
  - Should be the index of key if you need to fetch the value key from the key list, vector and association attribute.
- arglist : A list of attributes along with values. `ResourceName` and `CleanReason` should not be passed in this list.

Output parameters for Perl-based API:

- ret_val: This API returns value 0 on success and non-zero value on failure. The error gets printed at debug level DBG_1.

Output parameters for Python-based API:

- ret_status: This API returns value 0 on success and non-zero value on failure. The error gets printed at debug level DBG_1.
- ret_values: This API returns tuple in case of success. The first entry of the tuple is the attribute value and second entry of the tuple is the index of attribute in the argument list.

Using `VCSAG_GET_ATTR_VALUE` API to fetch value of scalar attribute

- Shell:

```
VCSAG_GET_ATTR_VALUE "MountPoint" -1 1 @ARGV
```

  The environment variable `VCSAG_ATTR_VALUE` stores the value.

- Perl:

```
my ($ret, $MountPoint) =
VCSAG_GET_ATTR_VALUE ("MountPoint", -1, 1, @ARGV);
```

- Python:

```
ret_status, ret_values =
VCSAG_GET_ATTR_VALUE("MountPoint", -1, 1, *sys.argv)
```

Using `VCSAG_GET_ATTR_VALUE` API to fetch the value of key list, vector and association type attribute

- The user needs to get the number of keys in key list attribute and index of attribute in argument list, and then calls the API in loop. The user can get the key or values in the key list, vector and association attributes.

To get number of keys in the key list attribute and the index of attribute in argument list

Shell:

```
VCSAG_GET_ATTR_VALUE "PidFiles" "$@"
```

The number of values will be stored in environment variable `VCSAG_ATTR_VALUE` , and the environment variable `VCSAG_ATTR_INDEX` holds the index of attribute in the argument list.

Perl:

For example:

```
my ($retval, $NumOfArgs, $indexofattr) =
VCSAG_GET_ATTR_VALUE("ACTION_ARGS", @ARGV);
```

Python:

For example:

```
ret_code, ret_values = VCSAG_GET_ATTR_VALUE ("ACTION_ARGS", *sys.argv)
```

The first entry of ret_values tuple will be the number of values and second entry is the index of attribute in the argument list.

To get a particular key in the key list and vector attribute

Shell:

```
VCSAG_GET_ATTR_VALUE "PidFiles" ${VCSAG_ATTR_INDEX} $index "$@"
```

The variable `VCSAG_ATTR_VALUE` holds the value of the key at the index ($index).

Perl:

```
my ($retval, $value_of_key) =
VCSAG_GET_ATTR_VALUE("ACTION_ARGS", $indexofattr, $index_of_key, @ARGV);
```

Python:

```
ret_code, ret_values =
VCSAG_GET_ATTR_VALUE("ACTION_ARGS", index_of_attr, index_of_key, *sys.argv)
```

By getting the value of the number of keys as mentioned above and by calling this API in the loop, the user can get all the keys of key list and vector attribute.

To get the number of keys in the association attribute, and index of attribute in the argument list

Shell:

```
VCSAG_GET_ATTR_VALUE  "RHEVMInfo"  "$@argv "
```

Perl:

```
my ($ret_val, $ NumOfArgs, $indexofattr) =
VCSAG_GET_ATTR_VALUE ("RHEVMInfo", @argv);
```

Python:

```
ret_code, ret_values = VCSAG_GET_ATTR_VALUE ("RHEVMInfo", *sys.argv)
```

The first entry of ret_values tuple will be the number of values and second entry holds the index of attribute in the argument list.

To get a particular key or value in the association attribute:

Shell:

```
VCSAG_GET_ATTR_VALUE " RHEVMInfo " ${VCSAG_ATTR_INDEX} $index "$@"
```

The variable `VCSAG_ATTR_VALUE` holds the value of key at the index ($index).

Perl:

```
($retval, $value) =
VCSAG_GET_ATTR_VALUE("RHEVMInfo", $indexofattr, $index_of_key_or_val, @argv);
```

Python:

```
ret_code, ret_values =
VCSAG_GET_ATTR_VALUE("RHEVMInfo", index_of_attr, index_of_key_or_val,
*sys.argv)
```

By getting the value of the number of keys as mentioned above and by calling this API in the loop, the user can get all the keys of association attribute.

VCSAG_SET_RESINFO

This API sets or modifies the `ResourceInfo` with specified key and value.

Input parameters:

- info_type: Set to '1' when you call this API for the first time so that the corresponding key value pair can be added to the attribute ResourceInfo, and it is set to '2' second time to update the values of key.
- key_name: Specifies the key that needs to be added or update.
- key_val: Specifies the value of the key that needs to be added or updated.

Output parameters:

- Returns `VCSAG_SUCCESS` when it successful adds or updates the key-value pair.

Shell:

```
VCSAG_SET_RESINFO "${info_type}"  "${key_name}"  "${key_val}"
```

Perl:

```
VCSAG_SET_RESINFO(${info_type}, ${key_name}, ${key_val});
```

VCSAG_MONITOR_EXIT

This API exits the entry point with online/offline/unknown status along with setting the ConfidenceLevel and ConfidenceMsg attributes, if desired.

Input parameters:

- Exit status of resource
  - `VCSAG_RES_UNKNOWN`: Monitor should return this value when resource state is unknown.

  - VCSAG_RES_OFFLINE: Monitor should return this value when resource state is OFFLINE.
  - VCSAG_RES_ONLINE: Monitor should return this value when resource state is ONLINE.
  - VCSAG_RES_INTENTIONALOFFLINE: Monitor should return this value when resource state is detected as intentionally offline.
- New confidence level when exit status is online, else ignored (optional). Confidence level is between 10 to 100%.
- New confidence message when exit status is online but confidence level is below 100%, else ignored (optional)

Perl usage:

- `VCSAG_MONITOR_EXIT($exit_code);`
- `VCSAG_MONITOR_EXIT($exit_code, $confidence_level);`
- `VCSAG_MONITOR_EXIT($exit_code, $confidence_level, $confidence_message);`

Example:

- `VCSAG_MONITOR_EXIT($VCSAG_RES_UNKNOWN);`
- `VCSAG_MONITOR_EXIT($VCSAG_RES_OFFLINE);`
- `VCSAG_MONITOR_EXIT($VCSAG_RES_ONLINE, 90);`
- `VCSAG_MONITOR_EXIT($VCSAG_RES_ONLINE, 20, "block device is 80% full");`

Shell usage:

- `VCSAG_MONITOR_EXIT $exit_code`
- `VCSAG_MONITOR_EXIT $exit_code $confidence_level`
- `VCSAG_MONITOR_EXIT $exit_code $confidence_level $confidence_message`

Example:

- `VCSAG_MONITOR_EXIT $VCSAG_RES_UNKNOWN`
- `VCSAG_MONITOR_EXIT $VCSAG_RES_OFFLINE`
- `VCSAG_MONITOR_EXIT $VCSAG_RES_ONLINE 90`
- `VCSAG_MONITOR_EXIT $VCSAG_RES_ONLINE 20 "block device is 80% full"`

VCSAG_SYSTEM

Entry points must use this function if they need to fork a command using system call.

Shell:

- Input parameter: A string of command with arguments.
- Usage:

```
VCSAG_SYSTEM "$command"; echo $?
```

  User can do `echo $?` to get the exit value of the command.

Perl:

- Input parameter: A string of command with arguments.
- Output parameter : Return value of system ($command).
- Usage:

```
$retval = VCSAG_SYSTEM($command);
```

VCSAG_SU

Entry points must use this function if they need to run a command in a different user's context.

Input parameters:

- User name
- A string of *su* options (if the string is space separated then needs quoted string)
- A string of command with arguments

Output parameters:

- Return value of system($command)

Shell usage:

```
VCSAG_SU "${user}" "-" "${program}"
```

Perl usage:

```
VCSAG_SU($user,"-", $program);
```

VCSAG_RETURN_IMF_RESID

This API is used by imf_register entry point to return the resource ID registered with underlying IMF notification module to the agent.

Shell usage: `VCSAG_RETURN_IMF_RESID`

Perl usage: `VCSAG_RETURN_IMF_RESID()`

VCSAG_RETURN_IMF_EVENT

imf_getnotification entry point uses this API to return the resource ID whose notification arrived from underlying IMF notification module to the agent.

Shell usage: `VCSAG_RETURN_IMF_EVENT`

Perl usage: `VCSAG_RETURN_IMF_EVENT ()`

VCSAG_BLD_PSCOMM

This API builds the ps command based on platform and Container type.

Note: This API is applicable only for Perl-based usage.

Output:

Built PS command. The output can be used to list the processes. The user must call `VCSAG_IS_PROC_IN_CONTAINER` to check if the process lies in the container in which the resource is managed.

Usage:

```
$cmd = VCSAG_BLD_PSCOMM();
open (PIDS, "$cmd |");
```

VCSAG_PHANTOM_STATE

This API determines "phantom" state of a resource, and it requires State and IState of the resource as input arguments

Input parameters: State and IState

Output: "phantom" state of the resource.

Note: This API is applicable only for Perl-based usage.

Perl usage:

```
$ret_state=VCSAG_PHANTOM_STATE($state, $istate);
```

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

See “Script entry point logging functions” on page 128.

VCSAG_LOG_MSG

This API can be used to log messages in the engine log from agent's script entry point.

See “Script entry point logging functions” on page 128.

VCSAG_LOGDBG_MSG

This API can be used to log debug messages in the engine log from agent's script entry point.

See “Script entry point logging functions” on page 128.

VCSAG_SQUEEZE_SPACES

This API removes leading and trailing spaces, and also squeezes the spaces in the value that is passed as arguments.

Input parameters: Any strings with extra spaces.

Output parameters: Space squeezed strings.

Note: This API is applicable only for Perl-based and Python-based usage.

Perl Usage:

```
($a1,$b1...) = VCSAG_SQUEEZE_SPACES ($a, $b...);
```

Example:

```
$str1 = VCSAG_SQUEEZE_SPACES ($str1);
($str1, $str2) = VCSAG_SQUEEZE_SPACES ($str1, $str2);
@str = VCSAG_SQUEEZE_SPACES (@str);
```

Python Usage:

This API accepts a string or a list of strings.

```
str1 = VCSAG_SQUEEZE_SPACES (str1)
str_list = VCSAG_SQUEEZE_SPACES(str_list)
```

Agent Framework primitives with container support

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

Note: Container support is available only with agent version V50 or later.

VCSAG_GET_CONTAINER_BASE_PATH

Note: This API is not supported for Linux Docker containers.

This API returns the base path of the container as mentioned under the ContainerInfo attribute at group level. This API must be called from the global zone or WPAR.

- For Solaris zones:
  If the agent is running on a Solaris machine, the API returns the base path of the zone where zone is installed.
- For WPARs
  If the agent is running on an AIX machine, the API returns the base path of the WPAR where WPAR is installed.

Perl-based: Returns the API exits status, command status and container base path as return value.

Return values:

0	If `VCSAG_GET_CONTAINER_INFO` is called before and ContainerInfo is set properly, which means ■ Name is set to *<valid_container_name>* ■ Type is set to *<valid_container_type>* ■ Enabled is set to 1 Container's base path will be returned as return value.

4	If `VCSAG_GET_CONTAINER_INFO` API is not called before or if ContainerInfo attribute is not set for the resource. For example, Name key of ContainerInfo is "" or Type key of ContainerInfo is invalid etc.
5	If command to fetch base path of container fails. the command status is returned as return value.

Usage:

```
my ($ret, $cmdstatus, $container_base_path) =
VCSAG_GET_CONTAINER_BASE_PATH();
```

Shell-based:

This API returns the base path of container in environment variable `VCSAG_CONTAINER_BASE_PATH` and the status of the command which is used by api to fetch base path name in the environment variable `VCSAG_CMD_STATUS`. The API returns the exit status in the environment variable `VCSAG_BASE_PATH_RET_VAL`.

Return values:

0	If `VCSAG_GET_CONTAINER_INFO` is called before and ContainerInfo is set properly, which means ■ Name is set to <*valid_container_name*> ■ Type is set to <*valid_container_type*> ■ Enabled is set to 1 Container's base path will be stored in environment variable `VCSAG_CONTAINER_BASE_PATH` parameter.
4	If `VCSAG_GET_CONTAINER_INFO` API is not called before or if ContainerInfo attribute is not set for the resource. For example, Name key of ContainerInfo is "" or Type key of ContainerInfo is invalid etc.
5	If command to fetch base path of container fails. The environment variable `VCSAG_CMD_STATUS`, will be updated accordingly with exit status of the command.

Usage:

```
VCSAG_GET_CONTAINER_BASE_PATH
base_path=${VCSAG_CONTAINER_BASE_PATH}
```

Note: Before you use this API, the user should call the API `VCSAG_GET_CONTAINER_INFO`. `VCSAG_GET_CONTAINER_INFO` API will set container name and type appropriately which will be required for this API.

VCSAG_GET_CONTAINER_INFO

Shell:

This API populates `VCSAG_CONTAINER_NAME` and `VCSAG_CONTAINER_TYPE` environment variables appropriately, based on the ContainerInfo attribute passed to ArgList.

Input parameter: ArgList

Output parameter

- Return `VCSAG_INFO_NOT_AVAIL` when ContainerInfo is not passed in the entry point.
- Return `VCSAG_INFO_DONT_CARE` when Container is disabled which means Enabled is set 2 in the ContainerInfo attribute at the group level.
- Return `VCSAG_INFO_AVAIL` when successful. `VCSAG_CONTAINER_NAME` and `VCSAG_CONTAINER_TYPE` will be set appropriately.

Usage:

```
VCSAG_GET_CONTAINER_INFO "$@"
```

Perl:

This API returns the container information, such as container name and container type, under which the resource is managed.

Input parameter: ArgList

Output parameter: Returns the name of container and container type as return value along with success and failure of the API.

Return value:

- `$VCSAG_INFO_NOT_AVAIL` – The Container Info is not available. You cannot use the values *cname* and *ctype*.
- `$VCSAG_INFO_AVAIL` - The Container Info is available. You can use the values *cname* and *ctype*.

- `$VCSAG_INFO_DONT_CARE` - The Container is disabled which means Enabled is set 2 in the ContainerInfo attribute at the group level.

Usage:

```
($ret, $cname, $ctype) = VCSAG_GET_CONTAINER_INFO(@ARGV);
```

VCSAG_IS_PROC_IN_CONTAINER

This API checks if the process is part of the container in which resource is managed. The API `VCSAG_BLD_PSCOMM` should be used for building the *ps* command while finding process name.

Note: This API is not supported for Linux Docker containers. It is applicable only for Perl-based usage and for Solaris Zone, Project, and AIX WPAR.

Input:

$psout - Process entry from the output of ps command as generated by API `VCSAG_BLD_PSCOMM` for the process that needs to be checked.

Return values:

- 1 - Process is part of the Container
- 0 - Process runs outside the Container

Usage:

```
$ret = VCSAG_IS_PROC_IN_CONTAINER($psout);
```

VCSAG_EXEC_IN_CONTAINER

Perl-based or Shell-based:

Executes the command that are passed as an argument to this API inside the appropriate container.

If RIC is set to 1, do not use this API as entry point runs inside the container and zlogin/clogin/newtask command will fail; instead use VCSAG_SYSTEM API.

Input parameter: Command that needs to be run.

Output parameter: Return value of command executed in the appropriate container.

Shell Usage:

```
retval=VCSAG_EXEC_IN_CONTAINER   "$cmd"
```

Perl Usage:

```
$retval = VCSAG_EXEC_IN_CONTAINER($cmd);
```

Note: Before using this API user should call `VCSAG_GET_CONTAINER_INFO`.`VCSAG_GET_CONTAINER_INFO` API will set the container name and type appropriately which are required. This API will execute the command in global container when the user fails to call `VCSAG_GET_CONTAINER_INFO` API.

Example script entry points

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

Online entry point for FileOnOff

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

- For agents that are registered as less than V50, the entry point expects the values of the attributes in the order the attributes have been specified in the ArgList attribute.
- For agents registered as V50 and greater, the entry point expects the ArgList in tuple format: the name of the attribute, the number of elements in the attribute's value, and the value.
  The below mentioned example is applicable for agent version V50 and later.

```
#!/bin/sh
# FileOnOff Online script
# Expects ResourceName and Pathname
. "${CLUSTER_HOME}/bin/ag_i18n_inc.sh"
RESNAME=$1
VCSAG_SET_ENVS $RESNAME
#check if second attribute provided
if [ -z "$4" ]
then
    VCSAG_LOG_MSG "W" "The value for PathName is not
    specified" 1020
else
    #create the file
    touch $4
fi
exit 0;
```

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

Monitor entry point for FileOnOff

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

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

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

The below mentioned example is applicable for agent version V50 and later.

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

. "${CLUSTER_HOME}/bin/ag_i18n_inc.sh"
RESNAME=$1
VCSAG_SET_ENVS $RESNAME
#check if second attribute provided
#Exit with unknown and log error if not provided.
if [ -z "$4" ]
then
    VCSAG_LOG_MSG "W" "The value for PathName is not specified" 1020
    exit 99
else
    if [ -f $4 ]; then exit 110;
    # Exit online (110) if file exists
    # Exit offline (100) if file does not exist
    else exit 100;
    fi
    fi
```

Monitor entry point with intentional offline

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

See “About on-off, on-only, and persistent resources ” on page 17.

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

```
#!/bin/sh

. "${CLUSTER_HOME}/bin/ag_i18n_inc.sh"

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

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

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

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

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

if [ ${REVAL} -eq 0 ]; then
// Found MyCustomAppCode 123 in the application's status
// file that gets created by the application on graceful
```

```
//shutdown
    exit $VCSAG_RES_INTENTIONALOFFLINE;
else
// Did not find MyCustomAppCode 123; hence application has
// crashed or gone down unintentionally
    exit $VCSAG_RES_OFFLINE;
fi

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

Offline entry point for FileOnOff

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

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

The below mentioned example is applicable for agent version V50 and later.

```
#!/bin/sh
# FileOnOff Offline script
# Expects ResourceName and Pathname
#
. "${CLUSTER_HOME}bin/ag_i18n_inc.sh"
RESNAME=$1
VCSAG_SET_ENVS $RESNAME
#check if second attribute provided
if [ -z "$4" ]
then
   VCSAG_LOG_MSG "W" "The value for PathName is not specified"\
   1020
else
#remove the file
   /bin/rm Df $4
fi
exit 0;
```

Monitor entry point for agent having basic (level-1) and detailed (level-2) monitoring

When the agent calls its entry point, the entry point expects the name of the resource as the first argument, followed by the values of the remaining ArgList attributes.

Following implementation is for the agents that are registered as V51. Implementation is same even if the agent is an IMF-aware agent.

Note: IMF is supported for agent version V51 and later.

```
# Implementation of Monitor entry point, which does Level-1
# and Level-2 monitoring.
#

eval 'exec /opt/VRTSperl/bin/perl -I ${CLUSTER_HOME}/lib
 -S $0 ${1+"$@"}'
        if 0;

use strict;
use warnings;

my ($Resource, $state) = ("", "ONLINE");

$Resource = $ARGV[0]; shift;

use ag_i18n_inc;
VCSAG_SET_ENVS ($Resource);

# Fetch the value of level-1 and level-2
my ($ret, $level_one, $level_two)=(0,0,0);
($ret, $level_one, $level_two) = VCSAG_GET_MONITOR_LEVEL();

# Check if level-1 monitoring need to be performed
if ( $level_one == 1 ) {
                # Do level-1 monitoring i.e. basic monitoring
                # This would return state as ONLINE or OFFLINE or
                  unknown
                state = do_level_one_monitoring();
}

# Check if level-2 monitoring need to be performed
if ( ($state eq "ONLINE") && ($level_two != 0) ) {
```

```
                # Do level-2 monitoring i.e. detailed monitoring
                state = do_level_two_monitoring();
}

if ( $state -eq "ONLINE" ) {
                exit 110;
}

if ( $state -eq "OFFLINE" ) {
                exit 100;
}

# unknown state
exit 99;
```

Chapter 5

Logging agent messages

This chapter includes the following topics:

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.

More information is available about how to create and manage messages for internationalization.

See "About internationalized messages" on page 247.

More information is also available about APIs that are used by VCS 3.5 and earlier.

See "Log messages in pre-VCS 4.0 agents" on page 267.

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," for logging applications or debug messages. Moreover, developers of script entry point can configure `LogViaHalog` attribute, to generate application or debug messages using `halog` utility.

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

Agent messages: format

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

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

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

```
Sep 26 2010 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 component, 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 component.

The mnemonic, must use all capital letters. All VCS bundled agents, enterprise agents, and custom agents use the mnemonic: VCS

Severity

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

UMI

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

- The originator ID is a decimal number preceded by a "V-" that defines the component that the message comes from. This ID is assigned by Veritas.
- The category ID is a number in the range of 0 to 65536 assigned by Veritas. The category ID indicates the agent that message came from. For each custom agent, you must contact Veritas so that a unique category ID can be registered for the agent.
  - For C++ messages, the category ID is defined in the `VCSAgStartup` function. See "Log category" on page 124.
  - For script-based entry points, the category is set within the VCSAG_SET_ENVS function See "VCSAG_SET_ENVS" on page 129.
  - For debug messages, the category ID, which is 50 by default, need not be defined within logging functions.
- Message IDs can range from 0 to 65536 for a category ID. Each normal message (that is, non-debug message) generated by an agent must be assigned a message ID. For C++ entry points, the *msgid* is set as part of the VCSAG_LOG_MSG and VCSAG_CONSOLE_LOG_MSG macros. For script-based entry points, the *msgid* is set using the VCSAG_LOG_MSG function. The *msgid* field is not used by debug functions or required in debug messages. See "VCSAG_LOG_MSG" on page 132.

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 129.).

Log unification of VCS agent's entry points

Earlier, the logs of VCS agent's implemented using script and C/C++ language were scattered between the engine log file and agent log file respectively.

From VCS 6.2 version, the logs of all the entry points will be logged in respective agent log file. For example, the logs of Mount agent can be found in the Mount agent log file located under /var/VRTSvcs/log directory.

Moreover, using LogViaHalog attribute, user can switch back to pre VCS 6.2 version log behavior. This attribute support two values 0 and 1. By default the value is 0, which means the agent's log will go into their respective agent log file. If value is set to 1, then the C/C++ entry point's logs will go into the agent log file and the script entry point's logs will go into the engine log file using `halog` command.

Note: Irrespective of the value of LogViaHalog, the script entry point's logs that are executed in the container will go into the engine log file.

C++ agent logging APIs

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

These APIs include two application logging macros:

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

and the macros for debugging:

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

Agent application logging macros for C++ entry points

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

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

`sev`	Severity of the message from the application. The values of `sev` are macros VCS_CRITICAL, VCS_ERROR, VCS_WARNING, VCS_NOTICE, and VCS INFORMATION; see Severity arguments for C++ macros.
`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 specifier 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 (See “LogDbg” on page 189.). 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.
`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 .

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.

More examples of the VCSAG_LOG_INIT macro are available.

See the Examples of logging APIs used in a C++ agent.

Log category

The log category for the agent is defined using the primitive `VCSAgSetLogCategory (cat_ID)` within the `VCSAgStartup` function. In the following example, the log category is set to 10051:

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

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

           VCSAgSetLogCategory(10051);

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

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

Examples of logging APIs used in a C++ agent

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

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

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

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

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

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

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

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

           VCSAG_CONSOLE_LOG_MSG(VCS_WARNING, 3001,
             VCS_DEFAULT_FLAGS,
               "The value for PathName attribute is not
```

```
                specified");

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

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

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

       close(fd);
    }
    return 0;
}

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

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

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

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

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

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

        VCSAgSetLogCategory(2001);

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

}
```

Script entry point logging functions

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

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

The logging functions are available in the ag_i18n_inc module.

