VERITAS® Cluster Server

Agent Developer's Guide

Version 1.2

July 1999
P/N 100-001165

TRADEMARKS

Table of Contents

Preface

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

For information on installing and configuring VCS, see the following documents:

- *VERITAS Cluster Server Installation Guide.*
- *VERITAS Cluster Server Quick-Start Guide.*

For information on using VCS, see the *VERITAS Cluster Server User's Guide.*

How This Guide is Organized

Chapter 1, “Introduction,” provides an overview of how agents work, and outlines the requirements for building an agent.

Chapter 2, “VCS Agent Entry Points,” includes a list of entry points and their definitions.

Chapter 3, “Implementing Entry Points Using C++,” describes how to implement the VCS entry points using C++. This chapter also describes the VCS agent primitives.

Chapter 4, “Implementing Entry Points Using Scripts,” describes how to implement the VCS entry points using scripts. This chapter also describes the script syntax for entry points.

Chapter 5, “Building a Custom VCS Agent,” provides step-by-step instructions for building a custom agent using C++ and scripts.

Chapter 6, “Setting Agent Parameters,” describes each agent parameter and default.

Chapter 7, “Testing VCS Agents,” provides two methods for testing VCS agents: AgentServer and the VCS engine had.

Technical Support

For assistance with this product, contact VERITAS Customer Support:

U.S. and Canadian Customers: 1-800-342-0652
International Customers: +1 (650) 335-8555
Fax: (650) 335-8428
Email: support@veritas.com

Conventions

Typeface	Usage
`courier`	computer output, files, attribute names, device names, and directories
`courier (bold)`	user input and commands, keywords in grammar syntax
italic	new terms, titles, emphasis, variables replaced with a name or value
italic (bold)	variables within a command
Symbol	**Usage**
%	C shell prompt
$	Bourne/Korn shell prompt
#	Superuser prompt (for all shells)
\	Command-line continuation if last character in line. Not to be confused with an escape character.

Introduction

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

Agents packaged with VCS are referred to as *bundled agents*. Examples of bundled agents include Share, IP (Internet Protocol) and NIC (Network Interface Card) agents. For more information on bundled agents, including their attributes and modes of operation, see the *VERITAS Cluster Server Installation Guide*.

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

Additional agents can be developed easily using the VCS agent framework included in the VCS package.

How Agents Work

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

When the VCS engine had comes up on a system, it automatically starts the required agents according to the type of configuration. When an agent is started, it "pulls" the necessary configuration information from had. It then periodically monitors the resources and updates had with their status.

The agent also carries out online and offline commands received from had. If an agent crashes or hangs, had detects it and restarts the agent.

Prerequisites

Before proceeding, make sure you have defined the resource type; specifically, that you've defined the attributes of the resource type in the file `types.cf`, and that you understand the semantics of the online, offline, and monitor operations. (If necessary, review the information on the VCS configuration language in the *VERITAS Cluster Server User's Guide*.)

VCS Agent Entry Points

Developing a VCS agent requires using the agent framework and implementing *entry points*. An entry point is a *plug-in*, defined by the user, that is called when an event occurs within the VCS agent. An entry point can be a C++ function or a script.

The VCS agent framework supports the entry points listed below. With the exception of `VCSAgStartup` and `monitor`, all entry points are optional. Definitions of each entry point begin on page 5.

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

The VCSAgStartup entry point must be implemented using C++. Other entry points may be implemented using C++ or scripts.

Note: If there are no other entry points in C++, you don't have to provide your own VCSAgStartup. You can use the VCSAgStartup provided by the agent framework. (See ScriptAgent in Chapter 5, "Building a Custom VCS Agent.")

The advantage to using C++ is that entry points are compiled and linked with the agent framework library to form the agent binary. They run as part of the agent process, so there is no system overhead when they are called. The advantage to using scripts is that you can modify the entry points dynamically; however, a new process is created each time they are called. Note that you may use any combination of C++, Perl, and shell to implement multiple entry points for a single agent.

The VCS agent framework ensures that a resource has only one entry point running at a time. If multiple requests or events are received for the same resource, they are queued, then processed one at a time. However, because the agent framework is multithreaded, a single agent process can run entry points of several resources simultaneously. For example, if a resource receives requests to offline first, then close, the offline entry point is called first. The close entry point is called only after the offline request returns or times out. However, if the offline request is received for one resource, and the close request is received for another, both are called simultaneously.

