VERITAS Cluster Server™ 1.3.0

Agent Developer's Guide

UNIX and Windows NT

October 2000
30-000033-399

VERITAS

Disclaimer

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

Copyright

Printed in the USA, October 2000.

VERITAS Software Corporation
1600 Plymouth St.
Mountain View, CA 94043
Phone 650–335–8000
Fax 650–335–8050
www.veritas.com

Contents

Preface

Introduction

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

Note Custom agents are not supported by VERITAS Technical Support.

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 Bundled Agents Reference Guide*.

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

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

How Agents Work

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

When the VCS engine process, HAD, comes up on a system, it automatically starts the required agents according to the type of configuration. When an agent is started, it gets 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.

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

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 regarding the VCS configuration language in the *VERITAS Cluster Server User's Guide*.)

How This Guide is Organized

Chapter 1, "VCS Agent Entry Points," provides a list of VCS entry points and explains how to implement them.

Chapter 2, "Implementing Entry Points Using C++," describes how to implement the VCS entry points using C++. This chapter also describes the VCS agent primitives.

Chapter 3, "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 4, "Building a Custom VCS Agent," provides step-by-step instructions for building a custom agent using C++ and scripts.

Chapter 5, "Setting Agent Parameters," describes each agent parameter and default.

Chapter 6, "Testing VCS Agents," provides two methods for testing VCS agents: the AgentServer utility and the VCS engine process "HAD."

Technical Support

For assistance with this product, or information regarding VERITAS service packages, contact Technical Support at the numbers listed below. You may also contact Technical Support via email at support@veritas.com.

VCS on UNIX: from U.S and Canada, call 800.342.0652.

VCS on Windows NT: from U.S and Canada, call 800.634.4747. To access Fast Code, enter 100 after the phone number.

For Customers Outside U.S. and Canada

From Europe, Middle East, or Asia, visit the Technical Support website at http:/ /support.veritas.com for a list of each country's contact information.

Conventions

Typeface	Usage
`courier`	computer output, command references within text
`courier` (bold)	user input, keywords in grammar syntax
italic	new terms, titles, emphasis, variables replaced with a name or value
italic (bold)	variables within a command

Symbol	Usage
#	UNIX superuser prompt (for all shells)
C:\>	DOS command prompt

VCS Agent Entry Points 1

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 3.

- 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 "Using Script Entry Points" on page 42.)

The advantage to using C++ is that entry points are compiled and linked with the agent framework library. They run as part of the agent process, so there is no system overhead when they are called. The advantage to using scripts is that you can modify the entry points dynamically; however, a new process is created each time 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 25.)

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 following example, 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 correct script to execute.

- **On UNIX:** $VCS_HOME/bin/*resource_type/entry_point*.
- **On Windows NT:** VCS_HOME\bin*resource_type**entry_point.extn*.

Note On Windows NT, the extension (*.extn*) assumes the value **exe** for executable programs, **sh** for shell scripts compatible with MKS Toolkit, **bat** for batch files, or **pl** for Perl scripts.

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 62). 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.
- 103–109 indicates online and confidence levels 30–90.
- 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 automatically by the framework when all ongoing actions associated with a resource must be terminated and the resource must be taken offline, perhaps forcibly. The entry point receives as input the resource name, an encoded reason describing why the entry point is being called, and the ArgList attribute values. It must return 0 if the operation is successful, and 1 if unsuccessful.

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

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

- VCSAgCleanOfflineHung

  The `offline` entry point did not complete within the expected time. (See "OnlineTimeout" on page 65.)

- VCSAgCleanOfflineIneffective

  The `offline` entry point was ineffective.

- VCSAgCleanOnlineHung

  The `online` entry point did not complete within the expected time. (See "OnlineTimeout" on page 65.)

- VCSAgCleanOnlineIneffective

  The `online` entry point was ineffective.

- VCSAgCleanUnexpectedOffline

  The online resource faulted because it was taken offline unexpectedly.

- VCSAgCleanMonitorHung

  The online resource faulted because the `monitor` entry point consistently failed to complete within the expected time. (See "FaultOnMonitorTimeouts" on page 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.

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 28). To register automatically, see the `RegList` parameter described on page 66. 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. Most agents do not require this functionality and will not implement this entry point.