VCSAG_SET_ENVS	See "VCSAG_SET_ENVS" on page 129.
VCSAG_LOG_MSG	See "VCSAG_LOG_MSG" on page 132.
VCSAG_LOGDBG_MSG	See "VCSAG_LOGDBG_MSG" on page 134.

Using functions in scripts

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

- Shell Script include file

```
. "${CLUSTER_HOME}/bin/ag_i18n_inc.sh"
```

- Perl Script include file

```
use ag_i18n_inc;
```

- Python Script include file

```
from ag_i18n_inc import *;
```

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 "UMI" on page 119. NOTE: For bundled agents, the category ID is pre-assigned, based on the platform (Solaris, Linux, or AIX) for which the agent is written.

VCSAG_LOG_AGENT_NAME	The absolute path to the agent. For example: UNIX: /opt/VRTSvcs/bin/*resource_type* Since the entry points are invoked using their absolute paths, this environment variable is set at invocation. If the agent developer wishes, this agent name can also be hard coded and passed as an argument to the VCSAG_SET_ENVS function.
VCSAG_LOG_SCRIPT_NAME	The absolute path to the entry point script. For example: UNIX: /opt/VRTSvcs/bin/*resource_type*/online Since the entry points are invoked using their absolute paths, this environment variable is set at invocation. The script name variable is can be overridden.
VCSAG_LOG_RESOURCE_NAME	The resource is specified in the call within the entry point: VCSAG_SET_ENVS $*resource_name*

VCSAG_SET_ENVS examples, Shell script entry points

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

- A minimal call:

```
VCSAG_SET_ENVS ${resource_name}
```

- Setting the category ID:

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

- Overriding the default script name:

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

- Setting the category ID and overriding the script name:

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

Or,

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

VCSAG_SET_ENVS examples, Perl script entry points

- A minimal call:

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

- Setting the category ID:

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

- Overriding the script name:

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

- Setting the category ID and overriding the script name:

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

Or,

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

VCSAG_SET_ENVS examples, Python 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 log all messages ranging in severity from CRITICAL to INFORMATION to the agent log file unless `LogViaHalog` attribute is configured. If `LogViaHalog` is configured then messages will be logged using `halog` command in the engine log.

Note: Messages of the entry points which runs in the container will be logged in the engine log using halog command.

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_LOG_MSG examples, Python 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", "{} files found".format(count), 140, str(count))
```

- Calling a function with one parameter and encoding format:

```
VCSAG_LOG_MSG ("<sev>", "<msg>", <msgid>, "-encoding
   <format>", "<param1>")
VCSAG_LOG_MSG("C", "{} files found".format(count), 140,  "-encoding utf8",
    str(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

The VCSAG_LOGDBG_MSG function can be used to log all debug messages to the agent log file unless LogViaHalog attribute is configured. If LogViaHalog is configured then messages will be logged using `halog` command in the engine log.

At a minimum, the severity must be indicated along with a message. Optionally, the encoding format and parameters may be specified.

Note: Messages of the entry points which runs in the container will be logged in the engine log using halog command.

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

VCSAG_LOGDBG_MSG examples, Shell script entry points

- Calling a function without encoding or parameters:

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

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

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

- Calling a function with a parameter and encoding format:

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

VCSAG_LOGDBG_MSG examples, Perl script entry points

- Calling a function:

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

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

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

- Calling a function with a parameter and encoding format:

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

VCSAG_LOGDBG_MSG examples, Python 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 {}".format(count),
        str(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 {}".format(count), "-encoding
   str(count))
```

Example of logging functions used in a script agent

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

```
!#/bin/ksh

ResName=$1

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


. $VCS_HOME/bin/ag_i18n_inc.sh

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

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

exit 0
```

Chapter 6

Building a custom agent

This chapter includes the following topics:

Files for use in agent development

The VCS installation program provides the Script agents and C++ agents to aid agent development

Script based agent binaries

These are ready to use agent binaries which has in-built VCSAgStartup function implemented. These binaries are located in the directory `$VCS_HOME/bin`.

Following is the list of all script based agent binaries that user can use to build agent

- ScriptAgent with agent framework version V40
- Script50Agent with agent framework version V50
- Script51Agent with agent framework version V51
- Script60Agent with agent framework version V60

For details on the features added in each these agent frame work version please refer:

See "Agent framework versions details" on page 154.

VCS does not support agents that lower then agent version V40. Please refer to *Guidelines for using pre-VCS 4.0 Agents* chapter for using ScriptAgent to work with older agent's entry point.

See "Guidelines for using pre-VCS 4.0 Agents" on page 266.

C++ based agent binaries

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

Table 6-1 C++ Agents

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

Creating the type definition file for a custom agent

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

Naming convention for the type definition file

For example, for the resource type XYZ on Solaris, the file would be `XYZTypes.sun.cf`.

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 };
)
```

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

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

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

You can add the custom type definition to the configuration.

To add the custom type definition to the configuration

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

   UNIX: `$VCS_CONF/conf/config`

2. Add "include FileOnOffTypes.cf" in the main.cf file.
3. Restart VCS.

Building a custom agent on UNIX

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

Note: The `glibc-devel` development package is required for compiling the agent binaries.

The examples assume:

- VCS is installed under `/opt/VRTSvcs` by default. If your installation directory is different, change VCS_HOME accordingly.
- You have created a FileOnOff type definition file.
  See "Creating the type definition file for a custom agent" on page 139.

Note the following about the FileOnOff agent entry points. A FileOnOff resource represents a regular file.

- The FileOnOff `online` entry point creates the file if it does not already exist.
- The FileOnOff `offline` entry point deletes the file.
- The FileOnOff `monitor` entry point returns online and confidence level 100 if the file exists; otherwise, it returns offline.

Implementing entry points using scripts

If entry points are implemented using scripts, the script file must be placed in the directory `$VCS_HOME/bin/`*`resource_type`*. It must be named correctly.

See "About creating entry points in scripts" on page 90.

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

Copy the script based agent binary into the agent directory as `$VCS_HOME/bin/`*`resource_type/resource_type`*`Agent`.

See "Script based agent binaries" on page 138.

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

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

The agent binary for Oracle would be `$VCS_HOME/bin/Oracle/OracleAgent`, for example.

If the agent file is different, for example `/foo/ora_agent`, the `types.cf` file must contain the following entry:

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

Example: Using script entry points on UNIX

The following example shows how to build the FileOnOff agent using scripts. For the below example, we are using Script51Agent script based agent binary.This example implements the `online`, `offline`, and `monitor` entry points only.

See "Script based agent binaries" on page 138.

Example: implementing entry points using scripts

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

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

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

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

   or create a link.

   To copy the agent binary:

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

   To create a link to the agent binary:

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

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

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

```
# !/bin/sh
# Create the file specified by the PathName
# attribute.
  touch $4
  exit 0
```

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

```
# !/bin/sh
# Remove the file specified by the PathName
# attribute.
  rm $4
  exit 0
```

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

```
# !/bin/sh
# Verify file specified by the PathName attribute
# exists.
  if test -f $4
  then
    exit 110;
  else
    exit 100;
  fi
```

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

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

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

To implement the agent using VCSAgStartup function and script entry points

1 Create the following directory:

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

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

```
cp -r /opt/VRTSvcs/src/agent/Sample/*
       /opt/VRTSvcs/src/agent/FileOnOff
```

3 Change to the new directory:

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

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

```
void VCSAgStartup() {
    VCSAgInitEntryPointStruct(V51);

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

    VCSAgSetLogCategory(10041);
    }
```

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

```
gmake
```

6 Create a directory for the agent:

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

7 Install the FileOnOff agent.

```
make install AGENT=FileOnOff
```

8 Implement the `online`, `offline`, and `monitor` entry points.

See Example: Using script entry points on UNIX.

Implementing entry points using C++

You can implement entry points by using C++.

To implement entry points by using C++

1. Edit `agent.C` to customize the implementation; `agent.C` is located in the directory `$VCS_HOME/src/agent/Sample`.
2. After completing the changes to `agent.C`, invoke the `make` command to build the agent. The command is invoked from `$VCS_HOME/src/agent/Sample`, where the Makefile is located.
3. Name the agent binary: *`resource_type`*`Agent`.
4. Place the agent in the directory `$VCS_HOME/bin/`*`resource_type`*.

   For example, the agent binary for Oracle would be `$VCS_HOME/bin/Oracle/OracleAgent`.

Example: Using C++ entry points on UNIX

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

To use VCSAgStartup and C++ entry points

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

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

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

2 Modify `res_online()`:

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

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

    int fd = -1;
    int ret = 0;
    char *pathname = NULL;

    VCSAG_LOG_INIT("res_online");

    /*
     * Get PathName attribute form attr_val parameter, passed to
res_online function and store
     * it under pathname variable.
     *
     */

     if (NULL == pathname) {
         return 0;
     }

     VCSAG_LOGDBG_MSG(VCS_DBG2, VCS_DEFAULT_FLAGS,
         "Creating file %s", pathname);

     if ((fd = open(pathname, S_IRUSR|S_IWUSR)) < 0) {
         VCSAG_LOG_MSG(VCS_ERROR, 2003, VCS_DEFAULT_FLAGS,
             "Attempt to create the file failed with errno=%d",
errno);
         VCSAG_CONSOLE_LOG_MSG(VCS_ERROR, 2003, VCS_DEFAULT_FLAGS,
             "Attempt to create the file failed with errno=%d",
errno);
    } else {
        close(fd);
    }
```

```
    return 0;
    }
```

3 Modify `res_offline()`:

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

  unsigned int res_offline(const char *res_name, void **attr_val)
    {
    char *pathname = NULL;

    VCSAG_LOG_INIT("res_offline");

    /*
     * Get PathName attribute form attr_val parameter, passed to
res_offline function and store
     * under pathname variable.
     *
     */

    if (NULL == pathname) {
        return 0; /* success: nothing to remove */
    }

    VCSAG_LOGDBG_MSG(VCS_DBG2, VCS_DEFAULT_FLAGS,
        "Removing file %s", pathname);

    if ((0 != remove(pathname)) && (ENOENT != errno)) {
        VCSAG_LOG_MSG(VCS_ERROR, 2002, VCS_DEFAULT_FLAGS,
            "Attempt to remove the file failed with errno=%d",
errno);
        VCSAG_CONSOLE_LOG_MSG(VCS_ERROR, 2002, VCS_DEFAULT_FLAGS,
            "Attempt to remove the file failed with errno=%d",
errno);

        return 1; /* failure: attempt to remove failed */
    }

    return 0; /* success: file removed */
    }
```

4 Modify the `res_monitor()`, function.

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

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

```
make
```

6 Create the directory for the agent binaries:

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

7 Install the FileOnOff agent.

```
make install AGENT=FileOnOff
```

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

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

To implement the agent using VCSAgStartup, C++, and script entry points

1 Create a directory for the agent:

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

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

```
cp -r /opt/VRTSvcs/src/agent/Sample/*
                /opt/VRTSvcs/src/agent/FileOnOff
```

3 Change to the new directory:

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

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

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

    VCSAgSetLogCategory(10051);
    VCSAgInitEntryPointStruct(V51);

    VCSAgValidateAndSetEntryPoint(VCSAgEPMonitor, res_monitor);
}
```

5 Modify the `res_monitor()` function:

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

   VCSAgResState res_monitor(const char *res_name, void
       **attr_val, int *conf_level)
   {
   int ret = 0;
   char *pathname = NULL;
   struct stat64 stat_buf;
   VCSAgResState state = VCSAgResUnknown;

   VCSAG_LOG_INIT("res_monitor");

   /*
    * Get PathName attribute form attr_val parameter, passed to
res_offline function and store
    * under pathname variable.
    *
    */
   if (NULL == pathname) {
       return VCSAgResUnknown;
   }

   VCSAG_LOGDBG_MSG(VCS_DBG2, VCS_DEFAULT_FLAGS,
       "Checking if file %s exists or not", pathname);

   if (0 == stat64(pathname, &stat_buf)) {
       /*
        * If the pathname is a directory, return status as unknown
        */
       if (S_ISDIR(stat_buf.st_mode) != 0) {
           VCSAG_LOG_MSG(VCS_ERROR, 2004, VCS_DEFAULT_FLAGS,
               "%s is a directory", pathname);
           VCSAG_CONSOLE_LOG_MSG(VCS_ERROR, 2004,
VCS_DEFAULT_FLAGS,
                "%s is a directory", pathname);

            *conf_level = 0;
            return VCSAgResUnknown;
```

```
        }
        *conf_level = 100;
        return VCSAgResOnline;
    }

    *conf_level = 0;
    return VCSAgResOffline;
    }
```

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

```
make
```

7 Build the script entry points for the agent.

See Example: Using script entry points on UNIX

8 Create a directory for the agent:

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

9 Install the `FileOnOff` agent.

Installing the custom agent

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

On UNIX:

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

Make sure you create the custom_type directory at only one of these locations.

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

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

Defining resources for the custom resource type

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

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

See the *Cluster Server User's Guide* for more information.

Sample resource definition

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

```
include types.cf
.
.
.
        FileOnOff temp_file1 (
        PathName = "/tmp/test"
        )
```

The type FileOnOff is defined in the file `types.cf`. The file `types.cf` is included in `main.cf` using the `include` directive.The resource defined in the `main.cf` file specifies:

- The resource type: `FileOnOff`
- The name of the resource, `temp_file1`
- The name of the attribute, `PathName`
- The value for the `PathName` attribute:

```
On UNIX:"/tmp/test"
```

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

How the FileOnOff agent uses configuration information

The information in the VCS configuration is passed by the engine to the FileOnOff agent when the agent starts up on a node in the cluster. 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. It also sends all the attribute and their repective value details to the agent.

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

Agent framework versions details

The following table describes the various agent binaries and its functions.

Agent binary	Description
Agent version V40	The support for action and info entry point is available for agents with agent version V40 or later. See "About the info entry point" on page 31. See "About the action entry point" on page 30.
Agent version V50	The AEPTimeout attribute feature is available for agents registered with version V50 or later. See "AEPTimeout" on page 180. Support for positional independent ArgListValues is available in agent framework version V50 and later. See "About the ArgList and ArgListValues attributes" on page 44. The container support on AIX and Solaris is available in agent framework version V50 or later for containers solaris zones, solaris project and aix wpars. See "ContainerOpts" on page 184.

Agent binary	Description
Agent version V51	Intentional offline and IMF features are available for agents framework version V51 or later. See "IntentionalOffline" on page 189. See "About intelligent monitoring framework (IMF) " on page 16. There are three new entry points for this feature ■ imf_init ■ imf_register ■ imf_getnotificaiton See "About building a script based IMF-aware custom agent" on page 156.
Agent version V60	Metering and migration features are available in the agent framework version V60. Entry points meters and migrate is added with this feature. See "About the migrate entry point" on page 36. See "About the meter entry point" on page 36. **Note:** Veritas recommends using the latest agent version.

Chapter 7

Building a script based IMF-aware custom agent

This chapter includes the following topics:

About building a script based IMF-aware custom agent

This chapter explains how you can build a script-based IMF-aware custom agent. VCS supports only process and script-based IMF-aware custom agents from VCS 6.0.1 and later release. The process to build a custom agent (without IMF) is similar to what is described in the previous chapter.

The following IMF entry points have been introduced in VCS 5.1SP1 to enable IMF for intelligent monitoring:

- imf_init
- imf_register
- imf_getnotification

You must use the above-stated IMF entry points along with the other entry points if you want the IMF feature enabled for your custom agent. Veritas supports only the AMF plugins while implementing these entry points.

See "About agent entry points" on page 25.

See "Syntax for the imf_init script " on page 95.

See "Syntax for the imf_register script " on page 95.

See "Syntax for the imf_getnotification script " on page 96.

Building a script based IMF-aware agent involves the following steps:

1. Linking AMF plugins with the script agent.
2. Creating XML file (amfregister.xml) required for AMF plugins to do resource registration for online and offline state monitoring.
3. Adding IMF and IMFRegList attributes in configuration files

   See "Adding IMF and IMFRegList attributes in configuration" on page 165.
4. Installing the custom script based agent to enable IMF. See "Installing the IMF-aware script-based custom agent" on page 170.

Linking AMF plugins with script agent

Change the current working directory to agent specific directory, and in the agent specific directory, create symbolic links (soft links) to the AMF plugins using the following commands:

```
ln -s /opt/VRTSamf/imf/imf_init imf_init
ln -s /opt/VRTSamf/imf/imf_register imf_register
ln -s /opt/VRTSamf/imf/imf_getnotification imf_getnotification
```

Creating XML file required for AMF plugins to do resource registration for online and offline state monitoring

Create the amfregister.xml file that is used by imf_register entry point to do registration of process-based resource for online and offline monitoring with AMF.

Since imf_register entry point is a generic script used by different agents to register resources for online and offline monitoring, you must specify what needs to be registered for a resource of a particular type with the help of amfregister.xml. You can refer the following table description to know about the tags used in amfregister.xml.

Table 7-1 Common tags for the amfregister.xml file

Tag name	Description
RegType	This tag is used to specify the type of registration. This tag is common between PRON and PROFF - specific tags. Set it to PROFF to do resource registration with AMF for process offline monitoring. Set it to PRON to do resource registration with AMF for process online monitoring.
ReaperName	It contains the reaper name (type name) for the agent. Agent will be registered with this name in the IMF notification module.

Table 7-2 PRON-specific tags

Tag name	Description
ProcPattern	Indicates how the process-based resource shows up in the process table. The specified ProcPattern is searched in the process table and the corresponding pid is registered with AMF for online monitoring.
PronOptions	Specifies additional options for ProcPattern matching. If it is set to IGNORE_ARGS, the value specified in ProcPattern is considered as the process path. While matching against the process table entries, only the process path is matched against the ProcPattern. The pid of the matching process is registered with AMF. If PronOptions is set to IGNORE_PATH, the value mentioned in the ProcPattern is considered as the process name followed by the process args. While matching against the process table entries, only the base name of the process path and the process args are matched against ProcPattern. The pid of the matching process is registered with AMF. If the PronOptions is set to IGNORE_ARGS IGNORE_PATH, the value mentioned in the ProcPattern is considered as just the process name. While matching against the process table entries, only the base name of the process path is matched against the process name mentioned in the ProcPattern. The arguments of the process are not considered. The pid of the matching process is registered with AMF.

Table 7-3 PROFF-specific tags

Tag name	Description
Owner	It is the user who executes this process. The UID and GID of this user are used to register the PROFF event with AMF.
Path	Complete path of the binary.
arg0	Name with which the binary is executed. If it is executed with the complete path, you need not provide this.
arg0flag	This is used to further refine the matching behavior of arg0. The string specified in arg0 (for example: string A) is searched inside the arg0 of the process that is being matched against (for example: string PA). This can have the following values: ■ FREE: Look for string A in string PA using a free substring match operation. ■ BOUNDLEFT: Look for string A in string PA using a left-bounded substring match operation. ■ BOUNDRIGHT: Look for string A in string PA using a right-bounded substring match operation. ■ EXACT: String A must exactly match string PA. If arg0flag is not provided, the default value for arg0flag is considered as EXACT. **Note:** BOUNDLEFT and BOUNDRIGHT can be specified together, separated by a space. BOUNDLEFT and BOUNDRIGHT specified together is not the same as EXACT.
args	Arguments with which the binary is executed. This is used for matching while finding the process in the process table. Here it looks for exact match.
ArgsSubString	Apart from the default exact match of the arguments, you can specify a list of substrings that must appear in the argument list of the process that is being matched against. Using the ArgsSubString tag, you can specify one substring. You can specify up to 8 such substrings. **Note:** If ArgsSubString is provided, you must not provide args.
ArgsSubStringFlag	For each substring specified, you can specify additional flags to control the matching behavior. Each substring specified (for example: string SS) is searched inside args of the process that is being matched against (for example: string PA). ■ FREE: Look for string SS in string PA using a free substring match operation. ■ BOUNDLEFT: Look for string SS in string PA using a left-bounded substring match operation. ■ BOUNDRIGHT: Look for string SS in string PA using a right-bounded substring match operation. Every ArgsSubString shall have a corresponding ArgsSubStringFlag. If no ArgsSubStringFlag is provided, the default flag is FREE.

Table 7-3 PROFF-specific tags *(continued)*

Tag name	Description
ArgsFlag	This tag is used to control the behavior of overall substring matching. It can have the following values: ■ MATCH_ALL: Match all substrings specified. ■ MATCH_ANY: Match any of the substrings specified. ■ IGNORE_ARGS: Ignore the args completely. You must not provide this if ArgsSubString or args tags are provided. A set of zero or more ArgsSubString and ArgsSubStringFlag tags can be followed by an optional tag ArgsFlag. If no ArgsFlag is provided, MATCH_ALL is considered as the default value for ArgsFlag.

Example of amfregister.xml for registration of process-based resource with AMF for online monitoring

Assuming the process in the `ps` output is displayed as follows, you can use the subsequent steps to register a process-based resource for online monitoring:

```
"/usr/sbin/rpc.statd -d 0 -t 50"
```

1. If you are sure about the path and arguments, you must specify your process in the following format in the amfregister.xml file:

```
<xml>
     <Register>
        <RegType>PRON</RegType>
        <ProcPattern>/usr/sbin/rpc.statd -d 0 -t 50</ProcPattern>
     </Register>
</xml>
```

2. If you are not sure about the arguments but are sure about the path, you must specify your process in the following format in the amfregister.xml file:

```
<xml>
     <Register>
        <RegType>PRON</RegType>
        <PronOptions>IGNORE_ARGS</PronOptions>
        <ProcPattern>/usr/sbin/rpc.statd</ProcPattern>
     </Register>
</xml>
```

Note: If there are more than one processes or instances with different arguments, all get registered.

For example:

```
"/usr/sbin/rpc.statd -d 0 -t 50"
"/usr/sbin/rpc.statd -xyz"
```

Both the above processes get registered with AMF.

3. If you are not sure about the path but are sure about the arguments, you must use the following format of the amfregister.xml:

```
<xml>
     <Register>
        <RegType>PRON</RegType>
        <PronOptions>IGNORE_PATH</PronOptions>
        <ProcPattern>rpc.statd -d 0 -t 50</ProcPattern>
     </Register>
</xml>
```

Note: If there are more than one processes/instances with different paths, all get registered.

For example:

```
"/usr/sbin/rpc.statd -d 0 -t 50"
"/home/<testuser>/rpc.statd -d 0 -t 50"
```

4. If you are not sure about the path or the arguments, you must use the following format of the amfregister.xml:

```
<xml>
     <Register>
        <RegType>PRON</RegType>
        <PronOptions>IGNORE_ARGS IGNORE_PATH</PronOptions>
        <ProcPattern>rpc.statd</ProcPattern>
     </Register>
</xml>
```

Note: If there are more than one processes with the same base name, all get registered irrespective of the path and arguments.