List of Agent Entry Points

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

VCSAgStartup

As stated previously, VCSAgStartup is required if other entry points are implemented using C++. This entry point is called once when the VCS agent starts. It receives no inputs and returns no value.

VCSAgStartup must register all entry points with the agent framework by calling the primitive VCSAgSetEntryPoints (VCSAgEntryPointStruct &ep). The structure VCSAgEntryPointStruct consists of function pointers, one for each VCS entry point except VCSAgStartup. (For information on VCS primitives, see page 29.)

Sample Structure

```
// Structure used to register the entry points.
typedef struct {
    void (*open)(const char *res_name,void **attr_val);
    void (*close)(const char *res_name,void **attr_val);
    VCSAgResState (*monitor)(const char
         *res_name, void **attr_val,
         int *conf_level);
    unsigned int (*online)(const char *res_name,
         void **attr_val);
```

```
        unsigned int (*offline) (const char *res_name,
             void **attr_val);
        void (*attr_changed) (const char *res_name,
             const char *changed_res_name, const char
             *changed_attr_name, void **new_val);
        unsigned int (*clean) (const char *res_name,
             VCSAgWhyClean reason, void **attr_val);
        void (*shutdown) ();
    } VCSAgEntryPointStruct;
```

When using C++ to implement an entry point, assign the function to the corresponding field of `VCSAgEntryPointStruct`. In the example on page 7, the function `my_shutdown` is assigned to the field `shutdown`. If you are using a script, or if you are not implementing an optional entry point, set the corresponding field to `NULL`.

For an agent to run an entry point whose field is set to `NULL`, the agent automatically looks for the script `$VCS_HOME/bin/`*resource_type/entry_point*, then executes the script if it exists.

The following example shows the `VCSAgStartup` entry point for a VCS agent implementing the `shutdown` entry point only. (Note that the `monitor` entry point is mandatory. In the following example, it is implemented using scripts.)

```
#include "VCSAgApi.h"
void my_shutdown() {
    ...
}
void VCSAgStartup() {
    VCSAgEntryPointStruct ep;
    ep.open = NULL;
    ep.online = NULL;
    ep.offline = NULL;
    ep.monitor = NULL;
    ep.attr_changed = NULL;
    ep.clean = NULL;
    ep.close = NULL;
    ep.shutdown = my_shutdown;

    VCSAgSetEntryPoints(ep);
}
```

monitor

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

Note: This entry point is mandatory.

The framework calls this entry point after completing the `online` and `offline` entry points. The `monitor` entry point determines if bringing the resource online or taking it offline was effective. The agent framework may also periodically call this entry point to detect if the resource was brought online or taken offline unexpectedly.

The `monitor` entry point receives a resource name and `ArgList` attribute values as input (see “ArgList” on page 60). It returns the resource status (online, offline, or unknown), and the confidence level 0–100. The confidence level is informative only; it is not used by VCS. It is returned only when the resource status is online.

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

- 100 indicates offline.
- 101 indicates online and confidence level 10.
- 102 indicates online and confidence level 20.
- 110 indicates online and confidence level 100.

online

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

The `online` entry point receives a resource name and `ArgList` attribute values as input. It returns an integer indicating the number of seconds to wait for the online to take effect. The typical return value is 0.

offline

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

The `offline` entry point receives a resource name and `ArgList` attribute values as input. It returns an integer indicating the number of seconds to wait for the offline to take effect. The typical return value is 0.