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

shutdown

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

Implementing Entry Points Using C++ 2

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

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

On UNIX, 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()`.
- 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.
      VCSAgCleanMonitorHung   // monitor entry point did
                               // not complete within the
                              // expected time.
};
// Structure used to register the entry points.

typedef struct {
      void (*open)(const char *res_name, void **attr_val);
      void (*close)(const char *res_name, void **attr_val);
      VCSAgResState (*monitor)(const char *res_name,
                  void **attr_val, int, *conf_level);
      unsigned int (*online)(const char *res_name,
                           void **attr_val);
      unsigned int (*offline)(const char *res_name,
                         void **attr_val);
      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 Attribute

The ArgList attribute is a predefined static attribute that specifies the list of attributes whose values are passed to the `open`, `close`, `online`, `offline`, 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 62 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 primitive `VCSAgSetEntryPoints()` to register the other entry points with the VCS agent framework. (VCS primitives are described on page 25.) 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. The return value, which indicates the resource status, must be a VCSAgResState value defined on page 11.

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 13.

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 28, or the agent parameter `RegList` described on page 66.

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 write `message` to the following agent log file,

- **On UNIX:** $VCS_LOG/log/*resource_type*_A.log.
- **On Windows NT:** VCS_HOME\log*resource_type*_A.txt.

`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 following VCS engine log file.

- **On UNIX:** $VCS_LOG/log/`engine`_A.log.
- **On Windows NT:** VCS_HOME\log\`engine`_A.txt.

`Tag` can be any value from TAG_A to TAG_Z. Tags A–E are enabled by default. `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 3

As mentioned in Chapter 2, on UNIX 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, UNIX developers may use ScriptAgent, and Windows NT developers may use VCSdefault.dll. See page 43 for examples of each.

Script entry points can be executables or scripts, such as shell or Perl (VCS includes a Perl distribution). On Windows NT, batch (.bat) files can also be used. To use shell scripts on Windows NT, you must first install MKSToolkit or a similar software application. Also, the PATH environment variable must include the directory where `sh` is installed.

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

- ✔ In the `VCSAgStartup` entry point, set the corresponding field of `VCSAgEntryPointStruct` to `NULL` prior to calling `VCSAgSetEntryPoints()`. (If necessary, review page 3.)
- ✔ Place the script file in the correct directory after verifying the name of the script file.
  - **On UNIX**:
    - Verify that the name of the script file is the same as the entry point.
    - Place the file in the directory $VCS_HOME/bin/*resource_type*.
  - **On Windows NT**:
    - Verify that the name of the script file is the same as the entry point, followed by a period and the appropriate extension (see page 4).
    - Place the file in the directory VCS_HOME\bin*resource_type*.

    For example, if the `online` entry point for Oracle is implemented using Perl, the `online` script must be:

    On UNIX: $VCS_HOME/bin/Oracle/online

    On Windows NT: VCS_HOME\bin\Oracle\online.pl

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. When using batch files on Windows NT, beware of a subtle problem in passing the exit code. The value returned by the final batch command is presented as the exit value of the program. There is no direct way to exit a batch program with a particular value. For example, when executing the batch command `exit 100`, the exit code of the program is 0 and *not* 100. This is because the `exit` command does not look at the arguments, and always exits with 0. To solve this problem, VCS on Windows NT includes a small program called `exitcode.exe` in the directory VCS_HOME\bin. This program exits with the value passed as the argument, so it can be used as the last command of a batch file to force a specific exit value. For example, the following batch file has the exit value of 110:

```
echo Hello World
"%VCS_HOME%\bin\exitcode" 110
```

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 is represented by *N*+1 command-line arguments. The first command-line argument is *N*, and the remaining *N* arguments correspond to the *N* components. (See page 62 for more information on ArgList. See the chapter on the VCS configuration language in the *VERITAS Cluster Server User's Guide* for attribute definitions.)