For example:

```
"/usr/sbin/rpc.statd -d 0"
"/home/<testuser>/rpc.statd -d 0 -t 50"
```

Example of amfregister.xml for registration of process-based resource with AMF for offline monitoring

- Example 1:
  - Process name: xyz
  - Complete path of the process: `/MyHome/veritas/xyz`
  - Arguments: -p abc -t qwe -m40
  - Process owner : vcsuser

```
<xml>
     <Register>
          <RegType>PROFF</RegType>
          <Owner>vcsuser</Owner>
          <Path>/MyHome/veritas/xyz</Path>
          <arg0>xyz</arg0>
          <args>-p abc -t qwe -m40</args>
     </Register>
</xml>
```

- Example 2: To register a process-based resource with AMF for offline monitoring assuming OwnerName and HomeDir as VCS attributes the amfregister.xml has the following format:

```
<xml>
      <Register>
         <RegType>PROFF</RegType>
         <Owner>${OwnerName}</Owner>
         <Path>${HomeDir}/veritas/xyz</Path>
         <arg0>xyz</arg0>
         <args>-p abc -t qwe -m40</args>
```

```
      </Register>
</xml>
```

- Example 3: To register a process-based resource with AMF for offline monitoring using substring matching for the arguments:
  - Process: xyz
  - Complete path of the process: /MyHome/veritas/xyz
  - Arguments substrings:
    - -t qwe with left bounded substring matching
    - -m 40 with right bounded substring matching
  - Argument flags: Match any of the substring

```
<xml>
   <Register>
     <RegType>PROFF</RegType
     <Path>/MyHome/veritas/xyz</Path>
     <arg0>xyz</arg0>
     <arg0flag>EXACT</arg0flag>
     <ArgsSubString>-t qwe</ArgsSubString>
     <ArgsSubStringFlag>BOUNDLEFT</ArgsSubStringFlag>
     <ArgsSubString>-m 40</ArgsSubString>
     <ArgsSubStringFlag>BOUNDRIGHT</ArgsSubStringFlag>
     <ArgsFlag>MATCH_ANY</ArgsFlag>
   </Register>
</xml>
```

Example of amfregister.xml for online and offline IMF monitoring for a given process

To register a process-based resource with AMF for online and offline monitoring with:

- Path: /opt/VRTSamf/bin/amfstat
- argv0: amfstat
- args: -s 5

```
<xml>
   <Register>
```

```
        <RegType>PRON</RegType>
        <ProcPattern>/opt/VRTSamf/bin/amfstat -s 5</ProcPattern>
    </Register>
    <Register>
        <RegType>PROFF</RegType>
        <Path>/opt/VRTSamf/bin/amfstat</Path>
        <arg0>amfstat</arg0>
        <args>-s 5</args>
    </Register>
</xml>
```

Examples for adding RepearName tag in amfregister.xml

The RepearName tag can be added both manually and automatically in the amfregister.xml.

- Adding ReaperName tag automatically
  The tag is created automatically when imf_init for the agent is called for the first time if not already present.
  For example, if the type name is CFSMount, the amfregister.xml will have following lines:

```
<xml>
   <!--ReaperName tag has been added by imf_init entry point-->

   <ReaperName>CFSMount</ReaperName>

     <Register>

       <RegType>PRON</RegType>

         <ProcPattern>/usr/sbin/rpc.statd -d 0 -t 50</ProcPattern>
     </Register>
</xml>
```

- Adding ReaperName tag manually
  Agent Developer may also choose to add this tag manually. In this case, imf_init will not update amfregister.xml.
  For example, if the reaper name (type of the resource) is Process, the xml file will look as follows:

```
<xml>
  <ReaperName>Process</ReaperName>
```

```
    <Register>

      <RegType>PRON</RegType>

        <ProcPattern>/usr/sbin/rpc.statd -d 0 -t 50</ProcPattern>

    </Register>
</xml>
```

Adding IMF and IMFRegList attributes in configuration

You need to add IMF and IMFRegList attribute in configuration files. Adding IMFRegList is optional.

See “IMF” on page 187.

See “IMFRegList” on page 188.

To add these attributes, you can either modify the configuration file if VCS is not running or modify the running configuration by `ha -` command if VCS is running. Refer to the following examples for this purpose.

Example of type definition for a custom agent to supports IMF when VCS is not running:

```
type MyCustomIMFApp (
    static int IMF{} = { Mode=3, MonitorFreq=1, RegisterRetryLimit=3 }
    static str IMFRegList[] = { PathName, Arguments }
    static str ArgList[] = { PathName, Arguments, HomeDir }
    str PathName
    str Arguments
    str HomeDir
)
```

Example of modify configuration for a custom agent to supports IMF when VCS is running:

Command to add IMF attribute:

```
haattr -add -static IMF MyCustomIMFApp -integer -assoc Mode 3
MonitorFreq 1 RegisterRetryLimit 3
```

Command to add IMFRegList attribute:

```
haattr -add -static Zone IMFRegList -string -vector PathName,
Arguments
```

See "IMFRegList" on page 188.

You can modify these IMF attribute values at the Agent Type or Resource level to suite your requirement. The following example describes how you can modify the values for Mode attribute.

In case of Online only Monitoring (PRON), Mode value can be set to 2. Run the following commands at the respective levels to modify the Mode value:

1 At the type level:

```
# hatype -display <resource-type> -attribute IMF
#Type         Attribute Value
CustomProcess IMF  Mode 3 MonitorFreq 1 RegisterRetryLimit 3

# hatype -modify <resource-type> IMF -update Mode 2

# hatype -display <resource-type> -attribute IMF
#Type         Attribute Value
CustomProcess IMF Mode 2 MonitorFreq 1 RegisterRetryLimit 3
```

2 At the Resource level, first check that whether static attribute IMF is overridden or not.

```
# hares -display <resource-name> -attribute IMF
VCS WARNING V-16-1-10554 No resource exists with attribute IMF
```

In case not overridden , you can now override static attribute IMF at resource level using following command:

```
# hares -override <resource-name> IMF
# hares -display <resource-name> -attribute IMF
#Resource Attribute System Value
pres1     IMF global Mode 3 MonitorFreq 1 RegisterRetryLimit 3

# hares -modify pres1 IMF -update Mode 2

# hares -display pres1 -attribute IMF
#Resource Attribute System Value
pres1     IMF global Mode 2 MonitorFreq 1 RegisterRetryLimit 3
```

Monitor without IMF integration

Monitor without IMF integration and having basic (Level-1) monitoring:

```
#!/bin/sh
# CustomAgent Monitor script
. $VCS_HOME/bin/ag_i18n_inc.sh
RESNAME=$1
VCSAG_SET_ENVS $RESNAME

# Logic for custom agent resource monitoring.
# Based on logic set STATE to "OFFLINE" or "ONLINE"
if resource is found in either OFFLINE or ONLINE state.

if [ ${STATE} = "OFFLINE" ]
then
 exit ${STATE}
fi
```

Monitor without IMF but with LevelTwo monitor frequency

If the custom agent monitor does the basic as well as detail monitoring, then detail monitoring code must be conditional. This avoids scheduling of detail monitoring if not required. VCSAG_GET_MONITOR_LEVEL API can be used to check if detail monitoring needs to be scheduled.

VCSAG_GET_MONITOR_LEVEL API fetches and sets the values of the LevelTwoMonitorFreq attribute.

```
#!/bin/sh
# CustomAgent Monitor script
. $VCS_HOME/bin/ag_i18n_inc.sh
RESNAME=$1
VCSAG_SET_ENVS $RESNAME

STATE=${VCS_RES_ONLINE};

# Fetch the value of detail (Level-2) monitoring.
# VCSAG_GET_MONITOR_LEVEL will store this values in
# VCSAG_MONITOR_LEVEL_TWO environment.
VCSAG_GET_MONITOR_LEVEL();
```

```
# Logic for custom agent basic monitoring.
# Based on logic set STATE to OFFLINE or ONLINE


# if basic monitoring of the resource state that resource is ONLINE,
# check if detail monitoring (Level-2) need to be performed.
if [ ${STATE} -eq ${VCS_RES_ONLINE} ]; then

 if [ ${VCSAG_MONITOR_LEVEL_TWO} -ne 0 ]; then
  # Logic for custom agent detail monitoring.
  # Based on logic return OFFLINE or ONLINE

  # If resource is found as OFFLINE
  STATE = ${VCS_RES_OFFLINE};

  # If resource is found as ONLINE
  STATE = ${VCS_RES_ONLINE};
 fi
fi

exit $(STATE);
```

Monitor with IMF integration

If the custom agent monitor does only basic monitoring, then you need not make any changes in the existing monitor entry point.

```
#!/bin/sh
# CustomAgent Monitor script
. $VCS_HOME/bin/ag_i18n_inc.sh
RESNAME=$1
VCSAG_SET_ENVS $RESNAME

# Logic for custom agent resource monitoring.
# Based on logic set STATE to "OFFLINE" or "ONLINE"
if resource is found in either OFFLINE or ONLINE state.

if [ ${STATE} = "OFFLINE" ]
then
 exit ${STATE}
fi
```

```
if [ ${STATE} = "ONLINE" ]
then
 exit ${STATE}
fi
```

Monitor with IMF but with LevelTwo monitor frequency

If the custom agent monitor does the basic as well as detail monitoring, then the basic monitoring code must be conditional. This avoids scheduling of basic monitoring if only detail monitoring is required to be scheduled. VCSAG_GET_MONITOR_LEVEL api can be used to check if basic, detail or both monitoring is required to be scheduled.

Using api VCSAG_GET_MONITOR_LEVEL monitor entry point can decides whether to perform basic and detail monitoring based on the values of MonitorFreq and LevelTwoMonitorFreq attributes respectively.

```
#!/bin/sh
# CustomAgent Monitor script
. $VCS_HOME/bin/ag_i18n_inc.sh
RESNAME=$1
VCSAG_SET_ENVS $RESNAME

STATE=${VCS_RES_ONLINE};

# Fetch the value of basic (Level-1) and detail (Level-2) monitoring.
# VCSAG_GET_MONITOR_LEVEL will store these values in
# VCSAG_MONITOR_LEVEL_ONE and CSAG_MONITOR_LEVEL_TWO environment.
VCSAG_GET_MONITOR_LEVEL();

# Check if basic monitoring (Level-1) need to be performed.
if [ ${VCSAG_MONITOR_LEVEL_ONE} -ne 0 ]; then

 # Logic for custom agent basic monitoring.
 # Based on logic set STATE to OFFLINE or ONLINE

 # If resource is found as OFFLINE
 STATE = ${VCS_RES_OFFLINE};

 # If resource is found as ONLINE
```

```
 STATE = ${VCS_RES_ONLINE};
fi

# if basic monitoring of the resource state that resource is ONLINE,
# check if detail monitoring (Level-2) need to be performed.
if [ ${STATE} -eq ${VCS_RES_ONLINE} ]; then

 if [ ${VCSAG_MONITOR_LEVEL_TWO} -ne 0 ]; then
  # Logic for custom agent detail monitoring.
  # Based on logic return OFFLINE or ONLINE

  # If resource is found as OFFLINE
  STATE = ${VCS_RES_OFFLINE};

  # If resource is found as ONLINE
  STATE = ${VCS_RES_ONLINE};
 fi
fi

exit $(STATE);
```

Installing the IMF-aware script-based custom agent

The procedure to install the custom script based-agent is similar to installing the custom agent. See “Installing the custom agent” on page 152.for more information.

Chapter 8

Testing agents

This chapter includes the following topics:

About testing agents

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

Using debug messages

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

See “LogDbg” on page 189.

Debugging agent functions (entry points).

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

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

To set debug severities for a particular resource-type:

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

To remove debug severities for a particular resource-type:

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

To remove all debug severities:

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

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

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

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 (UNIX: $VCS_LOG/log/<resource_type>_A.log) and from script entry points will get printed to the HAD log file. An example line from the agent log file:

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

Debugging the agent framework

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

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

Debugging using AdvDbg attribute

You can activate advanced debugging by setting the value of AdvDbg attribute. If configured, this directs the agent framework to invoke the following predefined actions on entry point time out:

- pstack: Used to generate the process tree or process stack or both.
- core: Used to generate the core of the agent process.

A process can be the agent process or any command executed from the agent entry point. All information is captured under `$VCS_LOG/diag/agents/<Agent Name>` directory.

Working of AdvDbg attribute

To understand how data is captured, you need to understand the variety of agents. Agents are divided into three varieties:

1. Agents with all entry points written in script.
2. Agents with all entry points implemented using C/C++. The implementation of these entry points is such that:
   - Some commands get invoked from the entry points
   - Only system calls (but no commands) get executed from the entry points.
3. Hybrid agents where some entry points are written using scripts and remaining using C/C++.

Working of pstack action

When configured with this action, the agent framework captures the process stack or process stack with the process tree on entry point timeout. The agent framework takes a decision internally to capture the process tree along with the process stack. In the following cases, agent framework also captures the process tree along with the process stack for the process:

- If the timed out entry point is implemented using script, the agent framework captures the process tree as well as the process stack.
- If the timed out entry point is implemented in C/C++ and waits for some command to complete, the agent framework captures the process tree along with the process stack.

In all the other cases, the agent framework captures the process stack of the agent process. All the information related to process stack and process tree is captured in the `FFDC_AGFWEP<ep_name>_A.log` file. This file contains the information of all the resources of the same type for which <ep_name> entry point has timed out. When file `FFDC_AGFWEP<ep_name>_A.log` is full, the current data is moved to `FFDC_AGFWEP<ep_name>_B.log` for backup.

Working of core action

When configured with this action, the agent framework captures the process core on the entry point timeout. In this release, agent framework only supports capturing of agent core. The core file is named as `core.<AgentPID>.<res_name>.<ep_name>`. This means that core is generated when entry point <ep_name> of the <res_name> resource is timed out. The core of the agent process gets generated in the last timeout of the entry point. For a given agent, if the same entry point time-outs multiple times for a same resource then this core will be obtained at the time of last timeout.

Impact of AdvDbg attribute on existing functionality of the entry point

You must clearly understand and consider the following impact while configuring the AdvDbg attribute:

- Processing of other resources may get delayed as one thread is occupied in capturing the debug information.
- System resources are consumed by extra activities like core generation, process stack and process tree generation.
- Agent may stop responding due to low system resources.

Considering the above impact, the default action can be set for rarely occurring or one-time occurring events like open, close, offline, online, or clean. For other entry

points, like monitor, if you face timeout issues, AdvDbg must be configured dynamically by overriding this attribute to the resource level for that resource which faces entry point timeout. As soon as the data is captured, AdvDbg attribute must be cleared for these entry points. If multiple resources of the same entry point are timing out, then choose only one resource to configure the AdvDbg attribute.

See "Recommended steps for configuring AdvDbg attribute for monitor entry points" on page 180.

Using the engine process to test agents

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

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

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

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

For example, to test the Oracle agent, type:

```
haagent -display Oracle
```

If the Oracle agent is running, the output resembles:

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

Test commands

The following examples show how to use commands to test the agent:

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

  ```
  hatype -modify <resource_type> LogDbg -add DBG_AGINFO
  ```

- To check the status of a resource, type:

  ```
  hares -display <resource_name>
  ```

- To bring a resource online, type:

  ```
  hares -online <resource_name> -sys system
  ```

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

- To take a resource offline, type:

  ```
  hares -offline <resource_name> -sys system
  ```

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

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

  ```
  hatype -modify <resource_type> LogDbg -delete DBG_AGINFO
  ```

Chapter 9

Static type attributes

This chapter includes the following topics:

About static attributes

Predefined static resource type attributes apply to all resource types.

See “Static type attribute definitions” on page 178.

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 in two ways:

- By explicitly defining the attribute in a resource definition in configuration file (main.cf) when VCS is not running.
- By using the hares command from the command line with the `-override` option when VCS is running.

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 *Cluster Server Administrator's Guide* for additional information about overriding the values of static attributes.

Static type attribute definitions

The following sections describe the static attributes for agents.

ActionTimeout

After the `hares -action` command has instructed the agent to perform a specified action, the agent waits for the number of seconds as specified in the ActionTimeout attribute (scalar-integer) to let the action entry point finish and after the time limit has been reached the agent terminates the running action entry as it might hung at any point. The value of ActionTimeout may be set for individual resources, if overridden.

The default is 30 seconds. The value of the ActionTimeout attribute is internally capped at MonitorInterval / 2.

If the ActionTimeout attribute is set to a value greater than MonitorInterval/2, then MonitorInterval/2 is used instead of ActionTimeout. If ActionTimeout value is less than MonitorInterval/2, then the ActionTimeout value is honored.

Note: You can extend this value by using the VCSAgSetResEPTimeout (for C/C++ entry point) /VCSAG_SET_RES_EP_TIMEOUT (for script entry point). Use this API cautiously as setting a value higher than MonitorInterval / 2 might result in delay of next periodic monitor which is used to check the state of the resource.

See "VCSAgSetResEPTimeout" on page 81.

See "VCSAG_SET_RES_EP_TIMEOUT" on page 98.

AdvDbg

AdvDbg attribute helps VCS agents to capture advance information like process stack, process tree, and core on unexpected events. This is helpful in RCA and saves time in troubleshooting the unexpected event. The present scope of unexpected event is entry point timeout.

This is a KeyList attribute where each key defines an action that gets executed when an entry point times out.

See "Debugging using AdvDbg attribute" on page 173.

Configuring AdvDbg attribute and formatting the individual key

AdvDbg attribute is a keylist attribute and the format of the individual key is:

```
<ep_name>:<event>:<action[,action...]>
```

In the above syntax:

`<ep_name>`: Name of the resource level entry point. For example monitor, offline, online, clean, and so on.

<event>: This should always be specified as **timeout** and is reserved for future use.

<action>: Specifies what information to capture on entry point timeout. Its value can be either **pstack** or **core** or both.

For example:

- `monitor:timeout:pstack` instructs the agent framework to generate pstack information on monitor entry point timeout.
- `offline:timeout:pstack` instructs the agent framework to generate pstack information on offline entry point timeout.
- `clean:timeout:pstack,core` instructs the agent framework to generate pstack information as well as core on clean entry point timeout.

Caution: The present scope of AdvDbg attribute is limited to resource level entry points. This includes all entry points except imf_init, imf_getnotification, and shutdown entry points. This could degrade the system performance based on how this attribute is configured and how many resources experience timeout.

Recommended steps for configuring AdvDbg attribute for monitor entry points

Set the AdvDbg attribute for the monitor entry points using the following steps:

1. Identify the resource for which the monitor entry point is timing out and analyze which information is useful.

   This helps in choosing the action (`pstack` or `core` or both)

2. Run the following commands:

```
# hares -override <res_name> AdvDbg
# hares -modify <res_name> AdvDbg -add monitor:timeout:pstack
```

3. After the data is captured, clear action by:

```
# hares -modify <res_name> AdvDbg -delete monitor:timeout:pstack
# hares -undo_override <res_name> AdvDbg
```

AEPTimeout

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

If any entry point needs to fetch the value of entry point timeout attribute from the running entry point (for example, MonitorTimeout and ActionTimeout for monitor entry point and action entry point respectively) then instead of executing `hatype/hares` command, the attribute AEPTimeout should be set to 1 and agent entry point will get the value of respective entry point timeout attribute in the AEPTimeout argument. The advantage of using AEPTimeout over `hatype/hares` command is, if the attribute is overridden at resource level, the AEPTimeout will automatically fetch the overridden value.

This feature does not apply to pre-V50 agents. The AEPTimeout attribute value cannot be overridden.

See “About the entry point timeouts” on page 48.

AgentClass

Indicates the scheduling class for agent process.

The default setting is TS. The AgentClass attribute value cannot be overridden.

AgentDirectory

Complete path of the directory in which the agent binary and scripts are located. Agents look for binaries and scripts in the following directories:

- The directory specified by the AgentDirectory attribute
- /opt/VRTSvcs/bin/type/
- /opt/VRTSagents/ha/bin/type/

If none of the above directories exist, the agent does not start. Use this attribute in conjunction with the AgentFile attribute to specify a different location or a different binary for the agent.

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. The AgentFailedOn attribute value cannot be overridden.

AgentFile

The complete name and path of the binary for an agent. If you do not specify a value for this attribute, VCS uses the agent binary at the path defined by the AgentDirectory attribute.

AgentPriority

Indicates the priority in which the agent process runs.

Default is 0. The AgentPriority attribute value cannot be overridden.

AgentReplyTimeout

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

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

The AgentReplyTimeout attribute value cannot be overridden.

AgentStartTimeout

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

Default is 60 seconds. The AgentStartTimeout attribute value cannot be overridden.

AlertOnMonitorTimeouts

Indicates the number of consecutive monitor failures after which VCS sends an SNMP notification to the user. 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 or a multiple of the value specified by the AlertOnMonitorTimeouts attribute, then VCS sends an SNMP notification to the user. If this attribute is set to a value, say N, then after sending the notification at the first monitor timeout, VCS also sends an SNMP notification at each N-consecutive monitor timeout including the first monitor timeout for the second-time notification.

In case of monitor timeouts, the AlertOnMonitorTimeouts attribute can be used in conjunction with the FaultOnMonitorTimeouts attribute to control the behavior of resources of a group configured under VCS. When FaultOnMonitorTimeouts is set to 0 and AlertOnMonitorTimeouts is set to some value for all resources of a service group, then VCS will not perform any action on monitor timeouts for resources configured under that service group, but will only send notifications at the frequency set in the AlertOnMonitorTimeouts attribute.

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

When AlertOnMonitorTimeouts is set to 0, VCS sends an SNMP notification to the user only for the first monitor timeout; VCS does not send further notifications to the user for subsequent monitor timeouts until the monitor returns a success.

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

ArgList

An ordered list of attributes whose values are passed to the open, close, online, offline, monitor, info, action, clean, imf_register, migrate, and meter entry points.

The default is an empty list. The ArgList attribute value cannot be overridden.

ArgList reference attributes

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

For example, say, there is a type T1 whose ArgList is of the form:

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

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

Note that one has to first add the attribute Attr3 to type T1 before adding Attr3:Attr_A to T1's ArgList. Only then should one modify Attr3 for a resource (res1T1) to reference another resource (res1T2). Also, the value of Attr3 can either be another resource of the same time (res2T1) or a resource of a different type (res1T2).

AttrChangedTimeout

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

AvailableMeters

List of meters that are supported by that agent of that resource type to measure. The AvailableMeters attribute value cannot be overridden.

The value of AvailableMeters attribute cannot be changed when VCS is running.

Default:`static str AvailableMeters{} = { SCPU="", SMem="" }`

User can set the value of this attribute in the types.cf file.

CleanRetryLimit

Defines the number of times clean operation can be retried before it succeeds. This parameter is meaningful only if the clean operation is implemented and the ManageFaults attribute is set to ALL. If CleanRetryLimit is set to 0, there will be no limit. Default = 0.

If this attribute is set to a non-zero value and the clean fails after the specified times then the resource enters into the ADMIN_WAIT state. The CleanRetryLimit attribute value can be overridden.

CleanTimeout

Maximum time (in seconds) within which the `clean` entry point must complete or else be terminated.

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

CloseTimeout

Maximum time (in seconds) within which the `close` entry point must complete or else be terminated.

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

ContainerOpts

This attribute helps you to control execution of agent entry point and allows you to control the container information passed to the agent entry point. For each application or resource type that you want to include as part of the zone, wpar, and project, you need to assign the following values to the ContainerOpts attribute:

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

Note: Container support is available only from V50 or later agent versions.

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 agent from restarting the resource indefinitely. The agent process attempts to restart the resource on the same system according to the number set in RestartLimit within ConfInterval before giving up and failing over. However, if the resource remains online for the interval designated in ConfInterval, earlier attempts to restart are not counted against RestartLimit. Default is 600 seconds.

The `ConfInterval` attribute value can be overridden.

EPClass

Indicates the scheduling class at which the entry points need to run. All entry points except Online are affected by this attribute.

Default = -1, which indicates that this attribute is not being used. Setting this attribute to a non-default value overrules the older AgentScript-Class attributes. The EPClass attribute value cannot be overridden.