clean

The `clean` entry point is called when all ongoing actions associated with a resource must be terminated and the resource must be taken offline, perhaps forcibly. It receives as input the resource name, an encoded reason describing why this 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
};
```

- **VCSAgCleanOfflineHung**

  The `offline` entry point did not complete within the expected time. (See “OfflineTimeout” on page 62.)

- **VCSAgCleanOfflineIneffective**

  The `offline` entry point was ineffective.

- **VCSAgCleanOnlineHung**

  The `online` entry point did not complete within the expected time. (See “OnlineTimeout” on page 62.)

- **VCSAgCleanOnlineIneffective**

  The `online` entry point was ineffective.

- **VCSAgCleanUnexpectedOffline**

  The `online` resource was taken offline unexpectedly. The agent is configured to automatically restart the resource. (See “RestartLimit” on page 63.)

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

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

For the above actions to occur, the `clean` entry point must return 0 within the time set in the `CleanTimeOut` attribute.

Implement this entry point only if there is a guaranteed method of taking a resource offline or terminating the effects of a hung or ineffective online attempt.

attr_changed

The `attr_changed` entry point is called when a resource attribute is modified, and only if that resource is registered with the agent framework for notification. See the primitives `VCSAgRegister()`and `VCSAgUnregister()`for details (page 33). To register automatically, see the `RegList` parameter described on page 63. This entry point receives as input the resource name registered with the agent framework for notification, the name of the changed resource, the name of the changed attribute, and the new attribute value. It does not return a value. This entry point provides a way to respond to resource changes. Most agents do not require this functionality and will not implement this entry point.

open

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

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

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

close

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

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

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

shutdown

The `shutdown` entry point is called before the VCS agent shuts down. It receives no input and returns no value.

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

Implementing Entry Points Using C++

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

Because the agent framework is multithreaded, all C++ code written by the agent developer must be MT-safe. We offer the following guidelines:

- Avoid using global variables. However, if you do use them, access must be serialized (for example, by using mutex locks).
- 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()`.
- Do not explicitly declare `errno`, as in `extern int errno`. Instead, include the header file *errno.h*, which provides the necessary declaration for obtaining thread-specific `errno` values.
- When acquiring resources (dynamically allocating memory, opening a file, etc.), 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.)

Data Structures

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

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

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

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

typedef struct {
    void (*open)(const char *res_name, void **attr_val);
    void (*close)(const char *res_name, void **attr_val);
    VCSAgResState (*monitor)(const char *res_name,
              void **attr_val, int, *conf_level);
    unsigned int (*online)(const char *res_name,
                           void **attr_val);
    unsigned int (*offline)(const char *res_name,
                            void **attr_val);
    void (*attr_changed)(const char *res_name,
        const char *changed_res_name, const char
        *changed_attr_name, void **new_val);
    unsigned int (*clean)(const char *res_name,
        VCSAgWhyClean reason, void **attr_val);
    void (*shutdown) ();
} VCSAgEntryPointStruct;
```

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

ArgList Attributes

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

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

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

The values of scalar attributes (integer and string) are each contained in a single element of attr_val. The values of non-scalar attributes (vector, keylist, and association) are contained in one or more elements of attr_val. If a non-scalar attribute contains *N* components, it will have *N*+1 elements in attr_val. The first element is *N*, and the remaining *N* elements correspond to the *N* components. (See page 60 for more information on ArgList.
See the chapter describing the VCS configuration language in the *VERITAS Cluster Server User's Guide* for attribute definitions.)

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

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

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

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

The parameter `attr_val` will be:

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

C++ Entry Point Syntax

VCSAgStartup

```
void VCSAgStartup();
```

The entry point `VCSAgStartup()` must use the VCS primitive `VCSAgSetEntryPoints()` to register the other entry points with the VCS agent framework. (VCS primitives are described on page 29.) Note that the name of the C++ function must be `VCSAgStartup()`.

For example:

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

#include "VCSAgApi.h"

void VCSAgStartup() {
    VCSAgEntryPointStruct ep;
      ep.open = NULL;
      ep.close = NULL;
      ep.monitor = res_monitor;
      ep.online = res_online;
      ep.offline = res_offline;
      ep.attr_changed = NULL;
      ep.clean = NULL;
      ep.shutdown = NULL;
      VCSAgSetEntryPoints(ep);
}
```

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

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

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

monitor

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

The parameter `conf_level` is an output parameter.

You may select any name for the function. The `monitor` field of `VCSAgEntryPointStruct` passed to `VCSAgSetEntryPoints()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

VCSAgResState
my_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() {
  VCSAgEntryPointStruct ep;
  ...
  ep.monitor = my_monitor;
  ...
  VCSAgSetEntryPoints(ep);
}
```

online

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

You may select any name for the function. The `online` field of `VCSAgEntryPointStruct` passed to `VCSAgSetEntryPoints()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

unsigned int
my_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() {
    VCSAgEntryPointStruct ep;
    ...
    ep.online = my_online;
    ...
    VCSAgSetEntryPoints(ep);
}
```

offline

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

You may select any name for the function. The `offline` field of `VCSAgEntryPointStruct` passed to `VCSAgSetEntryPoints()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

unsigned int
my_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() {
    VCSAgEntryPointStruct ep;
    ...
    ep.offline = my_offline;
    ...
    VCSAgSetEntryPoints(ep);
}
```

clean

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

You may select any name for the function. The `clean` field of `VCSAgEntryPointStruct` passed to `VCSAgSetEntryPoints()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