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

```
Type Foo (
      str Name
      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" }
)
```

On UNIX: the `online` script for `Bar` is invoked as:

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

On Windows NT: the Perl script `online.pl` script for `Bar` is invoked as:

```
perl online.pl 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.

5 - The monitor entry point consistently failed to complete within the expected time. (See “FaultOnMonitorTimeouts” on page 63.)

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 28, or the agent parameter `RegList` described on page 66.

open

`open` *resource_name ArgList_attribute_values*

The exit value is ignored.

close

`close` *resource_name ArgList_attribute_values*

The exit value is ignored.

shutdown

`shutdown`

The exit value is ignored.

Logging

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 `halog`(1M) manual page for more information.

Message Numbering

All VCS agent log messages contain message numbers in the range of 0 - 1,000,000. Log messages from agents developed by VERITAS consultants contain message numbers in the range of 1,000,001 - 2,000,000. The log message number range available for customer custom agents is: 2,000,001 and beyond.

Building a Custom VCS Agent 4

The VRTSvcs package includes the following files to facilitate agent development on UNIX and Windows NT platforms. Note that custom agents are not supported by VERITAS Technical Support.

Script Agents

Description	Pathname: UNIX	Pathname: Windows NT
Ready-to-use VCS agent that includes a built-in implementation of the `VCSAgStartup` entry point.	$VCS_HOME/bin/ScriptAgent **Note** ScriptAgent cannot be used be used with C++ entry points.	VCS_HOME\bin\VCSdefault.dll **Note** VCSdefault.dll cannot be used with C++ entry points.

C++ Agents

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

Compiling is not required if all entry points are implemented using scripts. A copy of `ScriptAgent` is sufficient on UNIX; a copy of VCSdefault.dll is sufficient on Windows NT.

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

1. Edit `agent.C` (UNIX) or `sample.C` (Windows NT) to customize the implementation.
   - **On UNIX:** `agent.C` is located in the directory $VCS_HOME/src/agent/Sample.
   - **On Windows NT:** `sample.C` is located in the directory VCS_HOME\src\agent\Sample.
2. After completing the changes to `agent.C` or `sample.C`, invoke the command to build the agent.
   - **On UNIX:** the command is `make`, and it is invoked from $VCS_HOME/src/agent/Sample.
   - **On Windows NT:** the command is `nmake`, and it is invoked from VCS_HOME\src\agent\Sample.

Additional Recommendations

We also recommend naming the agent binary (UNIX) or DLL (Windows NT) *resource_type*Agent or *resource_type*.dll, respectively.

- **On UNIX:** place the agent in the directory $VCS_HOME/bin/*resource_type*.

  For example, on UNIX the agent binary for Oracle would be $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"
      ...
)
```

- **On Windows NT:** place the agent in VCS_HOME\bin*resource_type*. The agent DLL for Oracle would be VCS_HOME\bin\Oracle\Oracle.dll.

If entry points are implemented using scripts, the script file must be placed in a specific directory, and must be named correctly (if necessary, review page 4).

- **On UNIX:** the directory for the script file is $VCS_HOME/bin/*resource_type*.
- **On Windows NT:** the directory for the script file is VCS_HOME\bin*resource_type*.

  For example, if the `online` entry point for Oracle is implemented using Perl, the `online` script must be:

  - **On UNIX:** $VCS_HOME/bin/Oracle/online
  - **On Windows NT:** VCS_HOME\bin\Oracle\online.pl

Building a VCS Agent for MyFile Resources

The following sections describe different ways to build a VCS agent for "MyFile" resources. For test purposes, instructions for installing the agent on a single VCS system are also provided. For multi-system configurations, you must install the agent on each system in the cluster. Note that examples are included for UNIX and Windows NT developers.

The UNIX examples assume that VCS is installed under /opt/VRTSvcs. If your VCS installation directory is different, change the commands accordingly. The Windows NT examples use the keyword VCS_HOME to denote the VCS installation directory. Substitute VCS_HOME with the actual directory name when entering commands.