EPPriority

Indicates the scheduling priority at which the entry points need to run. All entry points except Online are affected by this attribute.

Default = -1, which indicates that this attribute is not being used. Setting this attribute to a non-default value overrules the older AgentPriority attributes. The EPPriority attribute value cannot be overridden.

ExternalStateChange

Specifies what actions must be taken on the group when a resource of this type is detected online or offline outside of VCS control. If OnlineGroup is specified and the resource is detected as online, the group will be brought online. If OfflineGroup is specified and the resource is detected as intentionally offline, the group will be taken offline after due consideration of the group dependency. Default is no action.

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.

FaultPropagation

Specifies if VCS should propagate the fault up to parent resources and take the entire service group offline when a resource faults, or if VCS should not take the group offline, but fail over the group only when the system faults.

This attribute value can be set to 0 or 1. Default = 0.

If FaultPropagation is set to 1, then if a resource in the service group faults, the group is failed over if the group's AutoFailover attribute is set to 1. If FaultPropagation is set to 0, then if a resource in the service group faults, no other resources are taken offline nor the parent group regardless of the value set for the attribute Critical of a resource. VCS gives priority to the same attribute at a group level.

The FaultPropagation 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 can be overridden.

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

IMF

Determines whether the IMF-aware agent must perform intelligent resource monitoring. It is an association attribute with three keys—Mode, MonitorFreq, and RegisterRetryLimit.

- Mode defines for which state of the entry point, IMF monitoring must be performed. Mode can take values 0, 1, 2, or 3.

Mode value	Description
0	No IMF monitoring
1	Offline IMF monitoring
2	Online IMF monitoring
3	Offline and online IMF monitoring

- MonitorFreq specifies the frequency at which the agent invokes the monitor agent function.
- RegisterRetryLimit defines the maximum number of times the agent attempts to register a resource.

The IMF attribute value can be overridden.

Note: To make a custom agent IMF-aware, you must add IMF attribute in your configuration.

See “Adding IMF and IMFRegList attributes in configuration” on page 165.

IMFRegList

It is an ordered list of attributes. If IMFRegList attribute or any attribute defined in IMFRegList is changed then the registered resource gets unregistered from IMF.

If IMFRegList is not defined and if ArgList attribute or any attribute defined in ArgList gets changed, then the resource gets unregistered from IMF.

Note: If IMF support for custom agent is added by a user then it is recommended to define value of IMFRegList attribute, if the number of attributes used for resource registration with IMF is less then ArgList. Thus avoiding unregisteration of resources from IMF when any of the attributes gets changed that are not present in IMFResList.

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 info entry point can also be invoked by the user from the command line using the `hares -refreshinfo` command.

For example,

```
hares -refreshinfo <res> [-sys <system>] [-clus <cluster> | -localclus]
```

The InfoInterval attribute value can be overridden.

See "About the info entry point" on page 31.

InfoTimeout

Maximum time (in seconds) within which the info entry point must complete or be terminated.

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

You can extend this value by using the VCSAgSetResEPTimeout (for C/C++ entry point) /VCSAG_SET_RES_EP_TIMEOUT (for script entry point).

Note: You can extend this value by using the VCSAgSetResEPTimeout (for C/C++ entry point) /VCSAG_SET_RES_EP_TIMEOUT (for script entry point). Use this API cautiously as setting a value higher than MonitorInterval / 2 might result in delay of next periodic monitor which is used to check the state of the resource.

See “VCSAgSetResEPTimeout” on page 81.

See “VCSAG_SET_RES_EP_TIMEOUT” on page 98.

IntentionalOffline

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

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

Note that the intentional offline feature is available for agents registered as V51 or later. User can use any script agent registered as V51 or later

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

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

See “About on-off, on-only, and persistent resources ” on page 17.

LevelTwoMonitorFreq

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

The LevelTwoMonitorFreq attribute can be overridden at a resource level.

The monitor entry point can check if detail monitoring needs to be done through VCSAgGetMonitorLevel (for C/C++ based entry point) or VCSAG_GET_MONITOR_LEVEL (for script-based entry point).

See “VCSAG_GET_MONITOR_LEVEL” on page 97.

See “VCSAgGetMonitorLevel” on page 68.

LogDbg

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

Debug severities used by agent functions are in the range of DBG_1–DBG_21. By default, LogDbg is an empty list, meaning that no debug messages are logged for

a resource type. Users can modify this attribute for a given resource type, to specify the debug severities that they want to enable, which would cause those debug messages to be printed to the log files. For more information on agent debug levels, see the *Cluster Server Bundled Agents Reference* Guide.

The LogDbg attribute can be overridden at a resource level.

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

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

See "Using debug messages" on page 171.

More information is available about APIs that are available to log debug messages from agent entry points.

See "About logging agent messages" on page 117.

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

The LogDbg attribute is modified such that it can be overridden in VCS 6.2 or later releases. Although using this attribute, we can set DBG_AGINFO, DBG_AGTRACE, DBG_AGDEBUG at resource level but they will not have any impact as these levels are agent type specific. Hence we recommend to set values between DBG_1 to DBG_21 at resource level using this attribute.

LogFileSize

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

```
hatype -modify FileOnOff LogFileSize 2097152
```

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

```
hatype -display restype -attribute LogFileSize
```

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

LogViaHalog

Enables the agent's entry points logs to be logged in respective agent log file or engine log files based on the values configured.

- 0- The agent's log will be logged into their respective agent log file.
- 1- The C/C++ entry point's logs will go into the agent log file and the script entry point's logs will go into the engine log file using halog command.

Type: boolean-scalar

Default Value: 0

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 was ineffective. Resource state is ONLINE|ADMIN_WAIT.
2. The `offline` entry point did not complete within the expected time. Resource state is ONLINE|ADMIN_WAIT.
3. The `online` entry point did not complete within the expected time. Resource state is OFFLINE|ADMIN_WAIT.
4. The `online` entry point was ineffective. Resource state is OFFLINE|ADMIN_WAIT.
5. The resource was taken offline unexpectedly. Resource state is OFFLINE|ADMIN_WAIT.
6. For the online resource the monitor entry point consistently failed to complete within the expected time. Resource state is ONLINE| MONITOR_ TIMEDOUT|ADMIN_WAIT. The value configured in FaultOnMonitorTimeouts attribute indicates number of consecutive monitor failures after which the resource must move into ONLINE| MONITOR_ TIMEDOUT|ADMIN_WAIT state.

Meters

Defines the meters based on which fail-over decision will be taken for a service group that contains the resource of type that can perform metering. The keys of this attribute must be a subset of intersection of HostMeters (Cluster attribute) and AvailableMeters(Type level). This attribute cannot be overridden at resource level and cannot be modified at run time.

User can set the value of this attribute in the types.cf file.

Type and dimension: string-keylist

Default: `static keylist Meters = { SCPU, SMem }`

Example:{ SCPU }

See “AvailableMeters ” on page 183.

You can refer to the *Cluster Server Administrator's Guide* for information on HostMeters attribute

MeterControl

Indicates the intervals at which metering and forecasting are done for the keys specified in the Meters attribute. The attribute value cannot be overridden.

See “Meters” on page 192.

At every ForecastCycle, the ForecastFlag key of VCSInfo attribute will be set to 1 which is passed to the meter entry point to perform the forecasting.

This attribute includes the following keys:

- MeterInterval
  Frequency in seconds at which metering is done by the agent that supports metering. If the value is configured as 600, it indicates that agent calls meter entry point after every 600 seconds and sends the systems resource utilization data to the VCS engine.
- ForecastCycle
  The number of metering cycles after which forecasting of available capacity is done. If the value of MeterInterval is 60 seconds and ForcastCycle is 5 seconds then forecasting will be done after every 300 seconds.

You cannot modify the value at run time. User can set value in the type.cf file for the respective agent.

Type and dimension: integer-association

Default:`static int MeterControl{} = { MeterInterval=600, ForecastCycle=0 }`

MeterRegList

It is an ordered list of attributes. If MeterRegList attribute or any attribute that are defined in MeterRegList is changed then the meter entry point is called immediately.

This attribute cannot be overridden at resource level.

Type and dimension: string-vector

Default: `static str MeterRegList[] = { LDomName, CfgFile, NumCPU, Memory }`

MeterRetryLimit

Defines the number of times the meter operation can be retried before it succeeds. If MeterRetryLimit is set to 0, there is no limit on number of retries. If this attribute is set to a non-zero value and the meter entry point fails after the specified attempts, then the metering for that resource is disabled.

You can override this attribute at the resource level.

Default: 10

You can turn on metering in one of the following ways:

- Veritas recommends to increase the MeterRetryLimit attribute value to a higher value using the following command:
  If MeterRetryLimit is defined at type level, enter:

  ```
  # hatype -modify type_name MeterRetryLimit new_value
  ```

  If MeterRetryLimit is overridden at the resource level, enter:

  ```
  # hares -modify res_name MeterRetryLimit new_value
  ```

- Re-enable the resource using the following commands:

  ```
  # hares -modify res_name Enabled 0

  # hares -modify res_name Enabled 1
  ```

MeterTimeout

Maximum time for the meter entry point to complete.The value is in seconds.

Type and dimension: integer-scalar

Default: 300

Example: 900

MonitorInterval

Duration (in seconds) between two consecutive monitor calls for an ONLINE resource or a resource in transition.

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

MonitorStatsParam

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

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

- Frequency: Defines the number of monitor cycles after which the average monitor cycle time should be computed and sent to HAD. The value of this key can be from 1 to 30. A value of 0 (zero) indicates that the average monitor time need not be computed. This is the default value for this key.
- ExpectedValue: The expected monitor time in milliseconds for all resources of this type. Default=100.
- ValueThreshold: The acceptable percentage difference between the expected monitor cycle time (ExpectedValue) and the actual monitor cycle time. Default=100.
- AvgThreshold: The acceptable percentage difference between the benchmark average and the moving average of monitor cycle times. Default=40.

The MonitorStatsParam attribute values can be overridden.

For more information:

Refer to the *Administrator's Guide*.

MonitorTimeout

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

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

MigrateTimeout

Maximum time (in seconds) within which the migrate procedure must complete or else the procedure is terminated.

Default value is 600 secs. The MigrateTimeout attribute can be overridden.

MigrateWaitLimit

Number of monitor intervals to wait for resource to migrate after completing the migrate procedure. MigrateWaitLimit will be applicable for source as well as for target node; as the migrate operation brings the resource offline on the source node and online on the target node. We can also define MigrateWaitLimit as the number of monitor intervals required to wait for resource to go offline on source after completing the migrate procedure, and the number of monitor intervals to wait for resource to come online on target after the resource goes offline on source.

Default value is 2. The MigrateWaitLimit attribute can be overridden.

Probes fired manually are counted when MigrateWaitLimit is set and the resource is waiting to migrate. For example, if the MigrateWaitLimit of a resource is set to 5 and the MonitorInterval is set to 60 (seconds), the resource waits for a maximum of five monitor intervals (that is, 5 x 60), and if all five monitors within MigrateWaitLimit report the resource as online on source node, it sets the ADMIN_WAIT flag. If you run another probe, the resource waits for four monitor intervals (that is, 4 x 60), and if the fourth monitor does not report the state as offline on source , it sets the ADMIN_WAIT flag. This process is repated for remaining monitor intervals (3x60, 2x60 and 1x60). Similarly if the resource does not moved to online state within the MigrateWaitLimit then it sets the ADMIN_WAIT flag.

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. NumThreads does not control the number of threads used for other internal purposes.

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

If the entry points have a locking mechanism within them for synchronization, then set the NumThreads attribute to a relatively low value, for example, 2—5.

The maximum value that can be set for NumThreads is 100. If NumThreads need to be increased beyond 30, please contact Veritas support.

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 300 seconds. The OfflineMonitorInterval attribute value can be overridden.

OfflineTimeout

Maximum time (in seconds) within which the `offline` entry point must complete or else be terminated.

Default is 300 seconds. The OfflineTimeout attribute value can be overridden.

OfflineWaitLimit

Number of monitor intervals to wait after completing the offline procedure and before the resource goes offline.

Probes fired manually are counted when OfflineWaitLimit is set and the resource is waiting to go offline. For example, say the OfflineWaitLimit of a resource is set to 5 and the MonitorInterval is set to 60. The resource waits for a maximum of five monitor intervals (five times 60), and if all five monitors within OfflineWaitLimit report the resource as offline, it calls the clean agent function. If the user fires a probe, the resource waits for four monitor intervals (four times 60), and if the fourth monitor does not report the state as offline, it calls the clean agent function. If the user fires another probe, one more monitor cycle is consumed and the resource waits for three monitor intervals (three times 60), and if the third monitor does not report the state as offline, it calls the clean agent function.

Default = 0.

OnlineClass

Indicates the scheduling class at which the Online entry point needs to run. Only the Online entry point gets affected by this attribute.

Default = -1, which indicates that this attribute is not being used. Setting this attribute to a non-default value overrules the older AgentClass attributes. The OnlineClass attribute value cannot be overridden.

OnlinePriority

Indicates the scheduling priority at which the Online entry point needs to run. Only the Online entry point gets affected by this attribute.

Default = -1, which indicates that this attribute is not being used. Setting this attribute to a non-default value overrules the older AgentPriority attributes. The OnlinePriority attribute value cannot be overridden.

OnlineRetryLimit

Number of times to retry `online` if the attempt to bring a resource online is unsuccessful. This attribute is meaningful only if `clean` is implemented.

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

OnlineTimeout

Maximum time (in seconds) within which the `online` entry point must complete or else be terminated.

Default is 300 seconds. The OnlineTimeout attribute value can be overridden.

OnlineWaitLimit

Number of monitor intervals to wait after completing the online procedure, and before declaring the online attempt as ineffective.

This attribute is meaningful only if the `clean` entry point is implemented.

If `clean` is implemented, when the agent reaches the maximum number of monitor intervals it assumes that the online procedure was ineffective and runs `clean`. The agent then notifies HAD that the online attempt failed, or retries the procedure, depending on whether or not the OnlineRetryLimit is reached.

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

Each probe command fired from the user is considered as one monitor interval. For example, say the OnlineWaitLimit of a resource is set to 5. This means that the resource will be moved to a faulted state after five monitor intervals. If the user fires a probe, then the resource will be faulted after four monitor cycles, if the fourth monitor does not report the state as ONLINE. If the user again fires a probe, then

one more monitor cycle is consumed and the resource will be faulted if the third monitor does not report the state as ONLINE.

Default is 2. The OnlineWaitLimit attribute value can be overridden.

OpenTimeout

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

Operations

Indicates the valid operations for the resources of the type. The values are OnOff (can be brought online and taken offline), OnOnly (can be online only), and None (cannot be brought online or taken offline but can be monitored).

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 when VCS is not running. 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 when VCS is running; the commands are:

  ```
  haattr -add -static resource_type RegList -keylist
  ```

  - To set value

    ```
    # hatype -modify resource_type RegList attribute_name1
    attribute_name2
    ```

  - To add a new key

    ```
    # hatype -modify resource_type RegList -add attribute_name3
    ```

  For example:

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

The RegList attribute cannot be overridden.

RestartLimit

Affects how the agent responds to a resource fault.

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 agent process attempts to restart the resource according to the number set in RestartLimit before it gives up and attempts failover. However, if the resource remains online for the interval designated in ConfInterval, earlier attempts to restart are not counted against RestartLimit

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

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

See "ToleranceLimit" on page 201.

ScriptClass

Indicates the scheduling class of the script processes (for example, online) created by the agent. This attribute is not an overrideable static attribute.

The default setting is TS.

ScriptPriority

Indicates the priority of the script processes created by the agent. This attribute is not an overrideable static attribute.

Default is 0.

SourceFile

The file from which the configuration was read. This attribute is not an overrideable static attribute.

SupportedActions

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

See "About the action entry point" on page 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.

SupportedOperations

Indicates the operations that can be performed by the agent. The value can be set to "migrate", "meter" or both. The attribute value must be set only by those agents that support metering and migration.

This attribute is not an overrideable static attribute.

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.

Each probe command fired from the user is considered as one monitor. For example, when Tolerance limit of a resource is set to 5, the resource will be moved to a faulted state after five monitor intervals. If the user fires another probe and the resource is not reported as ONLINE, then the resource will be faulted after four monitor cycles. This process will be repeated for five monitoring cycles.

Chapter 10

State transition diagram

This chapter includes the following topics:

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
- Migrating a resource

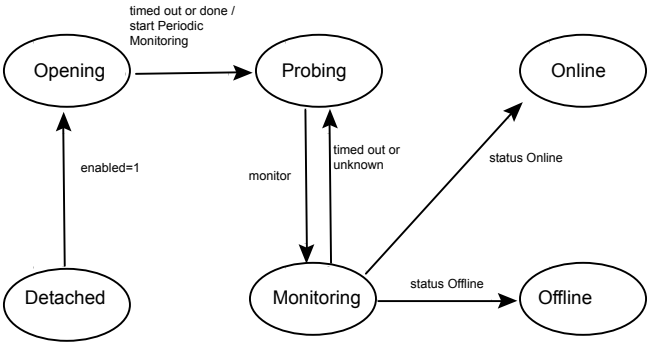
In addition, state transitions are shown for the handling of resources with respect to the `ManageFaults` service group attribute.

See "State transitions with respect to ManageFaults attribute" on page 235.

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.

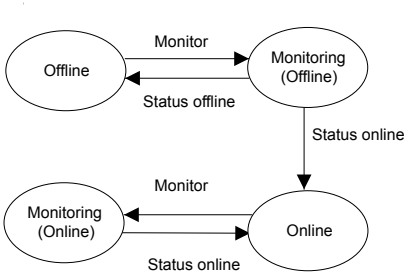
The agent writes resource state transition information into the agent log file when the `LogDbg` parameter, a static resource type attribute, is set to the value `DBG_AGINFO`. Agent developers can make use of this information when debugging agents.

Figure 10-1 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.

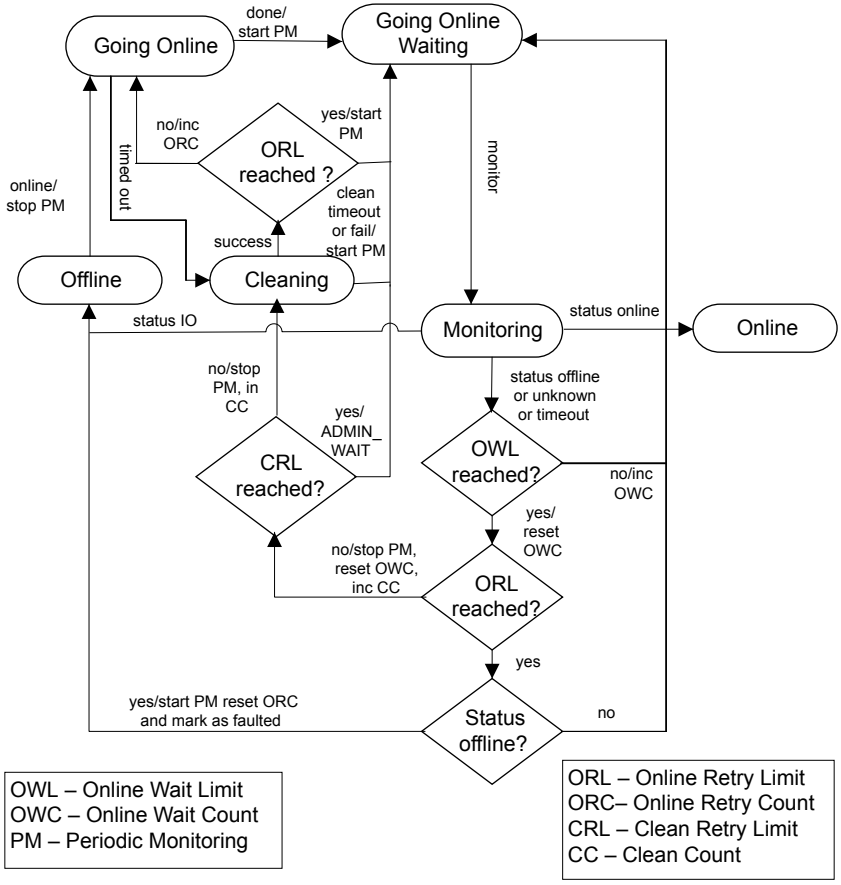
Figure 10-2 Resource in a steady state



When resources are in a steady state of Online or Offline, they are monitored at regular intervals. The intervals are specified by the `MonitorInterval` attribute in

the Online state and by the `OfflineMonitorInterval` attribute in the Offline state. An Online resource that is unexpectedly detected as Offline is considered to be faulted. Refer to diagrams describing faulted resources.

Figure 10-3 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 entry point completes, the resource enters the Going Online Waiting state where it waits for the next monitor cycle.

If online entry point timesout, the agent call *clean*.

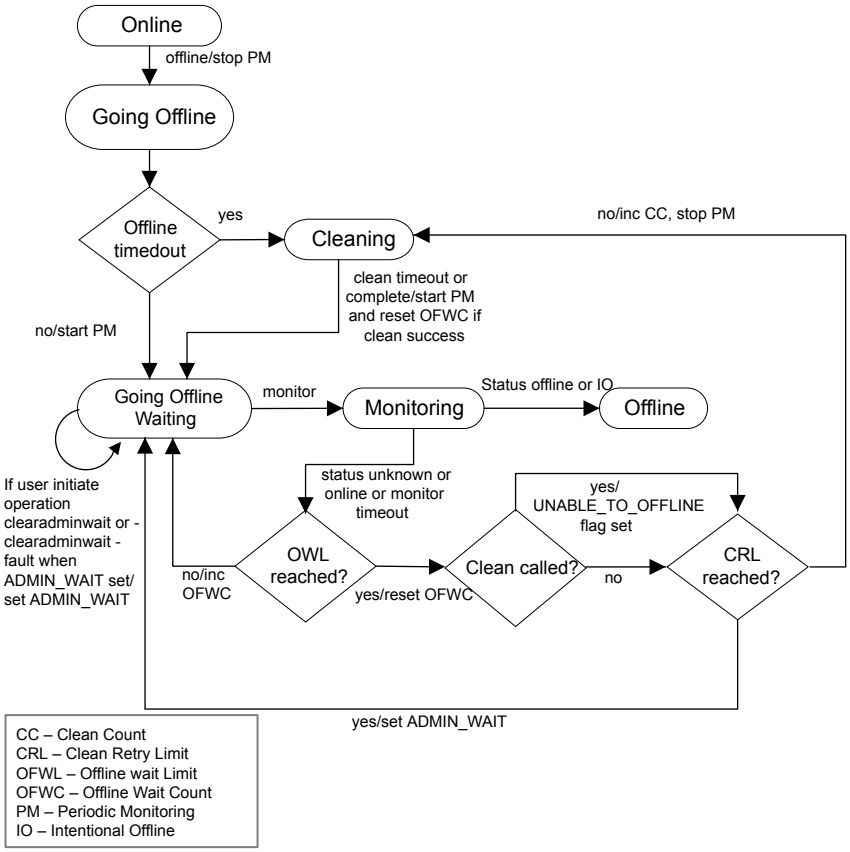
If monitor of GoingOnlineWaiting state returns a status as *online*, the resource moves to the Online state.

If monitor of GoingOnlineWaiting state returns a status as *Intentional Offline*, the resource moves to the Offline state.