unsigned int
my_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() {
    VCSAgEntryPointStruct ep;
    ...
    ep.clean = my_clean;
    ...
    VCSAgSetEntryPoints(ep);
}
```

attr_changed

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

The parameter `new_val` contains the attribute's new value. The encoding of `new_val` is similar to the encoding of the `ArgList` attributes described on page 16. You may select any name for the function. The `attr_changed` field of `VCSAgEntryPointStruct` passed to `VCSAgSetEntryPoints()` must be assigned a pointer to this function.

Note: This entry point is called only if you register for change notification using the primitive `VCSAgRegister()` described on page 32, or the agent parameter `RegList` described on page 63.

For example:

```
#include "VCSAgApi.h"

void
my_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() {
    VCSAgEntryPointStruct ep;
    ...
    ep.attr_changed = my_attr_changed;
    ...
    VCSAgSetEntryPoints(ep);
}
```

open

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

You may select any name for the function. The `open` field of `VCSAgEntryPointStruct` passed to `VCSAgSetEntryPoints()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

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

void VCSAgStartup() {
  VCSAgEntryPointStruct ep;
  ...
  ep.open = my_open;
  ...
  VCSAgSetEntryPoints(ep);
}
```

close

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

You may select any name for the function. The `close` field of `VCSAgEntryPointStruct` passed to `VCSAgSetEntryPoints()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

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

void VCSAgStartup() {
    VCSAgEntryPointStruct ep;
    ...
    ep.close = my_close;
    ...
    VCSAgSetEntryPoints(ep);
}
```

shutdown

```
void shutdown();
```

You may select any name for the function. The `shutdown` field of `VCSAgEntryPointStruct` passed to `VCSAgSetEntryPoints()` must be assigned a pointer to this function.

For example:

```
#include "VCSAgApi.h"

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

void VCSAgStartup() {
    VCSAgEntryPointStruct ep;
    ...
    ep.shutdown = my_shutdown;
    ...
    VCSAgSetEntryPoints(ep);
}
```

VCS Primitives

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

VCSAgSetEntryPoints

```
void VCSAgSetEntryPoints(VCSAgEntryPointStruct&
                         entry_points);
```

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

For example:

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

#include "VCSAgApi.h"

void VCSAgStartup() {
    VCSAgEntryPointStruct ep;
    ep.open = NULL;
    ep.close = NULL;
    ep.monitor = res_monitor;
    ep.online = res_online;
    ep.offline = res_offline;
    ep.attr_changed = NULL;
    ep.clean = NULL;
    ep.shutdown = NULL;
    VCSAgSetEntryPoints(ep);
}
```

VCSAgSetCookie

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

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

For example:

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

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

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

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

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

        return state;
}
```

VCSAgRegister

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

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

For example:

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

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

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

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

}
```

VCSAgUnregister

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

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

For example:

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

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

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

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

}
```

VCSAgGetCookie

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

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

For example:

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

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

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

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

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

        return state;
}
```

VCSAgLogMsg

```
void
VCSAgLogMsg(int tag, const char *message, int flags);
```

This primitive requests that the VCS agent framework log `message` to the agent log file `$VCS_`*`LOG`*`/log/`*`resource_type`*`_A`. Note that `tag` can be any value from TAG_A to TAG_Z.

`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"
...

VCSAgLogMsg(TAG_E, "This is a debug message",
            LOG_TAG|LOG_TIMESTAMP|LOG_NEWLINE);
...
```

VCSAgLogConsoleMsg

```
void
VCSAgLogConsoleMsg(int tag, const char *message,
                   int flags);
```

This primitive requests that the VCS agent framework write `message` to the VCS engine log. It can be called from any entry point. Note that `tag` can be any value from TAG_A to TAG_Z.

Note: Tags A–E are enabled by default. To enable tags F–Z, see the *VERITAS Cluster Server User's Guide*.

`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);
...
```

Implementing Entry Points Using Scripts

As mentioned in Chapter 2, the `VCSAgStartup` entry point must be implemented using C++. Other entry points may be implemented using C++ or scripts. If no other entry points are implemented in C++, implementing `VCSAgStartup` is not required. Instead, use the `VCSAgStartup` provided by the agent framework. (See ScriptAgent in Chapter 5.)