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 in this chapter 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" (UNIX) or
               "C:\\temp\\VRTSvcs_file1" (Windows NT)
      Enabled = 1
   )
```


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

Using Script Entry Points

The following example shows how to build the MyFile agent without writing and compiling any C++ code. This example implements the `online`, `offline`, and `monitor` entry points only. (UNIX users, see the example below. Windows NT users, see the example on page 43.)

Building an Agent Using Script Entry Points on UNIX

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

Building an Agent Using Script Entry Points on Windows NT

1. Create the directory VCS_HOME\bin\MyFile:

```
C:\> mkdir VCS_HOME\bin\MyFile
```

2. Use VCS_HOME\bin\VCSdefault.dll as the agent DLL.

   Copy the file to VCS_HOME\bin\MyFile.dll, as shown below:

```
C:\> cp VCS_HOME\bin\VCSdefault.dll
                VCS_HOME\bin\MyFile\MyFile.dll
```

3. Implement the `online`, `offline`, and `monitor` entry points using scripts. The examples below use shell scripts. MKS Toolkit or equivalent software is required for running shell scripts. Entry points can also be implemented as batch files.

   a. Using any editor, create the fileVCS_HOME\MyFile\online.sh with the following contents:

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

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

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

   c. Create the file VCS_HOME\bin\MyFile\monitor with the following contents:

```
# Verify that file specified by 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. (UNIX users, see the example below. Windows NT users, see the example on page 46.)

Building an Agent Using VCSAgStartup() and Script Entry Points on UNIX

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

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

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

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

3. Change to the new directory:

   ```
   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 42.

Building an Agent Using VCSAgStartup() and Script Entry Points on Windows NT

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

1. Create the directory VCS_HOME\src\agent\MyFile:

   ```
   C:\> mkdir VCS_HOME\src\agent\MyFile
   ```

2. Copy the contents of the directory VCS_HOME\src\agent\Sample to the directory you created in the previous step:

   ```
   C:\> cp VCS_HOME\src\agent\Sample\* VCS_HOME\src\agent\MyFile
   ```

3. Change to the new directory:

   ```
   C:\> cd VCS_HOME\src\agent\MyFile
   ```

4. Edit the file `sample.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 `sample.C` and build the agent by invoking `nmake`. (`Makefile` is provided.)

   ```
   C:\> nmake
   ```

6. Create the directory VCS_HOME\bin\MyFile:

   ```
   C:\> mkdir VCS_HOME\bin\MyFile
   ```

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

   ```
   C:\> copy sample.dll VCS_HOME\bin\MyFile\MyFile.dll
   ```

8. Implement the `online`, `offline`, and `monitor` entry points, as instructed in step 3 on page 43.

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. (UNIX users, see the example below. Windows NT users, see the example on page 51.)

Building an Agent Using C++ and Script Entry Points on UNIX

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

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

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

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

3. Change to the new directory:

   ```
   # 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;

   // 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` and `offline` entry points, as instructed in step 3 on page 42.

Building an Agent Using C++ and Script Entry Points on Windows NT

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

1. Create the directory VCS_HOME\src\agent\MyFile:

```
C:\> mkdir VCS_HOME\src\agent\MyFile
```

2. Copy the contents of the directory VCS_HOME\src\agent\Sample to the directory you created in the previous step:

```
C:\> cp VCS_HOME\src\agent\Sample\* VCS_HOME\src\
            agent\MyFile
```

3. Change to the new directory:

```
C:\> cd VCS_HOME\src\agent\MyFile
```

4. Edit the file `sample.C` and modify the `VCSAgStartup()` function (the last 15 lines) 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.
      DWORD attrs = GetFileAttributes(path_name);
      if (attrs!=0xffffffff) {
         state = VCSAgResOnline;
         *conf_level = 100;
      }
      else {
         state = VCSAgResOffline;
         *conf_level = 0;
      }
   }

   // Return the status of the resource.

   return state;
}
```

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

   ```
   C:\> nmake
   ```

7. Create the directory C:\Program Files\VERITAS\cluster server\bin\MyFile:

   ```
   C:\> mkdir VCS_HOME\bin\MyFile
   ```

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

   ```
   C:\> copy sample.dll VCS_HOME\bin\MyFile\MyFile.dll
   ```

9. Implement the `online` and `offline` entry points, as instructed in step 3 on page 43.

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. (UNIX users, see the example below. Windows NT users, see the example on page 57.)

Building an Agent Using C++ Entry Points on UNIX

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 creat() 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 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 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 49.

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
   ```

Building an Agent Using C++ Entry Points on Windows NT

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

1. Create the directory VCS_HOME\src\agent\MyFile:

```
C:\> mkdir VCS_HOME\src\agent\MyFile
```

2. Copy the contents of the directory VCS_HOME\src\agent\Sample to the directory you created in the previous step:

```
C:\> cp VCS_HOME\src\agent\Sample\* VCS_HOME\src\
            agent\MyFile
```