If, however, the *monitor* times out, or returns a status of "not Online" (that is, unknown or offline), the following actions are considered:

- If OnlineWaitLimit is not reached then resource returns to GoingOnlineWaiting and waits for next monitor.
- If OnlineWaitLimit and OnlineRetryLimit are reached and the status remains unknow then resource returns to GoingOnlineWaiting and waits for next monitor.
- If OnlineWaitLimit and OnlineRetryLimit are reached then the status remains *offline* then resource return to Offline state and marks the resource as faulted.
- If OnlineWaitLimit is reached and OnlineRetryLimit is not reached then run *clean*, if CleanRetryLimit is not reached.
- If OnlineWaitLimit and CleanRetryLimit are reached and OnlineRetryLimit is not reached then move the resource to GoingOnlineWaiting and mark it as *ADMIN_WAIT*.
- If CleanRetryLimit is not reached and agent calls *clean* then following things can happen:
  - If clean times out or fails, the resource again returns to the Going Online Waiting state and waits for the next monitor cycle.
  - If clean succeeds with the OnlineRetryLimit reached, and the subsequent monitor reports the status as offline, the resource transitions to the offline state and it is marked as FAULTED.

Figure 10-4 Taking a resource offline and ManageFault = ALL



Upon receiving a request from the engine to take a resource *offline*, the agent places the resource in a GoingOffline state and invokes the offline entry point and stop periodic monitoring.

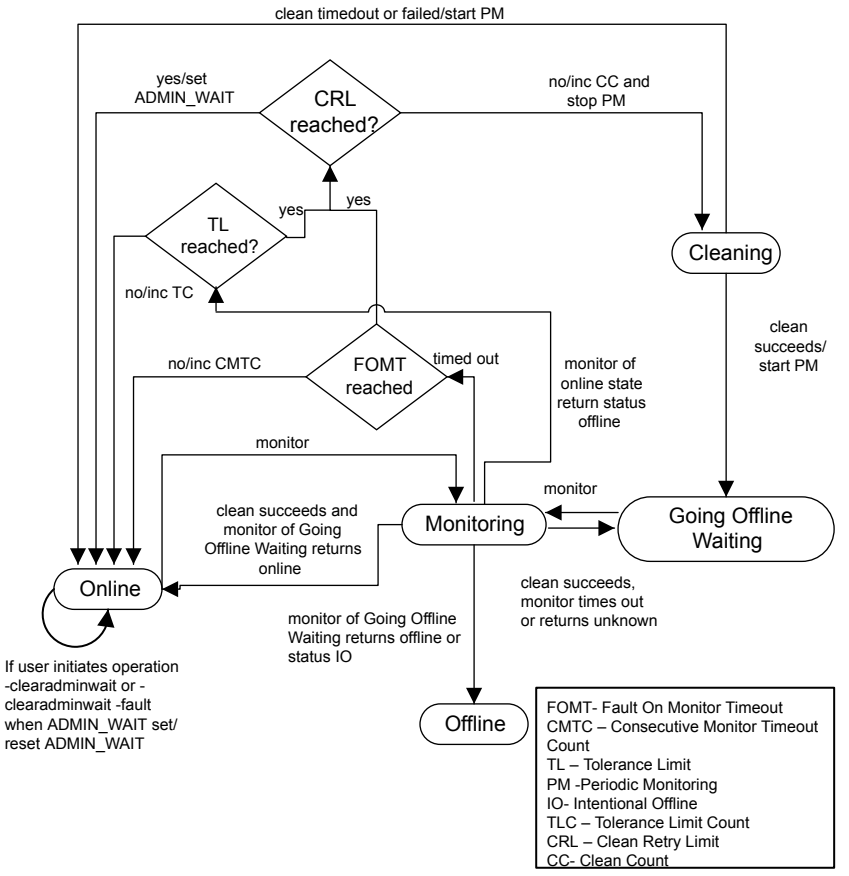
If *offline* completes, the resource enters the GoingOffline Waiting state, agent starts periodic monitoring of resource and also insert a monitor command for the resource. If *offline* times out, the clean entry point is called for the resource. If *clean* times out or complete then start periodic monitoring and reset Offline Wait Count if clean was success and move resource to Going Offline Waiting state

If monitor of Going Offline Waiting state returns offline or intentional offline then resource moves to *offline* state

If monitor of the GoingOffline Waiting state returns unknown or online, or if the monitor times out then,

- If OfflineWait Limit is not reached then the resource is moved to GoingOffline Waiting state.
- If Offline Wait Limit is reached then the resource which is cleaned earlier is called, then mark the resource as `UNABLE_TO_OFFLINE`
- If *CleantRetryLimit* is not reached then call clean.
- If *CleantRetryLimit* is reached then mark resource as `ADMIN_WAIT` state and move the resource to GoingOffline Waiting state.
- If the user initiates operation "*-clearadminwait*" then reset the `ADMIN_WAIT` flag. If user initiates operation "*-clearaminwait -fault*" then agent resets the `ADMIN_WAIT` flag

Figure 10-5 Resource fault when RestartLimit reached and ManageFault = ALL



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.

If *clean* retry limit is reached then set `ADMIN_WAIT` flag for resource and move resource to *online* state if not reached the agent invokes the clean entry point.

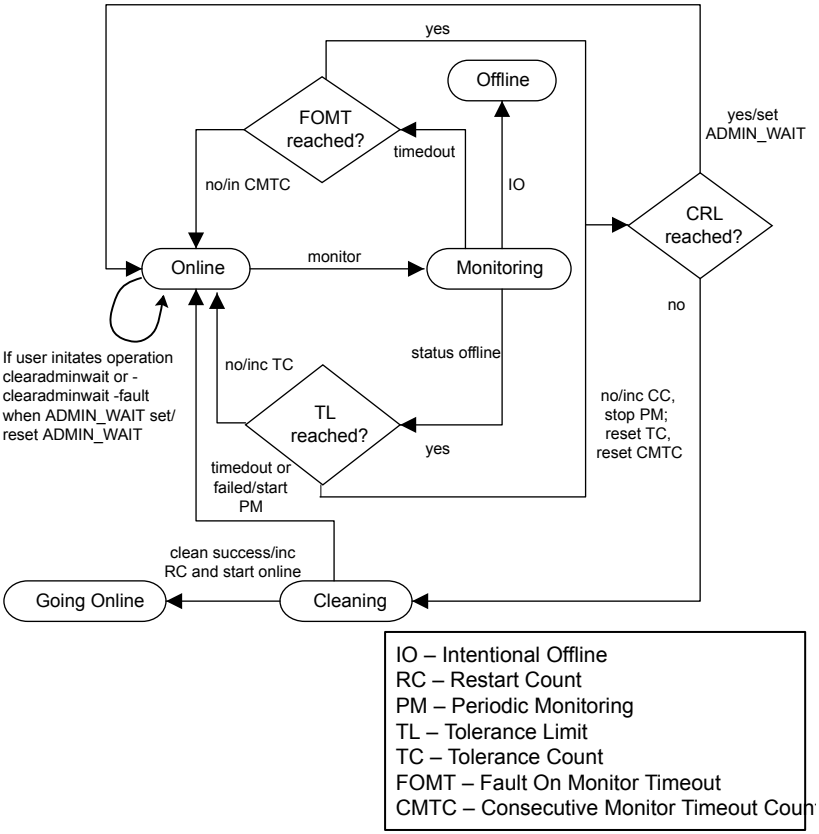
If *clean* fails, or if it times out, the agent places the resource in the *online* state as if no fault has occurred and starts periodic monitoring. If *clean* succeeds, the resource is placed in the Going Offline Waiting state and start periodic monitoring, where the agent waits for the next monitor.

If *clean* succeeds, the resource is placed in the GoingOffline 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 IO, the resource is placed in an *offline* state
- 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.

Note: If *clean* succeeds, the agent move resource to `GoingOfflineWait` and the resource is marked faulted. If monitoring of `GoingOfflineWaiting` returns *online* then the resource is moved to online state as engine does not expects the resource to go in offline state the as `GoingOfflineWaiting` state was set by the agent as a result of *clean* success.

Figure 10-6 Resource fault when RestartLimit not reached and ManageFault = ALL



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 then agent checks the clean counter to check if the clean entry point can be invoked.

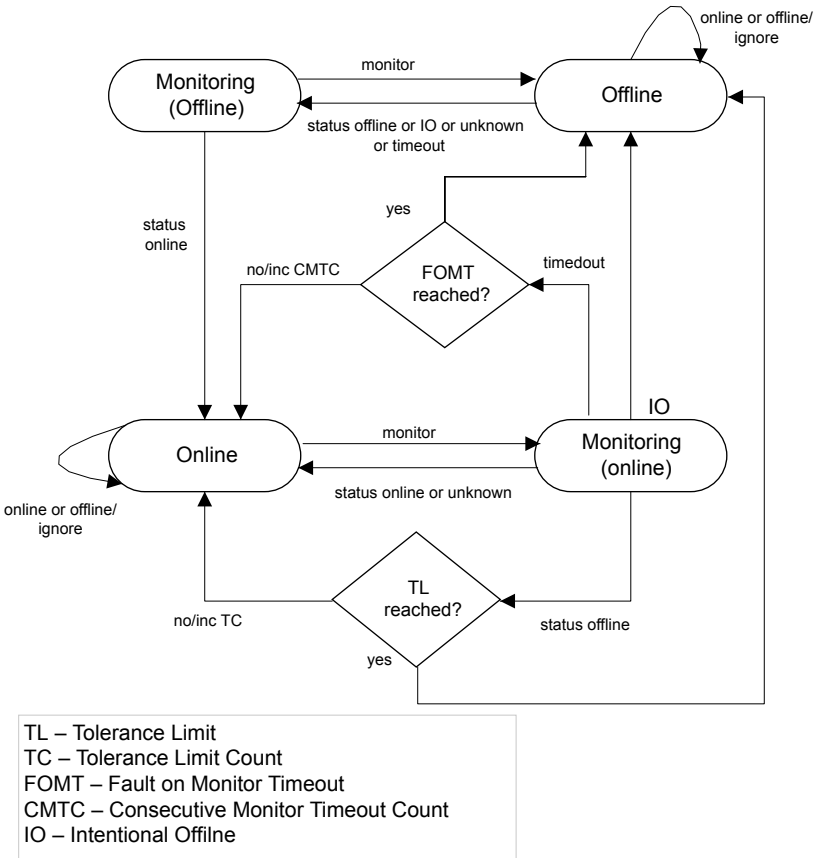
If CleanRetryLimit is reached then set `ADMIN_WAIT` flag for the resource and move the resource to online state. If clean retry limit fails to reach, the agent invokes the clean entry point.

- If *clean* succeeds, the resource is placed in the Going Online state and the *online* entry point is invoked to restart the resource; refer to the diagram, "Bringing a resource online."

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

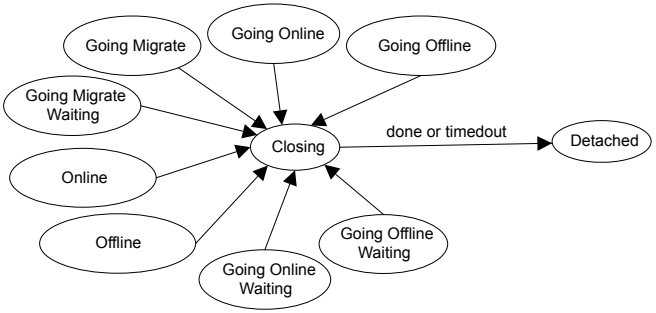
Refer to the diagram "Resource fault without automatic restart," for a discussion of activity when a resource faults and the `RestartLimit` is reached.

Figure 10-7 Monitoring of persistent resources



If *monitor* returns *offline* and the `ToleranceLimit` is reached, the resource is placed in an *Offline* state and noted as `FAULTED`. If monitor timeout and `FaultOnMonitorTimeouts` is reached, the resource is placed in an *Offline* state and noted as `FAULTED`.

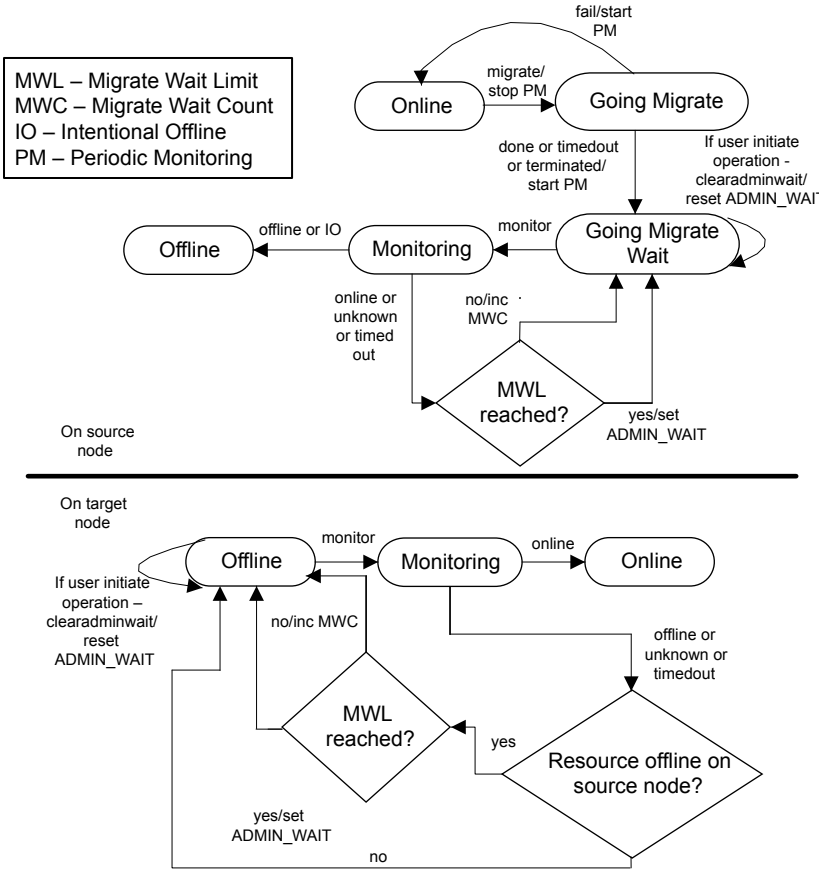
Figure 10-8 Closing a resource



The state diagram explains all the states from where a resource can move to *Closing* state. The following tables describes the actions performed in different state by which a resource can move to *Closing* state,

State	Action
Online to Closing	hastop –local –force or hares -delete or Enabled = 0 only if resource is persistent resource
Offline to Closing	Enabled = 0 or hastop –local or hastop –local –force or hares -delete
GoingOnlineWaiting	hastop –local –force or hares -delete
GoingOfflineWaiting	hastop –local –force or hares -delete
GoingMigrateWaiting	hastop –local –force or hares -delete
GoingOnline	hastop –local –force
GoingOffline	hastop –local –force
GoingMigrate	hastop –local –force
Probing	Enabled = 0 or hastop –local or hastop –local –force or hares –delete

Figure 10-9 Migrating a resource



The migration process is initiated from the source system, where virtual machine (VM) is online and the VM is migrated to the target system where it was offline. When the agent on the source system receives a migration request from the engine to migrate the resource, the resource goes to Going Migrate state, where migrate entry point is invoked. If the migrate entry point fails with return code 255, the resource is transitioned back to the online state and failure of migrate operation is communicated to the engine. This indicates that the migration operation cannot be performed.

Agent framework ignores any value returned between 101 to 254 range and will return to online state. If the migrate entry point completes successfully or times out is reached, the resource enters the Going Migrate Waiting state where it waits for the next monitor cycle and the monitor calls with the frequency as configured in

MonitorInterval. If monitor returns an offline status, the resource moves to the offline state and the migration on the source system is considered complete.

Even after moving to offline state the agent keeps on monitoring the resource with same monitor frequency as configured in MonitorInterval. This is to detect if VM fails back at source node early. However, if monitor entry point times out or reports the state as online or unknown, the resource waits for the MigrateWaitLimit resource cycle to complete.

If any of the monitor within MigrateWaitLimit reports the state as offline, the resource transitions to offline state and the same is reported to the engine. If the monitor entry point times out or reports the state as online or unknown even after MigrateWaitLimit has reached, the ADMIN_WAIT flag is set.

If resource migration operation is successful on source node then on target node the agent change the monitoring frequency from OfflineMonitorInterval to MonitorInternal to detect success full migration early. But if resource is not detected as online on target node even after MigrateWaitLimit is reached then resource is moved to ADMIN_WAIT state and agent fail back to monitor frequency as configured in OfflineMonitorInterval

---

Note: : The agent does not call clean if the migrate entry point times out or if monitor after migrate entry point times out or reports the state as online or unknown even after MigrateWaitLimit has reached. You need to manually clear the ADMIN_WAIT flag after resolving the issue.

Figure 10-10 Resource fault: ManageFaults attribute = ALL

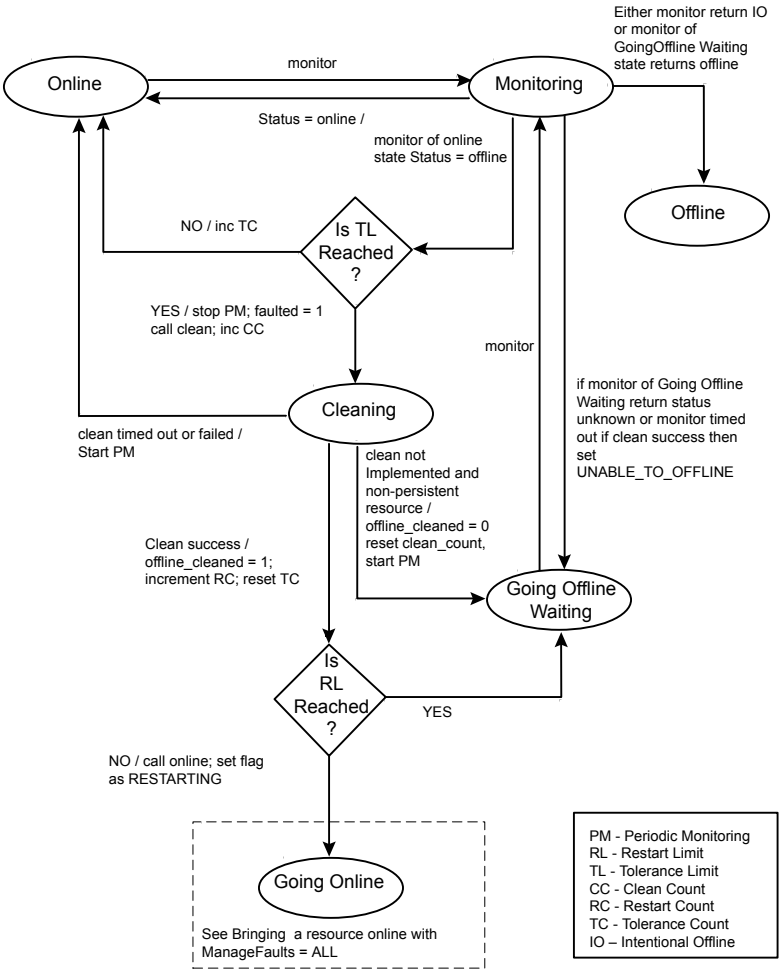
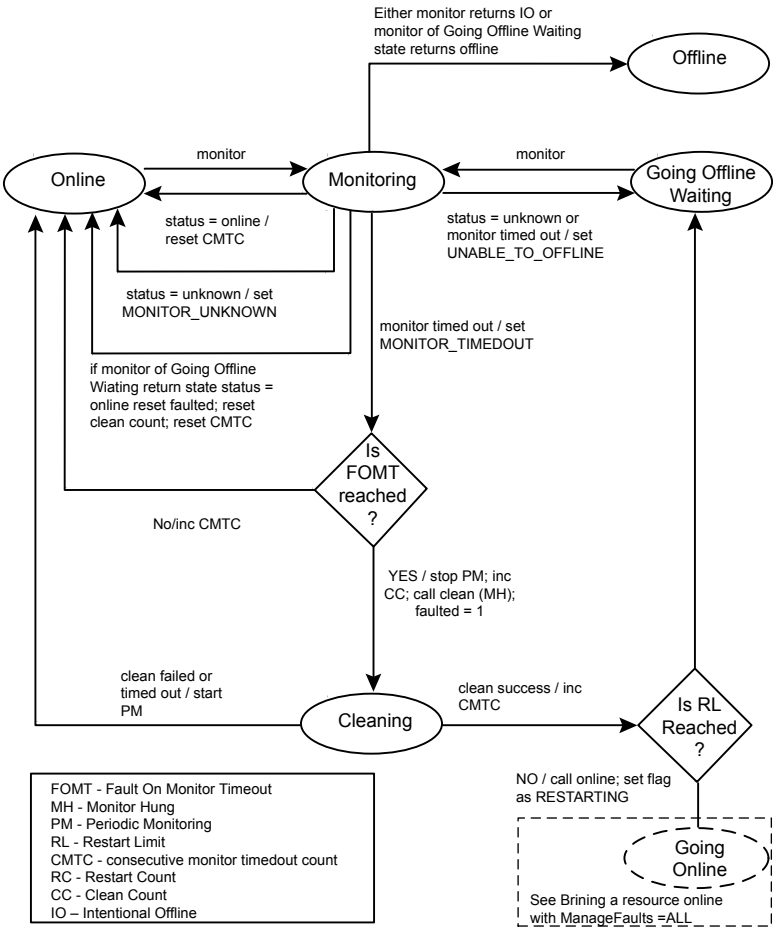


Figure 10-11 Resource fault (monitor hung): ManageFaults attribute = ALL



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 191.