You must adhere to the following rules when implementing a script entry point:

- ✓ In the `VCSAgStartup` entry point, set the corresponding field of `VCSAgEntryPointStruct` to `NULL` prior to calling `VCSAgSetEntryPoints()`. (If necessary, review page 5.)
- ✓ Verify that the name of the script file is the same as the name of the entry point.
- ✓ Place the script file under the directory `$VCS_`*`HOME`*`/bin/`*`resource_type`*. For example, if the `online` entry point for Oracle is implemented using scripts, the `online` script must be `$VCS_`*`HOME`*`/bin/Oracle/online`.

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

ArgList Attributes

The `open`, `close`, `online`, `offline`, `monitor`, and `clean` scripts receive the resource name and values of the `ArgList` attributes. The values of scalar `ArgList` attributes (integer and string) are each contained in a single command-line argument. The values of complex `ArgList` attributes (vector and association) are contained in one or more command-line arguments. If a vector or association attribute contains *N* components, it will be represented by *N*+1 command-line arguments. The first command-line argument is *N*; the remaining *N* arguments correspond to the *N* components. (See page 60 for more information on `ArgList`. See the chapter on the VCS configuration language in the *VERITAS Cluster Server User's Guide* for attribute definitions.)

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

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

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

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

The `online` script for `Bar` is invoked as:

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

Script Entry Point Syntax

monitor

`monitor` *resource ArgList_attribute_values*

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

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

If the exit value falls outside the range 100–110, the status is considered unknown.

online

`online` *resource ArgList_attribute_values*

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

offline

`offline` *resource ArgList_attribute_values*

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

clean

`clean` *resource 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.

`1` - The `offline` entry point was ineffective.

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

`3` - The `online` entry point was ineffective.

`4` - The resource was taken offline unexpectedly.

The exit value is 0 (successful), or 1.

attr_changed

`attr_changed` *resource_name changed_resource_name changed_attribute_name new_attribute_value*

The exit value is ignored.

Note: This entry point is called only if you register for change notification using the primitive `VCSAgRegister()` described on page 32, or the agent parameter `RegList` described on page 63.

---

open

`open` *resource_name values_of_ArgList_attributes*

The exit value is ignored.

close

`close` *resource_name values_of_ArgList_attributes*

The exit value is ignored.

shutdown

`shutdown`

The exit value is ignored.

Logging

Messages directed to the `stdout` and `stderr` of the script entry points are captured and sent to the global log. Additionally, script entry points can send any message to the global log using the `halog` command. See the *VERITAS Cluster Server User's Guide* for more information on `halog`.

Building a Custom VCS Agent

The VRTSvcs package includes the following files to facilitate agent development:

For Script Agents Only

- **$VCS_HOME/bin/ScriptAgent**

  Ready-to-use VCS agent that includes a built-in implementation of the VCSAgStartup entry point. Note that ScriptAgent cannot be used with C++ entry points.

For C++ Agents Only

- **$VCS_HOME/src/agent/Sample**

  Directory containing a sample C++ agent and Makefile.

- **$VCS_HOME/src/agent/Sample/Makefile**

  Sample Makefile for building a C++ agent.

- **$VCS_HOME/src/agent/Sample/agent.C**

  Entry point templates for C++ agents.

Compiling is not required if all entry points are implemented using scripts. A copy of, or a link to, `ScriptAgent` is sufficient. However, if any entry points are implemented using C++, compiling is required to build the agent. In this case, we recommend that you edit `agent.C` in the directory `$VCS_HOME/src/agent/Sample` to customize the implementation. After completing the changes to `agent.C`, invoke `make` from `$VCS_HOME/src/agent/Sample` to build the agent.

We also recommend naming the agent binary "*resource_type*Agent" and placing it in the directory `$VCS_HOME/bin/`*resource_type*. If the agent binary is named differently or placed in a different directory, the `AgentFile` attribute must be assigned the complete pathname of the agent binary in the `types.cf` file.

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

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

If entry points are implemented using scripts, the script file must be placed in the directory `$VCS_HOME/bin/`*resource_type*. For the Oracle example, if the `online` entry point is implemented using scripts, the script file must be named `online` and must exist in the directory `$VCS_HOME/bin/Oracle`.

The following sections describe different ways to build a VCS agent for "MyFile" resources:

- ✓ `ScriptAgent` and script entry points (page 48).
- ✓ `VCSAgStartup` and script entry points (page 50).
- ✓ C++ and script entry points (page 52).
- ✓ C++ entry points (page 55).