3. Change to the new directory:

```
C:\> cd VCS_HOME\src\agent\MyFile
```

4. Edit the file `sample.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);
}
```


5. 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.
      HANDLE h = CreateFile(path_name,
         GENERIC_READ|GENERIC_WRITE,
         0, // Not Shared
         NULL, // Default Security
         OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL,
         (HANDLE)NULL
         );
      if(h == INVALID_HANDLE_VALUE) {
         char msg[1024];
         sprintf(msg
            “CreateFile() failed with (%d) for (%s)”,
            GetLastError(), path_name);
         VCSAgLogMsg(TAG_A, msg, (LOG_TIMESTAMP|LOG_NEWLINE|
            LOG_TAG));
      }
      else {
         CloseHandle (h);
      }
   }
```

```
   // 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
      if(!DeleteFile(path_name)) {
         char msg[1024];
         sprintf(msg,
            "DeleteFile() failed with (%d) for (%s)",
            GetLastError(), path_name);
         VCSAgLogMsg(TAG_A, msg,
            LOG_TIMESTAMP|LOG_NEWLINE|LOG_TAG));
      }
   }

   // 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;
}
```


6. Modify the `res_monitor()` function, as instructed in step 5 on page 52.

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

   ```
   C:\> nmake
   ```

8. Create the directory VCS_HOME\bin\MyFile:

   ```
   C:\> mkdir VCS_HOME\bin\MyFile
   ```

9. Place the sample.dll as the MyFile agent library.

   ```
   C:\> copy sample.dll VCS_HOME\bin\MyFile\MyFile.dll
   ```

Setting Agent Parameters 5

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` (described in the *VERITAS Cluster Server User's Guide*). Beginning with AgentFile below, each agent parameter is listed and defined in the following sections.

Note VCS now allows you to specify priorities and scheduling classes for VCS processes. For details on the additional parameters included with this feature, and for instructions on initializing them in the types.cf file or setting them from the command line, see "Scheduling Class and Priority Configuration Support" on page 66.

AgentFile

The name of the agent file to be executed:

- **On UNIX**: the default is $VCS_HOME/bin/*resource_type/resource_type*Agent.
- **On Windows NT**: the default is VCS_HOME\bin*VCSAgDriver.exe*.

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.

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 parameters 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.

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

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 parameter 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.

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 parameter. When a monitor fails as many times as the value specified by this parameter, 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 parameter.

When FaultOnMonitorTimeouts is set to 0, monitor failures are not considered indicative of a resource fault. Default is 4.

Note This parameter applies only to online resources. If a resource is offline, no special action is taken during monitor failures. Also, VCS on Windows NT waits for the entry points to run to completion; therefore, FaultOnMonitorTimeouts is not useful on Windows NT.

LogLevel

Specifies the type of agent framework and entry point messages to be written to the agent log file (local to the system):

- **On UNIX:** $VCS_LOG/log/*resource_type*_A.log
- **On Windows NT:** VCS_HOME\log*resource_type*_A.txt

Value	Description
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 an ONLINE or transitioning 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.

NumThreads

Number of threads used within the agent process for managing resources. This number does not include the three threads used for other internal purposes. Default is 10 threads.

OfflineMonitorInterval

Duration (in seconds) between two consecutive monitor calls for an OFFLINE resource. If set to 0, OFFLINE resources are not monitored. Default is 300 seconds.

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

OfflineTimeout

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

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 (see "FaultOnMonitorTimeouts" on page 63 and "ToleranceLimit" on page 66). A non-zero RestartLimit causes VCS to invoke the `online` entry point instead of failing over the service group to another system. VCS attempts to restart the resource according to the number set in RestartLimit before it gives up and fails over. However, if the resource remains online for the interval designated in ConfInterval, earlier attempts to restart are not counted against RestartLimit. Default is 0.

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

RegList

Keylist of parameter names. All resources are automatically registered for change notification for specified parameters. Default is empty list.

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.