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

Figure 10-12 Bringing a resource online: ManageFaults attribute = NONE

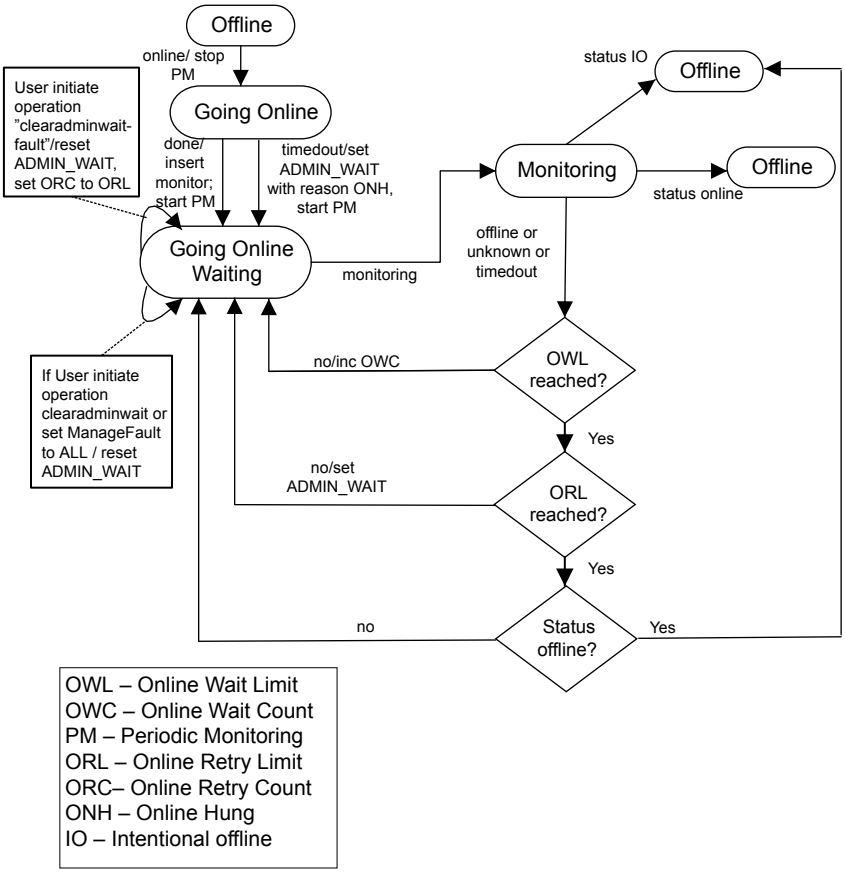


Figure 10-13 Taking a resource offline; ManageFaults = None

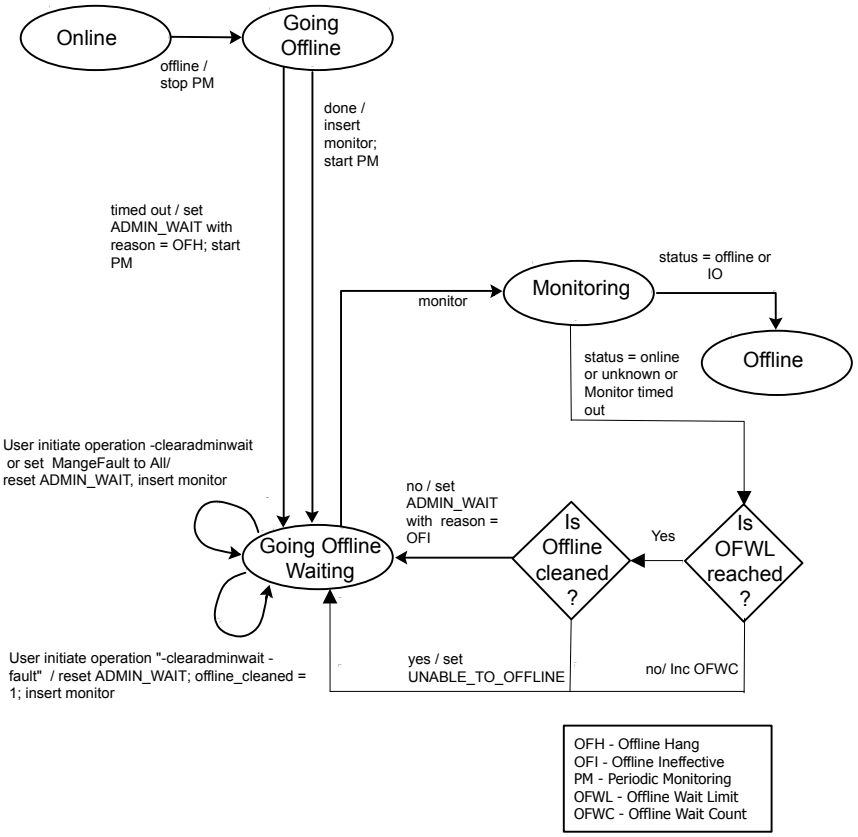


Figure 10-14 Resource fault (unexpected offline): ManageFaults attribute = NONE

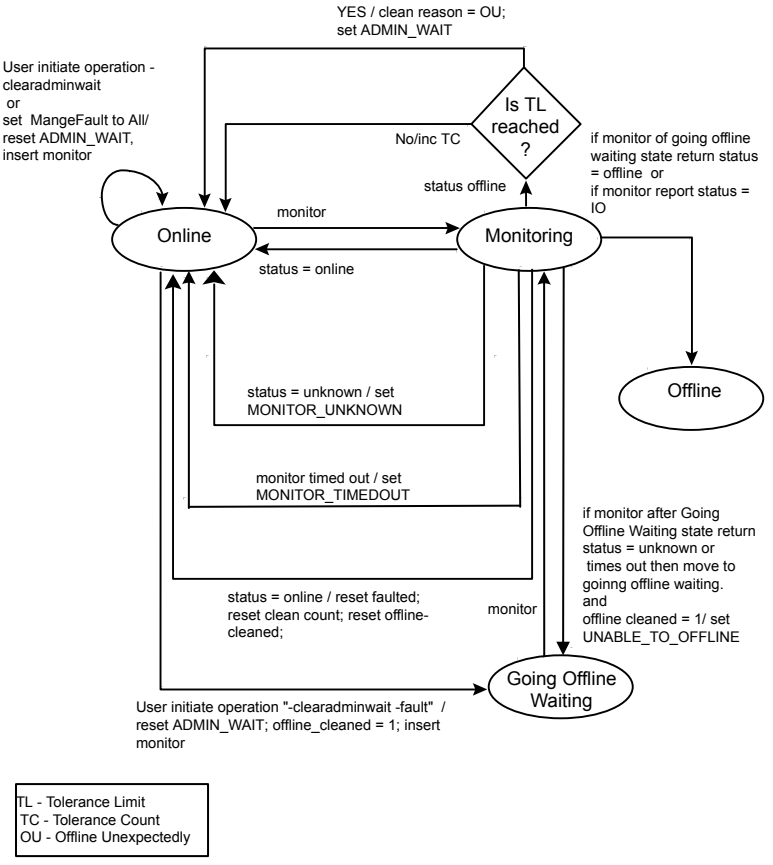
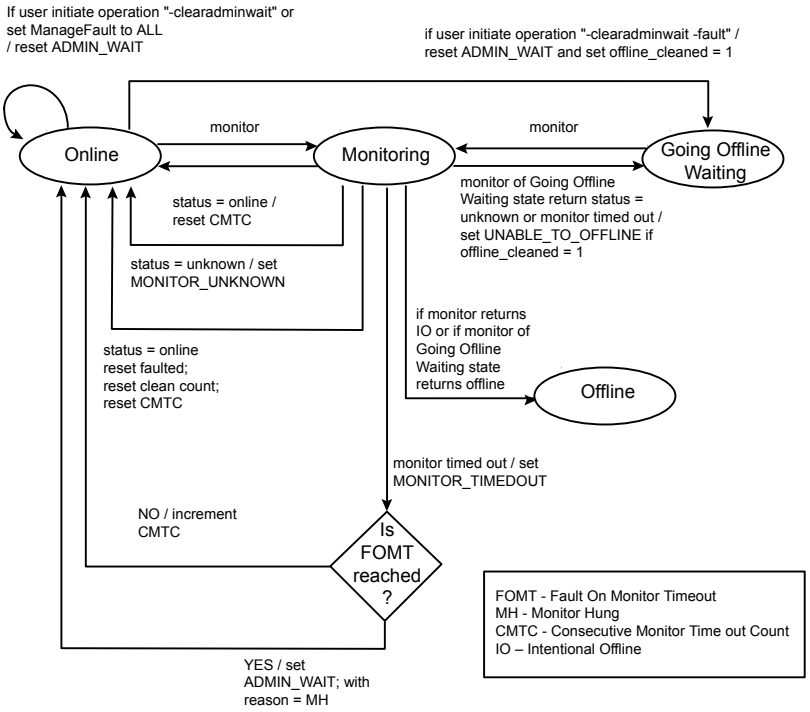


Figure 10-15 Resource fault (monitor hung): ManageFaults attribute = NONE



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

- Migrating a resource

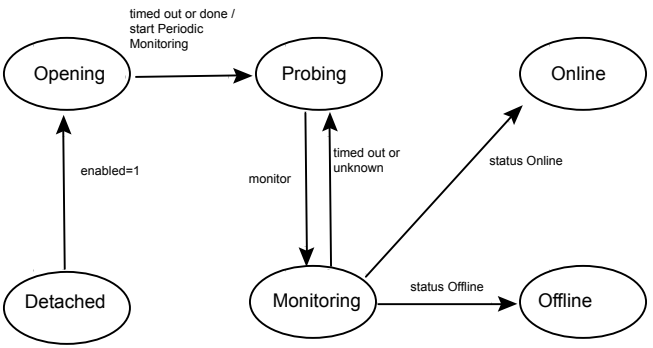
In addition, state transitions are shown for the handling of resources with respect to the `ManageFaults` service group attribute.

See "State transitions with respect to ManageFaults attribute" on page 235.

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.

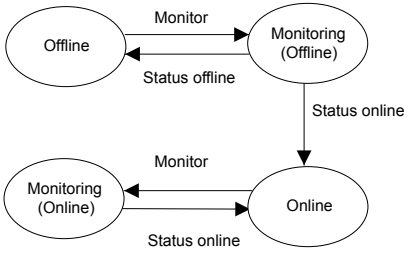
The agent writes resource state transition information into the agent log file when the `LogDbg` parameter, a static resource type attribute, is set to the value `DBG_AGINFO`. Agent developers can make use of this information when debugging agents.

Figure 10-16 Opening a resource



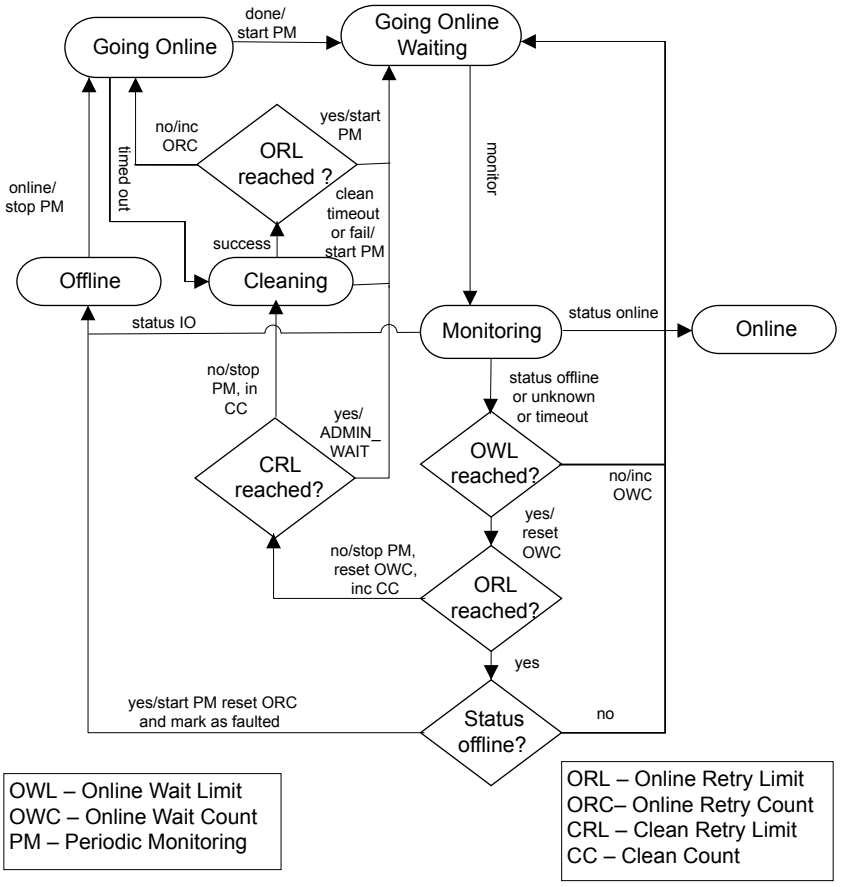
When the agent starts up, each resource starts with the initial state of Detached. In the Detached state (Enabled=0), the agent rejects all commands to bring a resource online or take it offline.

Figure 10-17 Resource in a 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` attribute in the Online state and by the `OfflineMonitorInterval` attribute in the Offline state. An Online resource that is unexpectedly detected as Offline is considered to be faulted. Refer to diagrams describing faulted resources.

Figure 10-18 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 entry point completes, the resource enters the Going Online Waiting state where it waits for the next monitor cycle.

If online entry point timesout, the agent call *clean*.

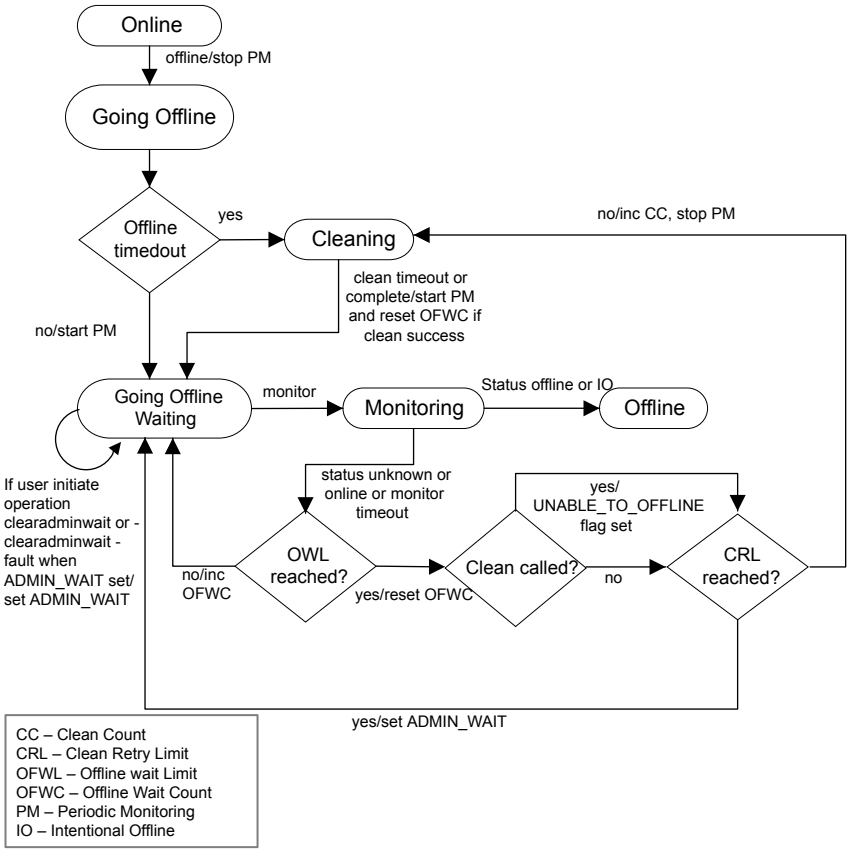
If monitor of GoingOnlineWaiting state returns a status as *online*, the resource moves to the Online state.

If monitor of GoingOnlineWaiting state returns a status as *Intentional Offline*, the resource moves to the Offline state.

If, however, the *monitor* times out, or returns a status of "not Online" (that is, unknown or offline), the following actions are considered:

- If OnlineWaitLimit is not reached then resource returns to GoingOnlineWaiting and waits for next monitor.
- If OnlineWaitLimit and OnlineRetryLimit are reached and the status remains unknow then resource returns to GoingOnlineWaiting and waits for next monitor.
- If OnlineWaitLimit and OnlineRetryLimit are reached then the status remains *offline* then resource return to Offline state and marks the resource as faulted.
- If OnlineWaitLimit is reached and OnlineRetryLimit is not reached then run *clean*, if CleanRetryLimit is not reached.
- If OnlineWaitLimit and CleanRetryLimit are reached and OnlineRetryLimit is not reached then move the resource to GoingOnlineWaiting and mark it as *ADMIN_WAIT*.
- If CleanRetryLimit is not reached and agent calls *clean* then following things can happen:
  - If clean times out or fails, the resource again returns to the Going Online Waiting state and waits for the next monitor cycle.
  - If clean succeeds with the OnlineRetryLimit reached, and the subsequent monitor reports the status as offline, the resource transitions to the offline state and it is marked as FAULTED.

Figure 10-19 Taking a resource offline and ManageFault = ALL



Upon receiving a request from the engine to take a resource *offline*, the agent places the resource in a GoingOffline state and invokes the offline entry point and stop periodic monitoring.

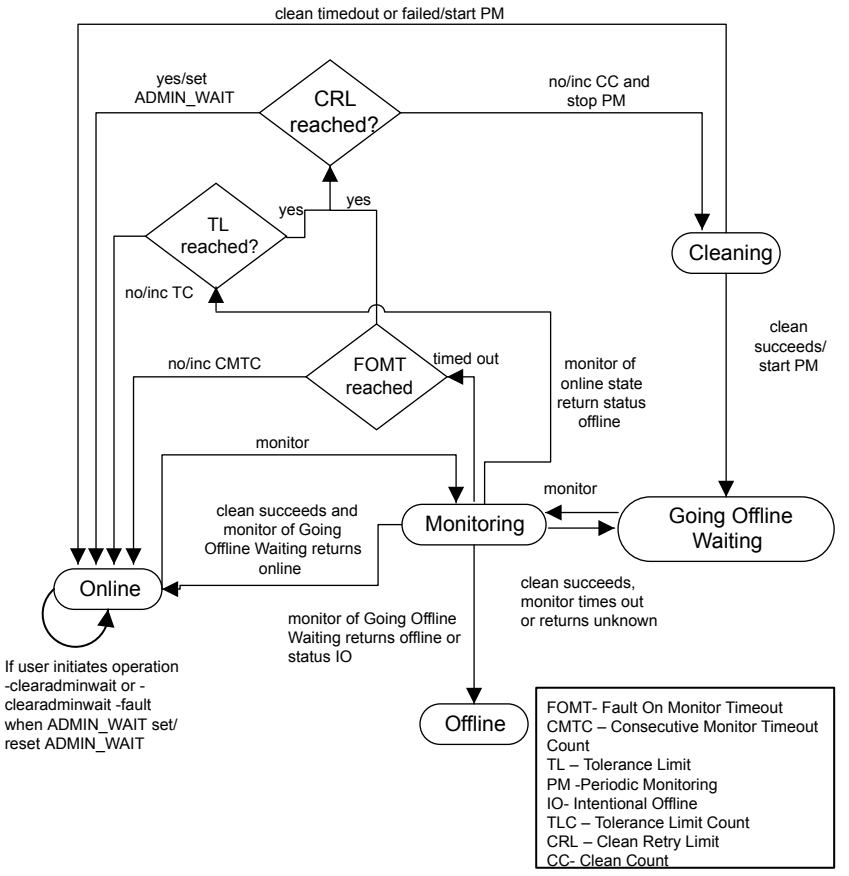
If *offline* completes, the resource enters the GoingOffline Waiting state, agent starts periodic monitoring of resource and also insert a monitor command for the resource. If *offline* times out, the clean entry point is called for the resource. If *clean* times out or complete then start periodic monitoring and reset Offline Wait Count if clean was success and move resource to Going Offline Waiting state

If monitor of Going Offline Waiting state returns offline or intentional offline then resource moves to *offline* state

If monitor of the GoingOffline Waiting state returns unknown or online, or if the monitor times out then,

- If OfflineWait Limit is not reached then the resource is moved to GoingOffline Waiting state.
- If Offline Wait Limit is reached then the resource which is cleaned earlier is called, then mark the resource as `UNABLE_TO_OFFLINE`
- If *CleantRetryLimit* is not reached then call clean.
- If *CleantRetryLimit* is reached then mark resource as `ADMIN_WAIT` state and move the resource to GoingOffline Waiting state.
- If the user initiates operation "*-clearadminwait*" then reset the `ADMIN_WAIT` flag. If user initiates operation "*-clearaminwait -fault*" then agent resets the `ADMIN_WAIT` flag

Figure 10-20 Resource fault when RestartLimit reached and ManageFault = ALL



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.

If *clean* retry limit is reached then set `ADMIN_WAIT` flag for resource and move resource to *online* state if not reached the agent invokes the clean entry point.

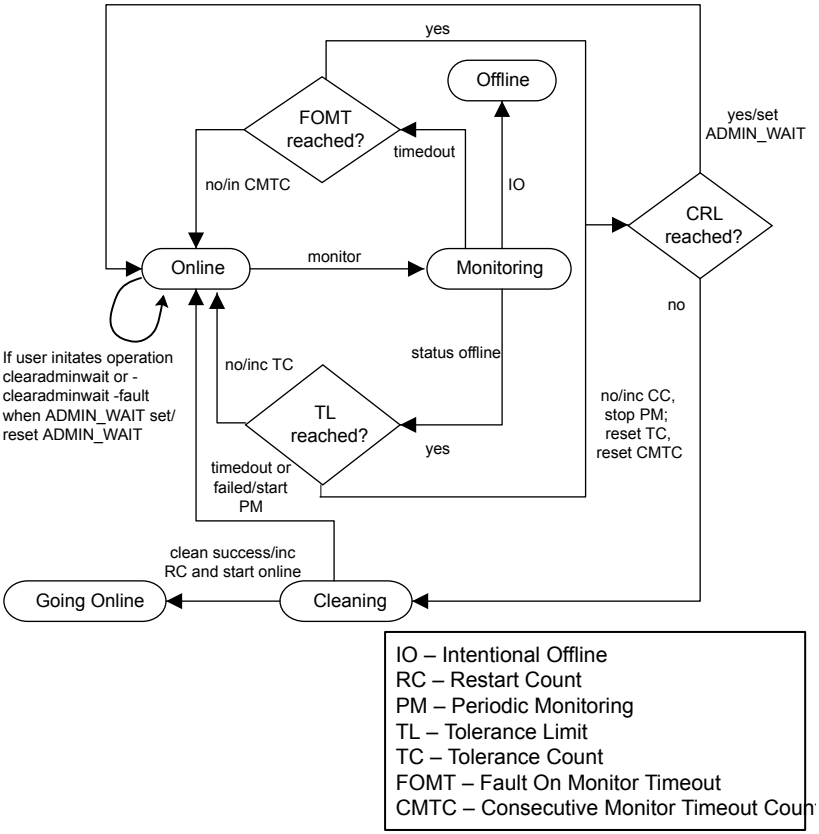
If *clean* fails, or if it times out, the agent places the resource in the *online* state as if no fault has occurred and starts periodic monitoring. If *clean* succeeds, the resource is placed in the Going Offline Waiting state and start periodic monitoring, where the agent waits for the next monitor.

If *clean* succeeds, the resource is placed in the GoingOffline 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 IO, the resource is placed in an *offline* state
- 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.

Note: If *clean* succeeds, the agent move resource to `GoingOfflineWait` and the resource is marked faulted. If monitoring of `GoingOfflineWaiting` returns *online* then the resource is moved to online state as engine does not expects the resource to go in offline state the as `GoingOfflineWaiting` state was set by the agent as a result of *clean* success.

Figure 10-21 Resource fault when RestartLimit not reached and ManageFault = ALL



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 then agent checks the clean counter to check if the clean entry point can be invoked.

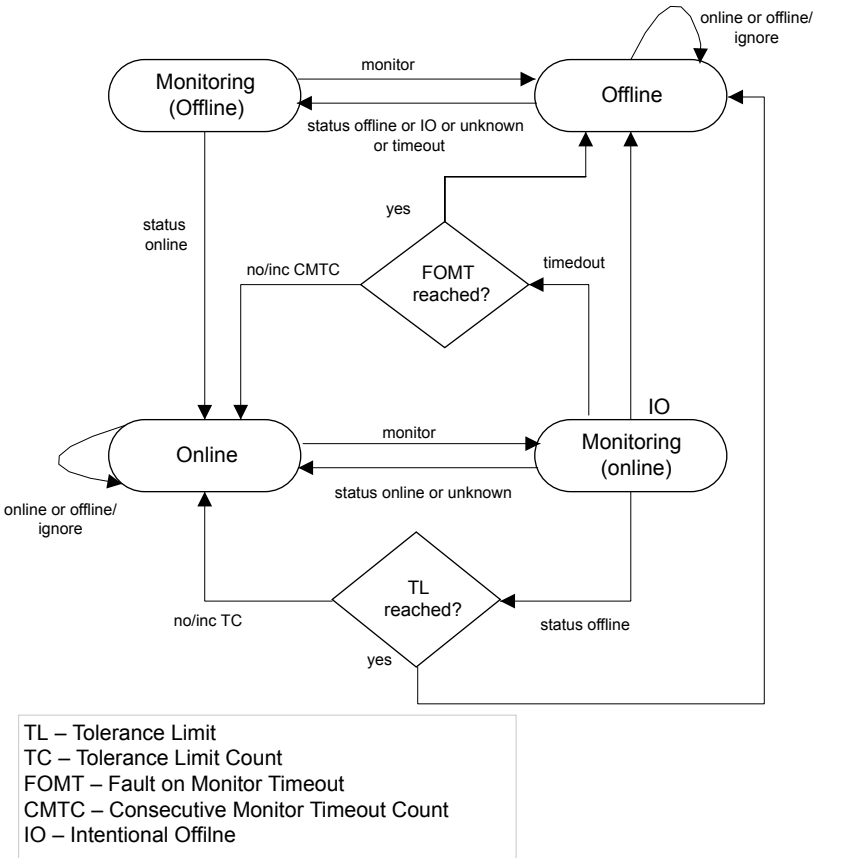
If CleanRetryLimit is reached then set `ADMIN_WAIT` flag for the resource and move the resource to online state. If clean retry limit fails to reach, the agent invokes the clean entry point.