A MyFile resource represents a regular file. The MyFile `online` entry point creates the file if it does not already exist. The MyFile `offline` entry point deletes the file. The MyFile `monitor` entry point returns online and confidence level 100 if the file exists; otherwise, it returns offline.

The examples below use the following type and resource definitions:

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

// Define a MyFile resource (in main.cf).
MyFile (
    PathName = "/tmp/VRTSvcs_file1"
    Enabled = 1
)
```

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

Using ScriptAgent and Script Entry Points

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

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

   ```
   # mkdir /opt/VRTSvcs/bin/MyFile
   ```

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

   To copy the agent binary:

   ```
   # cp /opt/VRTSvcs/bin/ScriptAgent \
       /opt/VRTSvcs/bin/MyFile/MyFileAgent
   ```

   To create a link to the agent binary:

   ```
   # ln -s /opt/VRTSvcs/bin/ScriptAgent \
       /opt/VRTSvcs/bin/MyFile/MyFileAgent
   ```

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

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

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

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

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

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

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

Using VCSAgStartup() and Script Entry Points

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

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

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

2. Copy the contents of the directory /opt/VRTSvcs/src/agent/Sample to /opt/VRTSvcs/src/agent/MyFile:

   ```
   # cp /opt/VRTSvcs/src/agent/Sample/* \
       /opt/VRTSvcs/src/agent/MyFile
   ```

3. Change to the directory /opt/VRTSvcs/src/agent/MyFile:

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

4. Edit the file agent.C and modify the VCSAgStartup()function (the last 15 lines) to match the following example:

   ```
   void VCSAgStartup() {
       VCSAgEntryPointStruct ep;
       // Set all the entry point fields to NULL because
       // this example does not implement any of them
       // using C++.
       ep.open = NULL;
       ep.close = NULL;
       ep.monitor = NULL;
       ep.online = NULL;
       ep.offline = NULL;
       ep.attr_changed = NULL;
       ep.clean = NULL;
       ep.shutdown = NULL;
       VCSAgSetEntryPoints(ep);
   }
   ```

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

   ```
   # make
   ```

6. Create the directory `/opt/VRTSvcs/bin/MyFile`:

   ```
   # mkdir /opt/VRTSvcs/bin/MyFile
   ```

7. Install the `MyFile` agent built in step 5.

   ```
   # make install AGENT=MyFile
   ```

8. Implement the `online`, `offline`, and `monitor` entry points, as instructed in step 3 on page 49.

Using C++ and Script Entry Points

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

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

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

2. Copy the contents of the directory /opt/VRTSvcs/src/agent/Sample to /opt/VRTSvcs/src/agent/MyFile:

   ```
   # cp /opt/VRTSvcs/src/agent/Sample/* \
         /opt/VRTSvcs/src/agent/MyFile
   ```

3. Change to the directory /opt/VRTSvcs/src/agent/MyFile:

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

4. Edit the file agent.C and modify the VCSAgStartup() function (the last 15 lines in the file) to match the following example:

   ```
   void VCSAgStartup() {

     VCSAgEntryPointStruct ep;

     // This example implements only the monitor entry
     // point using C++. Set all the entry point
     // fields, except monitor, to NULL.
     ep.open = NULL;
     ep.close = NULL;
     ep.monitor = res_monitor;
     ep.online = NULL;
     ep.offline = NULL;
     ep.attr_changed = NULL;
     ep.clean = NULL;
     ep.shutdown = NULL;
     VCSAgSetEntryPoints(ep);
   }
   ```

5. Modify the `res_monitor()` function:

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

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

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

  // Return the status of the resource.
  return state;

}
```

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

   ```
   # make
   ```

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

   ```
   # mkdir /opt/VRTSvcs/bin/MyFile
   ```

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

   ```
   # make install AGENT=MyFile
   ```

9. Implement the `online`, `offline`, and `monitor` entry points, as instructed in step 3 on page 49.

Using C++ Entry Points

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

1. Edit the file agent.C and modify the VCSAgStartup() function (the last 15 lines) to match the following example:

```
void VCSAgStartup() {
  VCSAgEntryPointStruct ep;
  // This example implements online, offline, and
  // monitor entry points using C++. Set the
  // corresponding fields of VCSAgEntryPointStruct
  // passed to VCSAgSetEntryPoints. Set all other
  // fields to NULL.
  ep.open = NULL;
  ep.close = NULL;
  ep.monitor = res_monitor;
  ep.online = res_online;
  ep.offline = res_offline;
  ep.attr_changed = NULL;
  ep.clean = NULL;
  ep.shutdown = NULL;
  VCSAgSetEntryPoints(ep);
}
```