Scheduling Class and Priority Configuration Support

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

- RealTime (specified as "RT" in the configuration file)
- TimeSharing (specified as "TS" in the configuration file)

Additional Information for Windows NT Users

On Windows NT, RT is mapped to HIGH_PRIORITY_CLASS and TS is mapped to NORMAL_PRIORITY_CLASS.

Priority Ranges

The following table displays platform-specific priority range for RealTime and TimeSharing processes.

Platform	Scheduling Class	Default Priority Range Weak / Strong	Priority Range Using #ps Commands
Solaris	RT TS	0 / 59 -60 / 60	100 / 159 N / A **Note** On Solaris, use `#ps -ae 0 pri, args`
HP-UX	RT TS	127 / 0 N / A	127 / 0 N / A **Note** On HP-UX, use `#ps -ael`

Default Scheduling Classes and Priorities

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

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

Parameters for Scheduling Class and Priorities

AgentClass

Indicates the scheduling class for the VCS agent process. Default is "TS".

AgentPriority

Indicates the priority in which the agent process runs. Default is "" (empty string).

ScriptClass

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

ScriptPriority

Indicates the priority of the script processes created by the agent. Default is "".

Initializing Parameters in the Configuration File

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

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

Setting Parameters Dynamically from the Command Line

▼ To update the AgentClass

Type:

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

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

```
hatype -modify FileOnOff AgentClass "RT"
```

▼ To update the AgentPriority

Type:

```
hatype -modify resource_type AgentPriority value
```

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

```
hatype -modify FileOnOff AgentPriority "10"
```

▼ To update the ScriptClass

Type:

```
hatype -modify resource_type ScriptClass value
```

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

```
hatype -modify FileOnOff ScriptClass "RT"
```

▼ To update the ScriptPriority

Type:

```
hatype -modify resource_type ScriptPriority value
```

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

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

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

Testing VCS Agents 6

VCS agents can be tested using the VCS engine or 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`. Debug messages are logged to a specific file.

- **On UNIX** debug messages are logged to $VCS_LOG/log/*resource_type*_A.log.
- **On Windows NT** debug messages are logged to VCS_HOME\log*resource_type*_A.txt

Complete the following requirements before testing an agent:

- ✔ Build the agent binary (UNIX) or DLL (Windows NT) and place it in the proper directory.
  - **On UNIX**: $VCS_HOME/bin/*resource_type*.
  - **On Windows NT**: VCS_HOME\bin*resource_type*.
- ✔ Install script entry points in the proper directory.
  - **On UNIX**: $VCS_HOME/bin/*resource_type*.
  - **On Windows NT**: VCS_HOME\bin*resource_type*.
- ✔ If you are using the VCS engine process, define the resource type in types.cf, define the resources in main.cf, and restart the engine. You may also define the new type and resources using commands listed in the *VERITAS Cluster Server User's Guide*.

Using the VCS Engine Process

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

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

`haagent -display` *resource_type*

For example, to test the Oracle agent, type:

```
haagent -display Oracle
```

If the Oracle agent is running, the output resembles:

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

Test Commands

To activate agent debug messages, type:

`hatype -modify` *resource_type* **`LogLevel info`**

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, type:

`hatype -modify` *resource_type* **`LogLevel error`**

Using AgentServer

The AgentServer utility enables you to test agents independent of the VCS engine process. It is part of the VCS package and is installed under the directory $VCS_HOME/bin on UNIX, or the VCS_HOME\bin on Windows NT. Instructions for using this utility begin on the next page.

To Access Help

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

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

   a. Add a type:

```
>addtype FileOnOff FileOnOff
```

   b. Add attributes of the type:

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

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

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

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

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


e. Add a resource:

```
>addres FileOnOff file1 FileOnOff
```

f. Set the resource attributes:

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

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

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

b. To take a resource offline:

```
>offlineres FileOnOff file1
```

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

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
   ```


▼ To test the FileOnOff agent on Windows NT

1. Type the following command to start AgentServer:

   ```
   C:\> VCS_HOME\bin\AgentServer
   ```

   Note that VCS_HOME should be substituted by the installation directory. 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 C:\WINNT\system32\drivers\etc\services. If `vcstest` is not specified, AgentServer uses the default value 14142.

2. Start the agent for the resource type:

   ```
   >startagent FileOnOff VCS_HOME\bin\VCSAgDriver.exe
   ```

   Note here also that VCS_HOME should be substituted by the installation directory. You will receive the following messages:

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

3. Review the sample configuration:

   ```
   types.cf:
      type FileOnOff (
            str PathName
            static str ArgList[] = { PathName }
            NameRule = resource.PathName
      )
   main.cf:
      ...
      group ga (
      ...
      )
      FileOnOff file1 (
         Enabled = 1
         PathName = "C:\\VRTSvcsfile001"
      )
   ```

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

   a. Add a type:

      ```
      >addtype FileOnOff FileOnOff
      ```

   b. Add attributes of the type:

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

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

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

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

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

   e. Add a resource:

      ```
      >addres FileOnOff file1 FileOnOff
      ```

   f. Set the resource attributes:

      ```
      >modifyres FileOnOff file1 PathName str
            C:\\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 `online` entry point of the FileOnOff agent. The following message is displayed when file1 is brought online:

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

b. To take a resource offline:

```
>offlineres FileOnOff file1
```

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

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
```