- If *clean* succeeds, the resource is placed in the Going Online state and the *online* entry point is invoked to restart the resource; refer to the diagram, "Bringing a resource online."

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

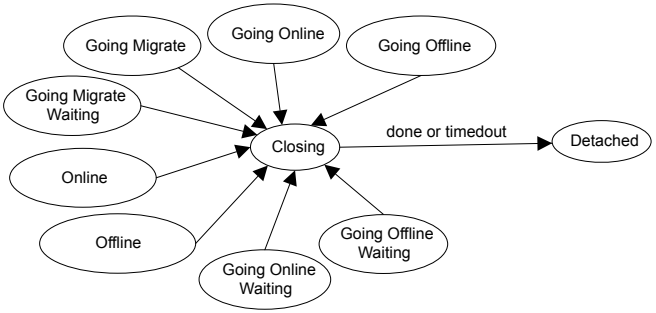
Refer to the diagram "Resource fault without automatic restart," for a discussion of activity when a resource faults and the `RestartLimit` is reached.

Figure 10-22 Monitoring of persistent resources



If *monitor* returns *offline* and the `ToleranceLimit` is reached, the resource is placed in an *Offline* state and noted as `FAULTED`. If monitor timeout and `FaultOnMonitorTimeouts` is reached, the resource is placed in an *Offline* state and noted as `FAULTED`.

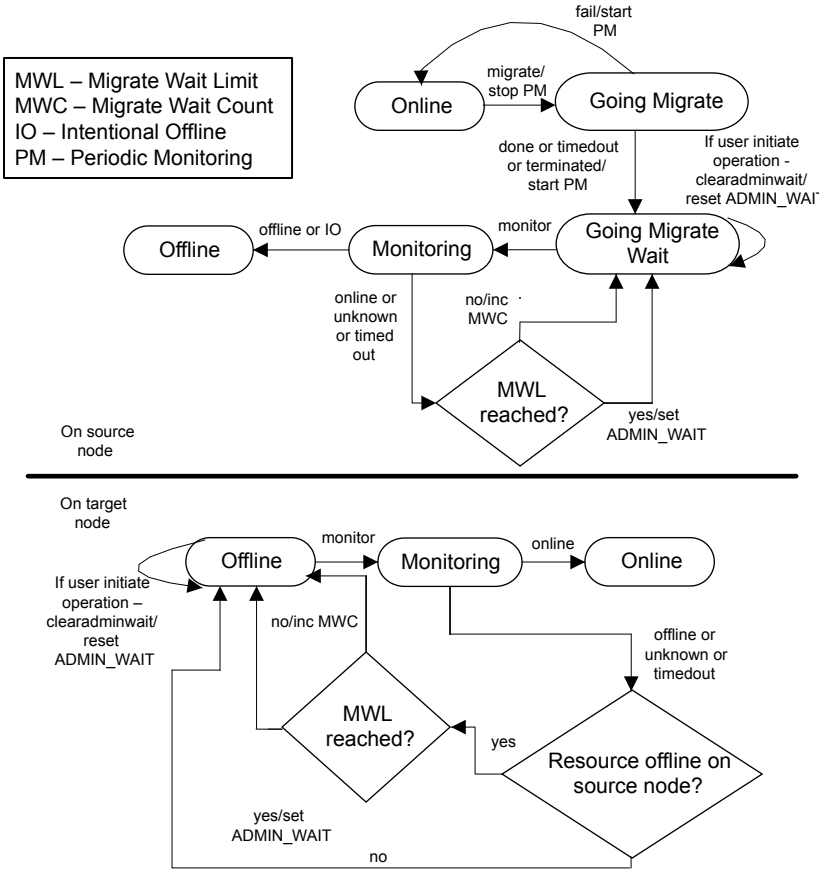
Figure 10-23 Closing a resource



The state diagram explains all the states from where a resource can move to *Closing* state. The following tables describes the actions performed in different state by which a resource can move to *Closing* state,

State	Action
Online to Closing	hastop –local –force or hares -delete or Enabled = 0 only if resource is persistent resource
Offline to Closing	Enabled = 0 or hastop –local or hastop –local –force or hares -delete
GoingOnlineWaiting	hastop –local –force or hares -delete
GoingOfflineWaiting	hastop –local –force or hares -delete
GoingMigrateWaiting	hastop –local –force or hares -delete
GoingOnline	hastop –local –force
GoingOffline	hastop –local –force
GoingMigrate	hastop –local –force
Probing	Enabled = 0 or hastop –local or hastop –local –force or hares –delete

Figure 10-24 Migrating a resource



The migration process is initiated from the source system, where virtual machine (VM) is online and the VM is migrated to the target system where it was offline. When the agent on the source system receives a migration request from the engine to migrate the resource, the resource goes to Going Migrate state, where migrate entry point is invoked. If the migrate entry point fails with return code 255, the resource is transitioned back to the online state and failure of migrate operation is communicated to the engine. This indicates that the migration operation cannot be performed.

Agent framework ignores any value returned between 101 to 254 range and will return to online state. If the migrate entry point completes successfully or times out is reached, the resource enters the Going Migrate Waiting state where it waits for the next monitor cycle and the monitor calls with the frequency as configured in

MonitorInterval. If monitor returns an offline status, the resource moves to the offline state and the migration on the source system is considered complete.

Even after moving to offline state the agent keeps on monitoring the resource with same monitor frequency as configured in MonitorInterval. This is to detect if VM fails back at source node early. However, if monitor entry point times out or reports the state as online or unknown, the resource waits for the MigrateWaitLimit resource cycle to complete.

If any of the monitor within MigrateWaitLimit reports the state as offline, the resource transitions to offline state and the same is reported to the engine. If the monitor entry point times out or reports the state as online or unknown even after MigrateWaitLimit has reached, the ADMIN_WAIT flag is set.

If resource migration operation is successful on source node then on target node the agent change the monitoring frequency from OfflineMonitorInterval to MonitorInternal to detect success full migration early. But if resource is not detected as online on target node even after MigrateWaitLimit is reached then resource is moved to ADMIN_WAIT state and agent fail back to monitor frequency as configured in OfflineMonitorInterval

---

Note: : The agent does not call clean if the migrate entry point times out or if monitor after migrate entry point times out or reports the state as online or unknown even after MigrateWaitLimit has reached. You need to manually clear the ADMIN_WAIT flag after resolving the issue.

Figure 10-25 Resource fault: ManageFaults attribute = ALL

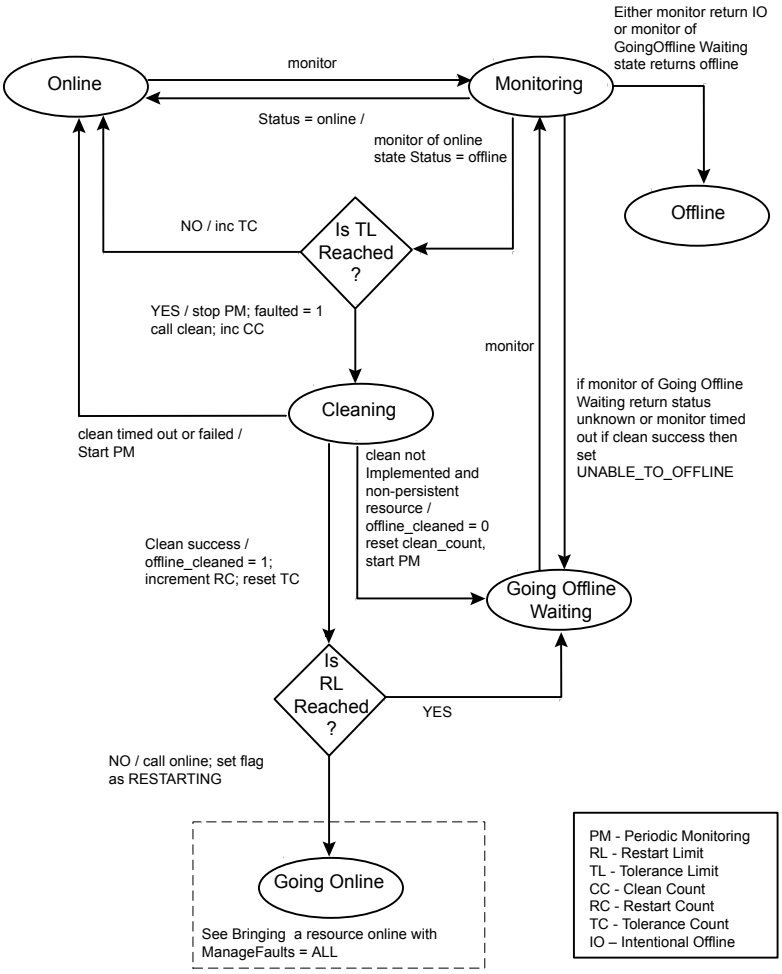
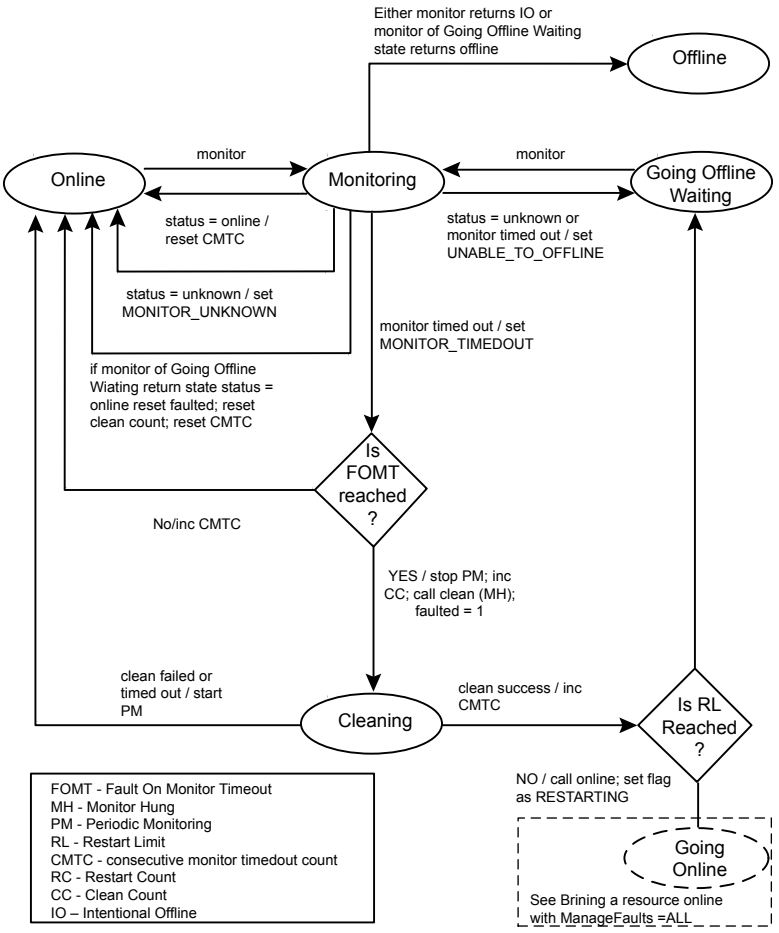


Figure 10-26 Resource fault (monitor hung): ManageFaults attribute = ALL



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 191.

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

Figure 10-27 Bringing a resource online: ManageFaults attribute = NONE

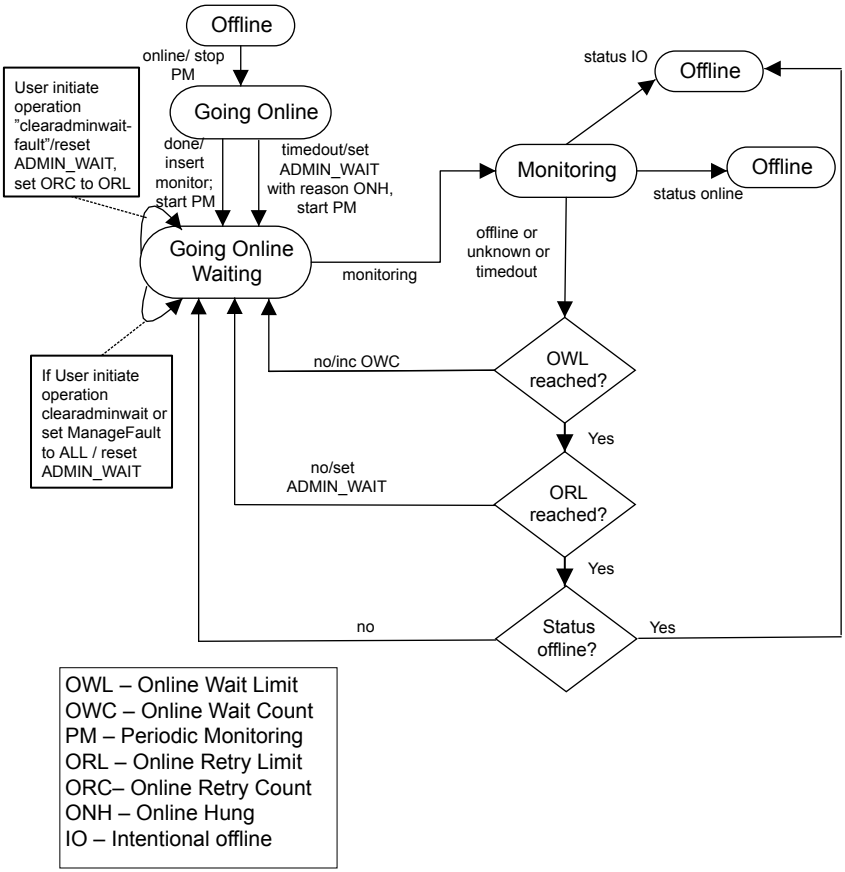


Figure 10-28 Taking a resource offline; ManageFaults = None

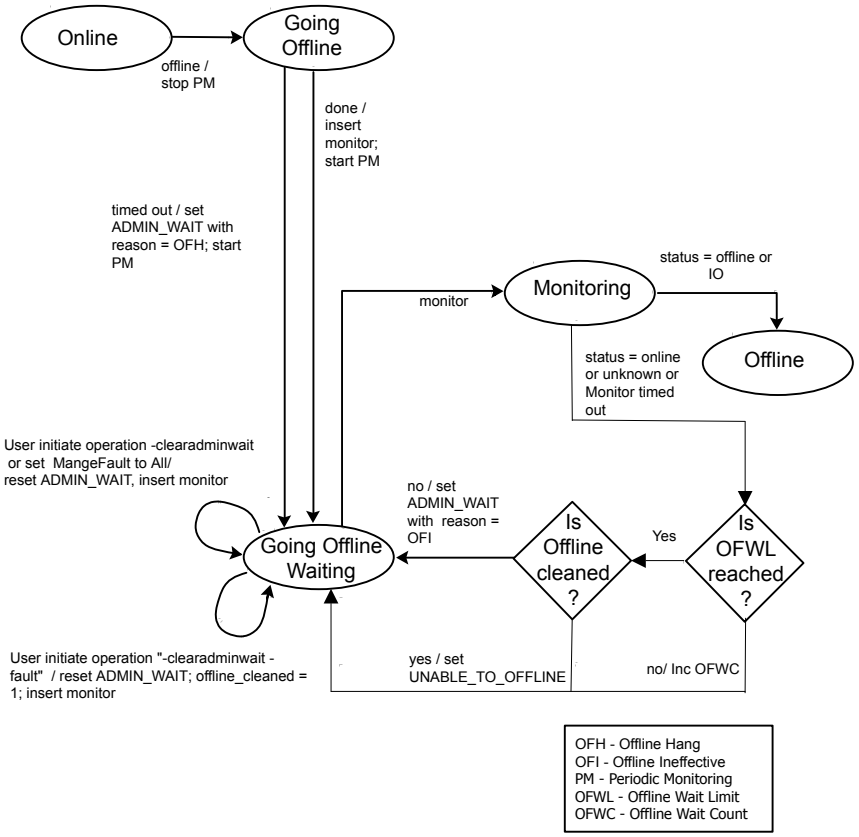


Figure 10-29 Resource fault (unexpected offline): ManageFaults attribute = NONE

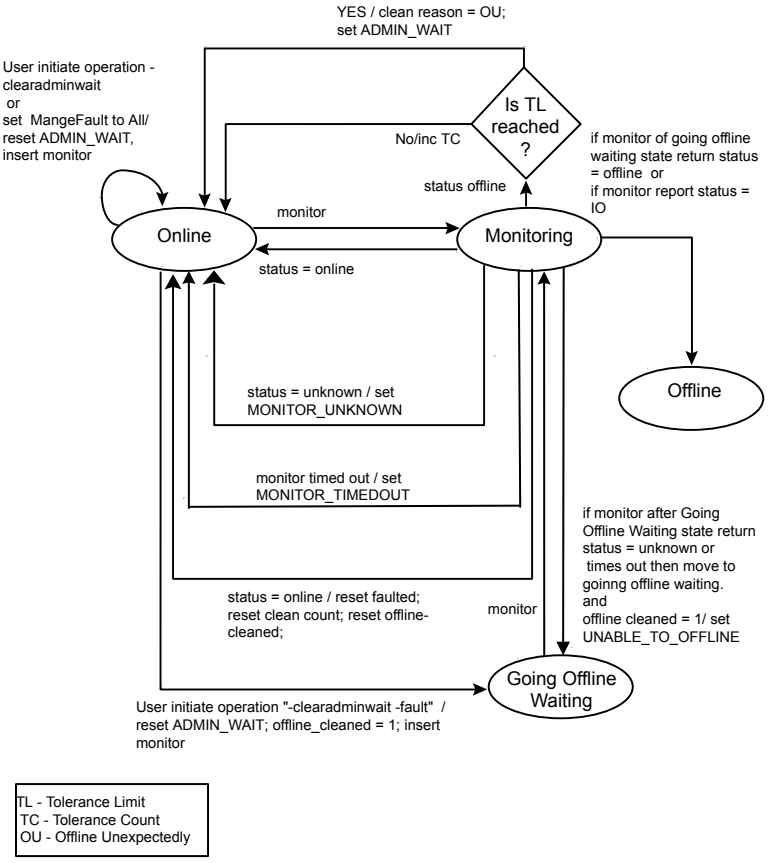
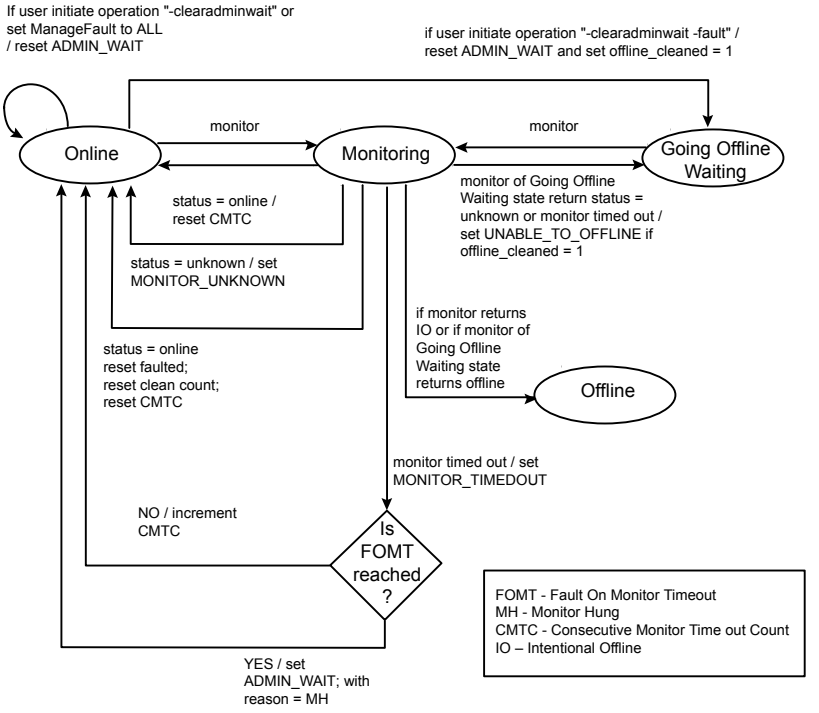


Figure 10-30 Resource fault (monitor hung): ManageFaults attribute = NONE



Chapter 11

Internationalized messages

This chapter includes the following topics:

- About internationalized messages
- Creating SMC files
- Converting SMC files to BMC files
- Using BMC Map Files
- Updating BMC Files

About 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

Writer: introductory text is required here.

```
#!language = language_ID
#!module = module_name
#!version = version
#!category = category_ID

# comment
message_ID1 {%s:msg}
message_ID2 {%s:msg}
message_ID3 {%s:msg}

# comment
message_ID4 {%s:msg}
message_ID5 {%s:msg}
...
```

Example SMC file

Examine an example SMC file:

```
VRTSvcsSunAgent.smc
```

The sample 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.
  A discussion of file naming conventions is available.
  See "Naming SMC files, BMC files" on page 249.
- 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" on page 248.
- 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" on page 253.
- 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|Lnx}{Agent_name}.bmc

where the *platform* and *agent_name* are included.

For example:

```
VRTSvcsLnxOracle.bmc
```

Message examples

- An illegal message, with hard carriage returns embedded with the message:

```
201 {"%s:To be or not to be!
    That is the question"}
```

- A valid message using "\n":

```
10010 {"%s:To be or not to be!\n
That is the question"}
```

- A valid message with text wrapping to the next line:

```
10012 {"%s:To be or not to be!\n
That is the question.\n Whether tis nobler in the mind to
suffer\n the slings and arrows of outrageous fortune\n or to
take arms against a sea of troubles"}
```

Using format specifiers

Using the "%s" specifier is appropriate for all message arguments unless the arguments must be reordered. Since the word order in messages may vary by language, a format specifier,%#$s, enables the reordering of arguments in a message; the "#" character is a number from 1 to 99.

In an English SMC file, the entry might resemble:

```
301 {"%s:Setting cookie for proc=%s, PID = %s"}
```

In a language where the position of message arguments need to switch, the same entry in the SMC file for that language might resemble:

```
301 {"%s:Setting cookie for process with PID = %3$s, name =
%2$s"}
```

Converting SMC files to BMC files

Use the `bmcgen` utility to convert SMC files to BMC files. For example:

```
bmcgen VRTSvcsLnxAgent.smc
```

The file `VRTSvcsLnxAgent.bmc` is created in the directory where the SMC file exists. A BMC file must have an extension: `.bmc`.

By default, the `bmcgen` utility assumes the SMC file is a Unicode (UCS-2) file. For ASCII or UTF-8 encoded files, use the `-ascii` option. For example:

```
bmcgen -ascii VRTSvcsSunAgent.smc
```

Storing BMC files

By default, BMC files must be installed in `/opt/VRTS/messages/`*`language`*, where *language* is a directory containing the BMCs of a given supported language. For example, the path to the BMC for a Japanese agent on a Solaris system resembles:

```
/opt/VRTS/messages/ja/VRTSvcsSunAgent.bmc.
```

VCS languages

The languages supported by VCS are listed as subdirectories, such as `/ja` (Japanese) and `/en` (English), in the directory `/opt/VRTS/messages`.