2. Modify `res_online()` and `res_offline()`:

```
// This is a C++ implementation of the online entry
// point for the MyFile resource type. This function
// brings online a MyFile 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) {
  if (attr_val) {
    // Get the pathname of the file.
    const char *path_name = (const char *) attr_val[0];

    // Create the file.
    int fd = creat (path_name,S_IRUSR | S_IWUSR);
    if (fd < 0) {
      // if create() failed, send a log message to
      // the console.
      char msg [1024];
      sprintf (msg,
         "Resource(%s) -creat() failed for file(%s)",
         res_name, path_name);
      VCSAgLogConsoleMsg (TAG_A, msg,
         LOG_TIMESTAMP | LOG_NEWLINE | LOG_TAG);
    }
    else {
      close(fd);
    }
  }
```

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

// This is a C++ implementation of the offline entry
// point for the MyFile resource type. This function
// takes offline a MyFile resource by deleting the
// corresponding file. It is assumed that the complete
// pathname of the file will be passed as the first
// ArgList attribute.
unsigned int
res_offline(const char *res_name, void **attr_val) {
  if (attr_val) {
    // Get the pathname of the file.
    const char *path_name = (const char *)
                            attr_val[0];
    // Delete the file
    remove (path_name);
  }
  // Completed offlining of resource. Return 0 so
  // monitor can start immediately. Note that return
  // value indicates how long agent framework must wait
  // before calling the monitor entry point to check if
  // offline was successful.
  return 0;
}
```

3. Modify `res_monitor()`, as shown on page 53.
4. Compile `agent.C` and build the agent by invoking `make`. (`Makefile` is provided.)

   ```
   # make
   ```

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

   ```
   # mkdir /opt/VRTSvcs/bin/MyFile
   ```

6. Install the MyFile agent built in step 4.

   ```
   # make install AGENT=MyFile
   ```

Setting Agent Parameters

The VCS agents can be customized for a resource type by setting the values of the agent parameters. Agent parameters are predefined static attributes of the resource type. They can be assigned values when defining the resource type in `types.cf`, and they can be set dynamically using the command `hatype -modify`. Beginning with `AgentFile` below, each agent parameter is listed and defined in the following sections.

AgentFile

Name of the agent file to be executed. The default is `$VCS_HOME/bin/`*resource_type*`/`*resource_type*`Agent`.

AgentReplyTimeout

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

AgentStartTimeout

After the engine has started the agent, this is the amount of time the engine waits for the initial agent “handshake” before attempting to restart. Default is 60 seconds.

ArgList

An ordered list of attributes whose values are passed to the `open`, `close`, `online`, `offline`, `monitor`, and `clean` entry points. Default is empty list.

ArgList Reference Attributes

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

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

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

AttrChangedTimeout

Maximum time (in seconds) within which the `attr_changed` entry point must complete or else be terminated. Default is 60 seconds.

CloseTimeout

Maximum time (in seconds) within which the `close` entry point must complete or else be terminated. Default is 60 seconds.

CleanTimeout

Maximum time (in seconds) within which the `clean` entry point must complete or else be terminated. Default is 60 seconds.

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`.

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

LogLevel

Specifies the type of messages to be logged to the log file `$VCS_LOG/log/`*res_type*`_A` (local to the system). The possible values are:

- **all**

  Log all messages. (Not recommended.)

- **debug**

  Log all messages, except function-tracing messages. (Not recommended.)

- **info**

  Log only error messages and messages useful to the agent developer.

- **error**

  Log only error messages (default).

- **none**

  Log no messages.

MonitorInterval

Duration (in seconds) between two consecutive monitor calls for a resource. Default is 60 seconds.

MonitorTimeout

Maximum time (in seconds) within which the `monitor` entry point must complete or else be terminated. Default is 60 seconds.

OfflineTimeout

Maximum time (in seconds) within which the `offline` entry point must complete or else be terminated. Default is 300 seconds.

OnlineRetryLimit

Number of times to retry `online`, if the attempt to online a resource is unsuccessful. This parameter is meaningful only if `clean` is implemented. Default is 0.

OnlineTimeout

Maximum time (in seconds) within which the `online` entry point must complete or else be terminated. Default is 300 seconds.

OnlineWaitLimit

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

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

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

OpenTimeout

Maximum time (in seconds) within which the `open` entry point must complete or else be terminated. Default is 60 seconds.

RestartLimit

Affects how the agent responds to a resource fault. A non-zero `RestartLimit` causes VCS to invoke the `online` entry point instead of failing over the service group to another system. VCS attempts to restart the resource according to the number set in `RestartLimit` before it gives up and fails over (applicable only if the `clean` is implemented). However, if the resource remains online for the interval designated in `ConfInterval`, any earlier attempts to restart are not counted against `RestartLimit`. Default is 0.

RegList

Keylist of attribute names. All resources are automatically registered for change notification for specified attributes. See page 11 for details. Default is empty list.

ToleranceLimit

A non-zero `ToleranceLimit` allows the `monitor` entry point to return `OFFLINE` several times before the 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.

Testing VCS Agents

VCS agents can be tested using the VCS engine, or using the AgentServer utility. In either case, you can activate the agent debug messages by setting the value of the `LogLevel` attribute of the resource type to `info`. The debug messages are logged to the file `$VCS_LOG/log/`*resource_type*`_A`.

Before testing an agent, complete the following requirements:

- ✓ Build the agent binary and install it in the directory `$VCS_HOME/bin/`*resource_type*.
- ✓ Install script entry points in the directory `$VCS_HOME/bin/`*resource_type*. (See Chapter 4, “Implementing Entry Points Using Scripts,” for details.)
- ✓ If you are using the VCS engine, define the resource type in `types.cf`, the resources in `main.cf`, and restart the engine. Or define the new type and resources using commands listed in the *VERITAS Cluster Server User’s Guide*.

Using the VCS Engine (had)

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

After had comes up, type the following command to verify that the agent has been started and is running:

```
# ps -ef | grep resource_type
```

For example, to test the Oracle agent, type:

```
# ps -ef | grep Oracle
```

If the Oracle agent is running, the output resembles:

```
root   2220      1   2 10:29:29 ?       0:00
/opt/VRTSvcs/bin/Oracle/OracleAgent -type Oracle
```

Test Commands

To activate agent debug messages:

```
# hatype -modify resource_type LogLevel info
```

To check the status of a resource:

```
# hares -display resource_name
```

To bring a resource online:

```
# hares -online resource_name -sys system_name
```

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

To take a resource offline:

```
# hares -offline resource_name -sys system_name
```

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

To deactivate agent debug messages:

```
# hatype -modify resource_type LogLevel error
```

Using AgentServer

The AgentServer utility enables you to test agents independent of had. It is part of the VCS package and is installed under the $VCS_HOME/bin directory. Instructions for using this utility begin on the next page.

To Access Help

When AgentServer is started, you will receive a message prompting you to enter a command or to type help for the complete list of the AgentServer commands. Type help to review the commands before getting started.

Output resembles:

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

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

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

```
# help onlineres
```

Output resembles:

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

To test the FileOnOff Agent:

1. Type the following command to start AgentServer:

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

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

2. Start the agent for the resource type:

   ```
   # startagent FileOnOff /opt/VRTSvcs/bin \
       /FileOnOff/FileOnOffAgent
   ```

   You will receive the following messages:

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

3. Review the sample configuration:

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

main.cf:
  ...
  group ga (
  ...
  )

  FileOnOff file1 (
    Enabled = 1
    PathName = "/tmp/VRTSvcsfile001"

  )
```

4. Pass this sample configuration to the VCS agent.

   a. Add a type:

```
# addtype FileOnOff FileOnOff
```

   b. Add attributes of the type:

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

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

```
# addstaticattr FileOnOff FileOnOff ArgList \
        vector PathName
```

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

```
# addstaticattr FileOnOff FileOnOff LogLevel \
        str info
```

e. Add a resource:

```
# addres FileOnOff file1 FileOnOff
```

f. Set the resource attributes:

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

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

```
# proberes FileOnOff file1
```

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

You will receive the following messages indicating the resource status:

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

a. To bring a resource online:

```
# onlineres FileOnOff file1
```

This calls the `offline` entry point of the FileOnOff agent. When `file1` is brought online or taken offline, the following message is displayed:

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

b. To take a resource offline:

```
# offlineres FileOnOff file1
```

This calls the `offline` entry point of the FileOnOff agent. When `file1` is taken offline, a status message is displayed.

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

```
# print
```

Output resembles:

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

7. Stop the agent:

```
# stopagent FileOnOff
```

8. Exit from the AgentServer:

```
# quit
```