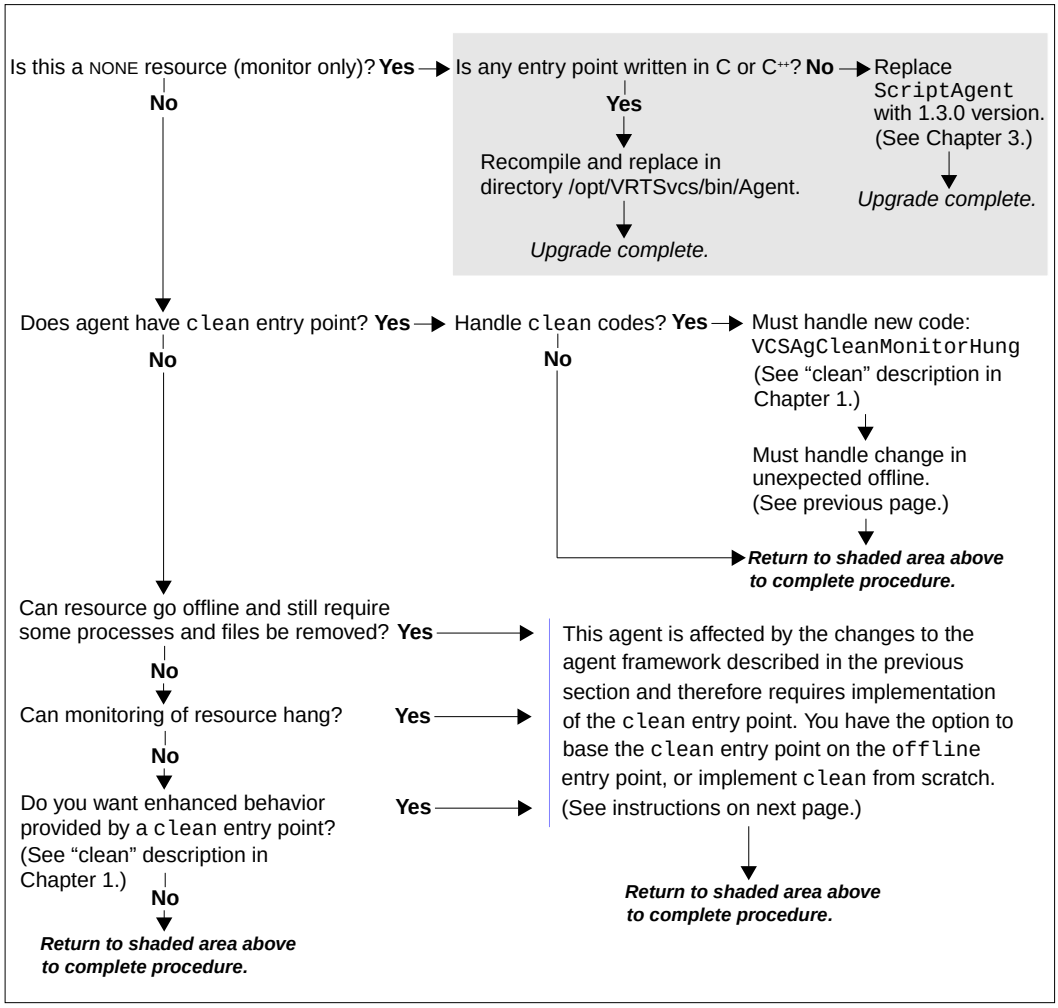
Upgrading Custom Agents

In VCS 1.2, a change was introduced to the agent framework that affects all custom agents. Specifically, when an agent reports an unexpected offline status, the agent framework now calls the `clean` entry point instead of the `offline` entry point. Therefore, if custom agents are to clean up following a resource fault, the agent must implement the `clean` entry point. Custom agents created prior to VCS 1.2. must be upgraded to accommodate this change, as described on the following page.

When a resource becomes offline unexpectedly, some agents may want to take action prior to VCS marking the resource as FAULTED. For example if a resource represents a collection of processes, and if the Monitor resource reports offline because one or more processes exited abruptly, the agent may determine to bring down the remaining processes before faulting the resource.

Some agents may not require any fault cleanup code, and are unaffected by this change in the agent framework. For example, the FileOnOff agent reports ONLINE if a given file exists, and OFFLINE if it does not. If a FileOnOff resource becomes offline unexpectedly, no additional cleanup is required.

The following information describes the procedures for upgrading custom agents built prior to VCS 1.2.



Sample clean Entry Point

The following samples illustrate how to implement the `clean` entry point for the FileOnOff resource. The differences between `offline` and `clean` entry points are also explained.

The meaning of the return value (or exit code in the case of scripts) of the `clean` entry point is different from that of the `offline` entry point. `Offline` entry points typically return 0, which indicates that the resource can be monitored immediately, however this return value can be any number. `Clean` entry points can return 0 (indicating success) or 1 (indicating failure). If `clean` returns 1, it may be called again after a monitor interval.

Also, compared with `offline`, the `clean` entry point receives an additional argument indicating the reason `clean` was called. This argument is positioned directly after the resource name argument. (See page 7 for more information.)

Using C++

Assume the following offline function:

```
/*
 * Offline entry point for FileOnOff resources. Removes the file
 * associated with the FileOnOff resource.
 *
 * The first ArgList attribute is supposed to be the pathname of
 * of the FileOnOff resource. See VCS Bundled Agents Guide for
 * more info on FileOnOff resources.
 */
unsigned int res_offline(const char *res_name, void **attr_val) {
   if (attr_val) {
      const char *path_name = (const char *)attr_val[0];
      remove(path_name);
   }
   return 0;
}
```

The clean entry point can be implemented as:

```
/*
 * Clean entry point for FileOnOff resources. Removes the file
 * associated with the FileOnOff resource.
 *
 * The first ArgList attribute is supposed to be the pathname of
 * of the FileOnOff resource. See VCS Bundled Agents Guide for
 * more info on FileOnOff resources.
 */
```


```
unsigned int res_clean(const char *res_name, VCSAgWhyClean reason,
      void **attr_val) {
   unsigned int rval = 1;// Failure
   if (attr_val) {
      const char *path_name = (const char *)attr_val[0];
      if ((remove(path_name) == 0) || (errno == ENOENT)) {
         rval = 0;// Successfully deleted file.
      }
   }
   return rval;
}
```

Using Shell Script

Assume the following offline script:

```
#!/bin/sh
#
# Offline entry point for FileOnOff resources. Removes the file
# associated with the FileOnOff resource.
#
# The first ArgList attribute is supposed to be the pathname of
# of the FileOnOff resource. See VCS Bundled Agents Guide for
# more info on FileOnOff resources.
#
# Note that $1 is resource name and $2 is the first ArgList attribute.
#
rm -f $2
exit 0
```

The clean entry point can be implemented as:

```
#!/bin/sh
#
# Clean entry point for FileOnOff resources. Removes the file
# associated with the FileOnOff resource.
#
# The first ArgList attribute is supposed to be the pathname of the
# FileOnOff resource. See VCS Bundled Agents Guide for more info on
# FileOnOff resources.
```

```
#
# Note that $1 is resource name, $2 is the reason code and $3 is the
# first ArgList attribute.
#
rm -f $3
if test -f $3
then exit 1;
else exit 0;
fi
```


Index