Displaying the contents of BMC files

The `bmcread` command enables you to display the contents of the binary message catalog file. For example, the following command displays the contents of the specified BMC file:

```
bmcread VRTSvcsLnxAgent.bmc
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: `/opt/VRTS/messages/`***language***.

Creating BMC Map Files

Developers can add BMCs to the BMC map file. After generating a BMC file:

1. Copy the BMC file to the corresponding directory. For example:

   ```
   cp VRTSvcsLnxOracle.bmc /opt/VRTS/messages/en
   ```

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

   ```
   cd /opt/VRTS/messages/en
   bmcmap -create en HAD
   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.bmc map is created.

Example BMC Map File

An example of a BMC Map file named HAD.bmcmap on a Solaris system.

```
# This is a program generated file, please do not edit.

Language=en

HAD=VRTSvcsHad VRTSvcsAgfw VRTSvcsWac \
    VRTSvcsHbfw VRTSvcsAlerts VRTSvcsTriggers \
    VRTSvcsApi gcoconfig fdsetup \
    hazonesetup hagetcf uuidconfig \
    hazoneverify VRTSvcsSunAgent VRTSvcsCommonAgent \
    VRTSvcsOracle VRTSvcsDb2udb VRTSvcsCVMCluster \
```

```
VRTSvcsCVMVolDg VRTSvcsSunVVR VRTSvcsCFSMount \
VRTSvcsSybase VRTSvcsCVMVxconfigd VRTSvcsCFSfsckd \
VRTSvcsCommon

VRTSvcsHad.version=5.1
VRTSvcsHad.category=1
VRTSvcsHad.IDstart=0
VRTSvcsHad.IDend=53502

VRTSvcsAgfw.version=5.1
VRTSvcsAgfw.category=2
VRTSvcsAgfw.IDstart=0
VRTSvcsAgfw.IDend=60019

VRTSvcsWac.version=5.1
VRTSvcsWac.category=3
VRTSvcsWac.IDstart=0
VRTSvcsWac.IDend=53006

VRTSvcsHbfw.version=5.1
VRTSvcsHbfw.category=4
VRTSvcsHbfw.IDstart=0
VRTSvcsHbfw.IDend=13301

VRTSvcsAlerts.version=5.1
VRTSvcsAlerts.category=5
VRTSvcsAlerts.IDstart=10018
VRTSvcsAlerts.IDend=52026
```

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:

- 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.

- Generate the new BMC file using the `bmcgen` command; place the new BMC file in the corresponding language directory.
- In the directory containing the BMC file, run the `bmcmap` command to create a new BMC map file.

About 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

Writer: introductory text is required here.

```
#!language = language_ID
#!module = module_name
#!version = version
#!category = category_ID

# comment
```

```
message_ID1 {%s:msg}
message_ID2 {%s:msg}
message_ID3 {%s:msg}

# comment
message_ID4 {%s:msg}
message_ID5 {%s:msg}
...
```

Example SMC file

Examine an example SMC file:

```
VRTSvcsSunAgent.smc
```

The sample 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.
  A discussion of file naming conventions is available.
  See “Naming SMC files, BMC files” on page 249.
- 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” on page 248.
- 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" on page 253.

- 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|Lnx}{Agent_name}.bmc

where the *platform* and *agent_name* are included.

For example:

```
VRTSvcsLnxOracle.bmc
```

Message examples

- An illegal message, with hard carriage returns embedded with the message:

```
201 {"%s:To be or not to be!
    That is the question"}
```

- A valid message using "\n":

```
10010 {"%s:To be or not to be!\n
That is the question"}
```

- A valid message with text wrapping to the next line:

```
10012 {"%s:To be or not to be!\n
That is the question.\n Whether tis nobler in the mind to
suffer\n the slings and arrows of outrageous fortune\n or to
take arms against a sea of troubles"}
```

Using format specifiers

Using the "%s" specifier is appropriate for all message arguments unless the arguments must be reordered. Since the word order in messages may vary by language, a format specifier,%#$s, enables the reordering of arguments in a message; the "#" character is a number from 1 to 99.

In an English SMC file, the entry might resemble:

```
301 {"%s:Setting cookie for proc=%s, PID = %s"}
```

In a language where the position of message arguments need to switch, the same entry in the SMC file for that language might resemble:

```
301 {"%s:Setting cookie for process with PID = %3$s, name =
%2$s"}
```

Converting SMC files to BMC files

Use the `bmcgen` utility to convert SMC files to BMC files. For example:

```
bmcgen VRTSvcsLnxAgent.smc
```

The file `VRTSvcsLnxAgent.bmc` is created in the directory where the SMC file exists. A BMC file must have an extension: `.bmc`.

By default, the `bmcgen` utility assumes the SMC file is a Unicode (UCS-2) file. For ASCII or UTF-8 encoded files, use the `-ascii` option. For example:

```
bmcgen -ascii VRTSvcsSunAgent.smc
```

Storing BMC files

By default, BMC files must be installed in `/opt/VRTS/messages/`*`language`*, where *language* is a directory containing the BMCs of a given supported language. For example, the path to the BMC for a Japanese agent on a Solaris system resembles:

```
/opt/VRTS/messages/ja/VRTSvcsSunAgent.bmc.
```

VCS languages

The languages supported by VCS are listed as subdirectories, such as `/ja` (Japanese) and `/en` (English), in the directory `/opt/VRTS/messages`.

Displaying the contents of BMC files

The `bmcread` command enables you to display the contents of the binary message catalog file. For example, the following command displays the contents of the specified BMC file:

```
bmcread VRTSvcsLnxAgent.bmc
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: `/opt/VRTS/messages/`*language*.

Creating BMC Map Files

Developers can add BMCs to the BMC map file. After generating a BMC file:

1. Copy the BMC file to the corresponding directory. For example:

   ```
   cp VRTSvcsLnxOracle.bmc /opt/VRTS/messages/en
   ```

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

   ```
   cd /opt/VRTS/messages/en
   bmcmap -create en HAD
   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.bmc map is created.

Example BMC Map File

An example of a BMC Map file named HAD.bmcmap on a Solaris system.

```
# This is a program generated file, please do not edit.

Language=en

HAD=VRTSvcsHad VRTSvcsAgfw VRTSvcsWac \
    VRTSvcsHbfw VRTSvcsAlerts VRTSvcsTriggers \
    VRTSvcsApi gcoconfig fdsetup \
    hazonesetup hagetcf uuidconfig \
    hazoneverify VRTSvcsSunAgent VRTSvcsCommonAgent \
    VRTSvcsOracle VRTSvcsDb2udb VRTSvcsCVMCluster \
    VRTSvcsCVMVolDg VRTSvcsSunVVR VRTSvcsCFSMount \
    VRTSvcsSybase VRTSvcsCVMVxconfigd VRTSvcsCFSfsckd \
    VRTSvcsCommon

    VRTSvcsHad.version=5.1
    VRTSvcsHad.category=1
    VRTSvcsHad.IDstart=0
    VRTSvcsHad.IDend=53502

    VRTSvcsAgfw.version=5.1
    VRTSvcsAgfw.category=2
    VRTSvcsAgfw.IDstart=0
    VRTSvcsAgfw.IDend=60019

    VRTSvcsWac.version=5.1
    VRTSvcsWac.category=3
    VRTSvcsWac.IDstart=0
    VRTSvcsWac.IDend=53006

    VRTSvcsHbfw.version=5.1
    VRTSvcsHbfw.category=4
    VRTSvcsHbfw.IDstart=0
    VRTSvcsHbfw.IDend=13301

    VRTSvcsAlerts.version=5.1
    VRTSvcsAlerts.category=5
    VRTSvcsAlerts.IDstart=10018
    VRTSvcsAlerts.IDend=52026
```

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:

- 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.
- Generate the new BMC file using the `bmcgen` command; place the new BMC file in the corresponding language directory.
- In the directory containing the BMC file, run the `bmcmap` command to create a new BMC map file.

Chapter 12

Troubleshooting VCS resource's unexpected behavior using First Failure Data Capture (FFDC)

This chapter includes the following topics:

Enhancing First Failure Data Capture (FFDC) to troubleshoot VCS resource's unexpected behavior

FFDC is the process of generating and dumping debug information on unexpected events.

Earlier, FFDC information is generated on following unexpected events:

- Segmentation fault
- When agent fails to heartbeat with engine

From VCS 6.2 version, the capturing of the FFDC information on unexpected events has been extended to resource level and to cover VCS events. This means, if a resource faces an unexpected behavior then FFDC information will be generated.

The current version enables the agent to log detailed debug logging during unexpected events with respect to resource, such as,

- Monitor entry point of a resource reported OFFLINE/IO when it was in ONLINE state.
- Monitor entry point of a resource reported UNKNOWN.
- If any entry point times-out.
- If any entry point reports failure.

Now whenever an unexpected event occurs FFDC information will be automatically generated. And this information will be logged in their respective agent log file.

Enhancing First Failure Data Capture (FFDC) to troubleshoot VCS resource's unexpected behavior

FFDC is the process of generating and dumping debug information on unexpected events.

Earlier, FFDC information is generated on following unexpected events:

- Segmentation fault
- When agent fails to heartbeat with engine

From VCS 6.2 version, the capturing of the FFDC information on unexpected events has been extended to resource level and to cover VCS events. This means, if a resource faces an unexpected behavior then FFDC information will be generated.

The current version enables the agent to log detailed debug logging during unexpected events with respect to resource, such as,

- Monitor entry point of a resource reported OFFLINE/IO when it was in ONLINE state.
- Monitor entry point of a resource reported UNKNOWN.
- If any entry point times-out.
- If any entry point reports failure.

Now whenever an unexpected event occurs FFDC information will be automatically generated. And this information will be logged in their respective agent log file.

Appendix A

Using pre-5.0 VCS agents

This appendix includes the following topics:

- Using pre-5.0 VCS agents and registering them with V50 or later
- Guidelines for using pre-VCS 4.0 Agents
- Log messages in pre-VCS 4.0 agents
- Pre-VCS 4.0 Message APIs

Using pre-5.0 VCS agents and registering them with V50 or later

With VCS 5.0 release, the agent framework has been enhanced. For using this enhanced agent framework for your agent, you need to register the agent with the agent framework version V50 or later. 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 or later agents after making necessary modifications. Making this conversion affords you advantages, which include:

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

Outline of steps to change V40 agents to V50 or later

- Modifications to PATH variables and links to the agent binary registered with agent version V50 or later may be necessary.
- Change the way attributes and their values are passed to the entry points from the V40 format to V50 or later name-value tuple format.
- Include /opt/VRTSvcs/lib in path for Perl and shell to source them.
- Set necessary environment variables.

See "About the ArgList and ArgListValues attributes" on page 44.

Example script in V40 and V50 or later

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 or later

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

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

Sourcing ag_i18n_inc modules in script entry points

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

For entry points in Perl:

```
...
$ResName = shitf;
use ag_i18n_inc;
```

```
VCSAG_SET_ENVS ($ResName);
...
```

For entry points in Shell:

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

Guidelines for using pre-VCS 4.0 Agents

The agent framework supports all VCS agents by enabling them to communicate with the engine about the definitions of resource types, the values configured for the resource attributes, and entry points they use.

Changes made to the agent framework with VCS 4.0 and VCS 5.0 releases affect how agents developed using the pre-VCS 4.0 agent framework can be used. While not necessary, all pre-VCS 4.0 agents may be modified to work with the VCS 4.0 and later agent framework so that the new entry points can be used.

Note the following guidelines:

- If the pre-VCS 4.0 agent is implemented strictly in scripts, then the VCS 4.0 and later ScriptAgent can be used on UNIX. If desired, the VCS 4.0 and later `action` and `info` entry points can be used directly.
- If the pre-VCS 4.0 agent is implemented using any C++ entry points, the agent can be used if developers do not care to implement the `action` or `info` entry points. The VCS 4.0 and later agent framework assumes all pre-VCS 4.0 agents are version 3.5.
- If the pre-VCS 4.0 agent is implemented using any C++ entry points, and you want to implement the `action` or the `info` entry point:
  - Add the `action` or `info` entry point, C++ or script-based, to the agent.
  - Use the API VCSAgInitEntryPointStruct with the parameter V40 to register the agent as a VCS 4.0 agent. Use the VCSAgValidateAndSetEntryPoint API to register your C++ entry points.
  - Recompile the agent.

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

Log messages in pre-VCS 4.0 agents

The log messages in pre-VCS 4.0 agents are automatically converted to the VCS 4.0 and later message format.

See *Logging agent messages* for more information.

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 *Logging agent messages* for more information.

For the purpose of comparison, the examples that follow show a pair of messages in C++ that are formatted using the pre-VCS 4.0 API and the VCS 4.0 macros.

- Pre-VCS 4.0 APIs:

```
sprintf(msg,
"VCS:140003:FileOnOff:%s:online:The value for PathName attribute
is not specified", res_name);
       VCSAgLogI18NMsg(TAG_C, msg, 140003,
              res_name, NULL, NULL, NULL, LOG_DEFAULT);
       VCSAgLogI18NConsoleMsg(TAG_C, msg, 140003, res_name,
              NULL,NULL,NULL,LOG_DEFAULT);
```

- VCS 4.0 macros:

```
VCSAG_LOG_MSG(VCS_WARNING, 14003, VCS_DEFAULT_FLAGS,
       "The value for PathName attribute is not specified");
VCSAG_CONSOLE_LOG_MSG(VCS_WARNING, 14003, VCS_DEFAULT_FLAGS,
       "The value for PathName attribute is not specified");
```

Pre-VCS 4.0 Message APIs

The message APIs described in this section of the document are maintained to allow VCS 4.0 and later to work with the agents developed on the 2.0 and 3.5 agent framework.

VCSAgLogConsoleMsg

```
void
VCSAgLogConsoleMsg(int tag, const char *message, int flags);
```

This primitive requests that the VCS agent framework write `message` to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log.

The `message` must not exceed 4096 bytes. A message greater that 4096 bytes is truncated.

`tag` can be any value from TAG_A to TAG_Z. Tags A 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"
...

VCSAgLogConsoleMsg(TAG_A, "Getting low on disk space",
                          LOG_TAG|LOG_TIMESTAMP);
...
```

VCSAgLogI18NMsg

```
void
VCSAgLogI18NMsg(int tag, const char *msg,
    int msg_id, const char *arg1_string, const char
*arg2_string,
    const char *arg3_string, const char *arg4_string, int
flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and four string arguments to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log

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

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

For example:

```
#include "VCSAgApi.h"
...
char buffer[256];
sprintf(buffer, "VCS:2015001:IP:%s:monitor:Device %s address
   %s", res_name, device, address);
```

```
VCSAgLogI18NConsoleMsg(TAG_B, buffer, 2015001, res_name, device,
   address, NULL, LOG_TAG|LOG_TIMESTAMP|LOG_NEWLINE);
```

VCSAgLogI18NMsgEx

```
void
VCSAgLogI18NMsgEx(int tag, const char *msg,
    int msg_id, const char *arg1_string, const char
*arg2_string,
    const char *arg3_string, const char *arg4_string,
    const char *arg5_string, const char *arg6_string, int
flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and six string arguments to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log

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

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

For example:

```
#include "VCSAgApi.h"
...
char buffer[256];
sprintf(buffer, "VCS:2015004:Oracle:%s:%s:During scan for
    process %s ioctl failed with return code %s, errno = %s",
res_name, ep_name, proc_name, ret_buf, err_buf);

VCSAgLogI18NConsoleMsgEx(TAG_A, buffer, 2015004, res_name,
ep_name, proc_name, ret_buf, err_buf, NULL, flags);
```

VCSAgLogI18NConsoleMsg

```
void
VCSAgLogI18NConsoleMsg(int tag,
    const char *msg, int msg_id, const char *arg1_string,
    const char *arg2_string, const char *arg3_string,
    const char *arg4_string, int flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and four string arguments to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log

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

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

For example:

```
#include "VCSAgApi.h"
...

char buffer[256];
sprintf(buffer, "VCS:2015002:IP:%s:monitor:Device %s address
    %s", res_name, device, address);

VCSAgLogI18NConsoleMsg(TAG_B, buffer, 2015002, res_name, device,
    address, NULL, LOG_TAG|LOG_TIMESTAMP|LOG_NEWLINE);
```

VCSAgLogI18NConsoleMsgEx

```
void
VCSAgLogI18NConsoleMsgEx(int tag,
    const char *msg, int msg_id, const char *arg1_string,
    const char *arg2_string, const char *arg3_string,
    const char *arg4_string, const char *arg5_string,
    const char *arg6_string, int flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and six string arguments to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log

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

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

For example:

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

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

Using pre-5.0 VCS agents and registering them with V50 or later

With VCS 5.0 release, the agent framework has been enhanced. For using this enhanced agent framework for your agent, you need to register the agent with the agent framework version V50 or later. 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 or later agents after making necessary modifications. Making this conversion affords you advantages, which include:

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

Outline of steps to change V40 agents to V50 or later

- Modifications to PATH variables and links to the agent binary registered with agent version V50 or later may be necessary.
- Change the way attributes and their values are passed to the entry points from the V40 format to V50 or later name-value tuple format.
- Include /opt/VRTSvcs/lib in path for Perl and shell to source them.
- Set necessary environment variables.

See "About the ArgList and ArgListValues attributes" on page 44.

Example script in V40 and V50 or later

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 or later

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

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

Sourcing ag_i18n_inc modules in script entry points

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

For entry points in Perl:

```
...
$ResName = shitf;
use ag_i18n_inc;
```

```
VCSAG_SET_ENVS ($ResName);
...
```

For entry points in Shell:

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

Guidelines for using pre-VCS 4.0 Agents

The agent framework supports all VCS agents by enabling them to communicate with the engine about the definitions of resource types, the values configured for the resource attributes, and entry points they use.

Changes made to the agent framework with VCS 4.0 and VCS 5.0 releases affect how agents developed using the pre-VCS 4.0 agent framework can be used. While not necessary, all pre-VCS 4.0 agents may be modified to work with the VCS 4.0 and later agent framework so that the new entry points can be used.

Note the following guidelines:

- If the pre-VCS 4.0 agent is implemented strictly in scripts, then the VCS 4.0 and later ScriptAgent can be used on UNIX. If desired, the VCS 4.0 and later `action` and `info` entry points can be used directly.
- If the pre-VCS 4.0 agent is implemented using any C++ entry points, the agent can be used if developers do not care to implement the `action` or `info` entry points. The VCS 4.0 and later agent framework assumes all pre-VCS 4.0 agents are version 3.5.
- If the pre-VCS 4.0 agent is implemented using any C++ entry points, and you want to implement the `action` or the `info` entry point:
  - Add the `action` or `info` entry point, C++ or script-based, to the agent.
  - Use the API VCSAgInitEntryPointStruct with the parameter V40 to register the agent as a VCS 4.0 agent. Use the VCSAgValidateAndSetEntryPoint API to register your C++ entry points.
  - Recompile the agent.

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

Log messages in pre-VCS 4.0 agents

The log messages in pre-VCS 4.0 agents are automatically converted to the VCS 4.0 and later message format.

See *Logging agent messages* for more information.

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 *Logging agent messages* for more information.

For the purpose of comparison, the examples that follow show a pair of messages in C++ that are formatted using the pre-VCS 4.0 API and the VCS 4.0 macros.

- Pre-VCS 4.0 APIs:

```
sprintf(msg,
"VCS:140003:FileOnOff:%s:online:The value for PathName attribute
is not specified", res_name);
       VCSAgLogI18NMsg(TAG_C, msg, 140003,
              res_name, NULL, NULL, NULL, LOG_DEFAULT);
       VCSAgLogI18NConsoleMsg(TAG_C, msg, 140003, res_name,
              NULL,NULL,NULL,LOG_DEFAULT);
```

- VCS 4.0 macros:

```
VCSAG_LOG_MSG(VCS_WARNING, 14003, VCS_DEFAULT_FLAGS,
       "The value for PathName attribute is not specified");
VCSAG_CONSOLE_LOG_MSG(VCS_WARNING, 14003, VCS_DEFAULT_FLAGS,
       "The value for PathName attribute is not specified");
```

Pre-VCS 4.0 Message APIs

The message APIs described in this section of the document are maintained to allow VCS 4.0 and later to work with the agents developed on the 2.0 and 3.5 agent framework.

VCSAgLogConsoleMsg

```
void
VCSAgLogConsoleMsg(int tag, const char *message, int flags);
```

This primitive requests that the VCS agent framework write `message` to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log.

The `message` must not exceed 4096 bytes. A message greater that 4096 bytes is truncated.

`tag` can be any value from TAG_A to TAG_Z. Tags A 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"
...

VCSAgLogConsoleMsg(TAG_A, "Getting low on disk space",
                          LOG_TAG|LOG_TIMESTAMP);
...
```

VCSAgLogI18NMsg

```
void
VCSAgLogI18NMsg(int tag, const char *msg,
    int msg_id, const char *arg1_string, const char
*arg2_string,
    const char *arg3_string, const char *arg4_string, int
flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and four string arguments to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log

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

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

For example:

```
#include "VCSAgApi.h"
...
char buffer[256];
sprintf(buffer, "VCS:2015001:IP:%s:monitor:Device %s address
   %s", res_name, device, address);
```

```
VCSAgLogI18NConsoleMsg(TAG_B, buffer, 2015001, res_name, device,
   address, NULL, LOG_TAG|LOG_TIMESTAMP|LOG_NEWLINE);
```

VCSAgLogI18NMsgEx

```
void
VCSAgLogI18NMsgEx(int tag, const char *msg,
    int msg_id, const char *arg1_string, const char
*arg2_string,
    const char *arg3_string, const char *arg4_string,
    const char *arg5_string, const char *arg6_string, int
flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and six string arguments to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log

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

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

For example:

```
#include "VCSAgApi.h"
...
char buffer[256];
sprintf(buffer, "VCS:2015004:Oracle:%s:%s:During scan for
    process %s ioctl failed with return code %s, errno = %s",
res_name, ep_name, proc_name, ret_buf, err_buf);

VCSAgLogI18NConsoleMsgEx(TAG_A, buffer, 2015004, res_name,
ep_name, proc_name, ret_buf, err_buf, NULL, flags);
```

VCSAgLogI18NConsoleMsg

```
void
VCSAgLogI18NConsoleMsg(int tag,
    const char *msg, int msg_id, const char *arg1_string,
    const char *arg2_string, const char *arg3_string,
    const char *arg4_string, int flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and four string arguments to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log

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

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

For example:

```
#include "VCSAgApi.h"
...

char buffer[256];
sprintf(buffer, "VCS:2015002:IP:%s:monitor:Device %s address
    %s", res_name, device, address);

VCSAgLogI18NConsoleMsg(TAG_B, buffer, 2015002, res_name, device,
    address, NULL, LOG_TAG|LOG_TIMESTAMP|LOG_NEWLINE);
```

VCSAgLogI18NConsoleMsgEx

```
void
VCSAgLogI18NConsoleMsgEx(int tag,
    const char *msg, int msg_id, const char *arg1_string,
    const char *arg2_string, const char *arg3_string,
    const char *arg4_string, const char *arg5_string,
    const char *arg6_string, int flags);
```

This primitive requests that the VCS agent framework write an internationalized `message` with a message ID and six string arguments to the agent log file

UNIX: $VCS_LOG/log/resource_type_A.log

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

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

For example:

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

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