Please note that you can always find the most up-to-date technical documentation on our Web site at http://www.vmware.com/support/.

The VMware Web site also provides the latest product updates.
# Table of Contents

**Introduction to the C API Programming Guide 5**
- Organization of This Chapter 5
- About the C API 6
  - Compatibility 6
  - 32-bit Client Support 6
- Using the C API Documentation 7
  - The Document Set 7
  - Using the Programming Guide 7
- Installing the C API 8
  - Compiling a Client on a Windows Host with VMware Server Installed 8
  - Compiling a Client on a Linux Host with VMware Server Installed 8
  - Installing Client Files on Windows 9
  - Installing Client Files on Linux 9

**C API Programming Concepts 11**
- Objects and Handles 12
- Reference Counting 13
  - Handle Independence 13
  - Deleting Handles 14
  - Opening Handles 14
  - Internal Handle References 15
- Error Codes 16
  - Error Code Functions 16
  - Error Code Bit Masking 17
- Multithreading 18
  - Thread Safety of Handles 18
  - Performance Implications 18
- Handle Properties 19
  - The GetProperties() Function 19
- Property Lists 21
- Asynchronous Operations and Job Handles 22
  - Signaling a Job Handle 22
  - Using the Job Object in a Blocking Fashion 23
  - Retrieving Results from Job Object Properties 23
Callback Functions 25
  Using a Callback Function 25
Callback Events 27
Event Pump 28
Local and Remote Host Handles 29

Using the C API 31
Connecting to a Host 32
  Connecting to a Specified Host 32
  Connecting to the Local Host As a Specified User 33
  Connecting to the Local Host As the Current User 34
Registering and Unregistering Virtual Machines 36
  Registering a Virtual Machine 36
  Unregistering Virtual Machines 36
Getting a Handle to a Virtual Machine 38
Starting Or Resuming a Virtual Machine 39
  Starting a Virtual Machine from a Powered-Off State 39
  Resuming a Suspended Virtual Machine 40
Installing VMware Tools in a Virtual Machine 42
Virtual Machine Guest Operations 44
  VMware Tools 44
  Authentication 44
Powering Off or Suspending a Virtual Machine 46
  Choosing Whether To Power Off or Suspend 46
  Powering Off a Virtual Machine 46
  Suspending a Virtual Machine 47

Glossary 49

Revision History 53
Introduction to the C API Programming Guide

The C API Programming Guide describes an API that allows users to automate virtual machine operations on VMware Server. This API does not apply to any other VMware products at this time.

Organization of This Chapter
The following topics are covered in this chapter:

- About the C API on page 6 describes the purpose for the C API.
- Using the C API Documentation on page 7 describes the C API document set.
- Installing the C API on page 8 explains the basic installation requirements for creating and running C API clients.
About the C API

The C API (known as “Vix”) is an API that lets customers write scripts and programs to manipulate virtual machines. It is high-level, easy to use, and practical for both scripters and application programmers. The C API is designed for three kinds of customers:

- Technically adventurous customers. Often such a customer is a corporation with dedicated IT personnel that build their own in-house tools.
- Partners. These are typically software tools vendors that may use this C API to better integrate VMware products with their own products or to build management products specifically for virtual machines.
- VMware products. VMware uses the C API in its own products as a general abstraction layer between core virtual machine processes and associated service processes.

The C API runs on Windows and Linux. This release supports clients written in C.

Compatibility

This release of the C API is compatible with the following versions of VMware products:

Table 1-1 C API Compatibility

<table>
<thead>
<tr>
<th>VMware Product</th>
<th>Compatible Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>VMware Server</td>
<td>1.0</td>
</tr>
</tbody>
</table>

32-bit Client Support

The C API has only 32-bit libraries in this release. You cannot compile 64-bit programs to use the Server API.
Using the C API Documentation

The Document Set
The following documents describe how to use the C API to create clients that manage virtual machines and hosts:

- C API Reference Guide
- C API Programming Guide (this book)

Using the Programming Guide
This section explains how to use this programming guide. This programming guide describes the API in the following chapters:

- Introduction to the C API Programming Guide on page 5 describes the purpose, documentation, and installation of the C API.
- C API Programming Concepts on page 11 explains the fundamental concepts necessary for using the C API.
- Using the C API on page 31 describes how to perform a number of tasks using the C API.
- Glossary on page 49 lists some VMware terms commonly used in describing the C API.
Installing the C API

Vix is easy to install. Vix includes components both on the client machine and the server machine:

- You do not need to install any C API software components on the host machine. The server-side components of the C API are already present in VMware Server. When you installed VMware Server, the server components for the API were installed automatically.
- To use the C API on a client machine, you need the header files `vix.h` and `vm_basic_types.h`, as well as one or more library files. These files are present on the VMware Server host machine. You can also copy these files to another client machine and use them as described in the section Installing Client Files on Windows or Installing Client Files on Linux below.

Compiling a Client on a Windows Host with VMware Server Installed

To compile your client code with VMware Server installed:

1. Add the header file to an #include statement in your source code:

```c
#include "vix.h"
```

... client code here ...

2. Set the include path in your development environment to include:

```text
C:\Program Files\VMware\VMware VIX
```

3. Compile your client code to link in `vix.lib` statically.

4. Set your runtime environment to include DLLs from:

```text
C:\Program Files\VMware\VMware VIX
```

Compiling a Client on a Linux Host with VMware Server Installed

To compile your client code with VMware Server installed:

1. Add the header file to an #include statement in your source code:

```c
#include "vix.h"
```

... client code here ...

2. Compile your client code to link in `libvix.so` statically:

```bash
 gcc -I/usr/include/vmware-vix -o myprogram main.c /usr/lib/vmware-vix/libvix.so
```

3. Run your client program from any directory on the client machine.
Installing Client Files on Windows
To compile your client code, you need the following files:

- vix.h
- vm_basic_types.h
- vix.lib
- vix.dll
- ssleay32.dll
- libeay32.dll

These files are available on the host machine where VMware Server is installed. By default, you can find them in the directory:

C:\Program Files\VMware\VMware VIX

To compile and run a Vix client:
1. Copy the files from the VMware Server host machine to a directory on your client machine.
2. Add the header file to an #include statement in your source code:

   #include "vix.h"

   ... client code here ...

3. Set the include path in your development environment to include the current directory.
4. Compile your client code to link in vix.lib statically.
5. Run your client program from the directory containing the DLL files.
6. (optional) Install the three DLL files in your system directory (C:\WINNT). This allows you to run the client program from any directory on the client machine.

Installing Client Files on Linux
To compile your client code, you need the files vix.h, vm_basic_types.h, and libvix.so. These files are available on the host machine where VMware Server is installed. The default locations for these files on the host machine are:

- /usr/include/vmware-vix/vix.h
- /usr/include/vmware-vix/vm_basic_types.h
- /usr/lib/vmware-vix/libvix.so

If you do not find the files in the default locations, contact your system administrator.

To compile and run a Vix client:
1. Copy the files from the VMware Server host machine to a directory on your client machine.
2. Add the header file to an #include statement in your source code:
   
   ```c
   #include "vix.h"
   ...
   ...
   ```

3. Compile your client code to link in `libvix.so`:
   
   ```bash
   gcc -o myprogram main.c /usr/lib/libvix.so
   ```

4. Run your client program from any directory on the client machine.
C API Programming Concepts

This chapter contains the following topics:

- Objects and Handles on page 12
- Reference Counting on page 13
- Error Codes on page 16
- Multithreading on page 18
- Handle Properties on page 19
- Property Lists on page 21
- Asynchronous Operations and Job Handles on page 22
- Callback Functions on page 25
- Event Pump on page 28
- Local and Remote Host Handles on page 29
Objects and Handles

The Vix API is object-based. The API defines several types of objects and functions that operate on those objects.

Client applications reference Vix objects with handles. Handles are opaque identifiers (actually integers) that can be passed as references. Handles are run-time only and are unique only within a client's address space.

Most functions in the C-language API take a handle as a parameter. Because a handle value represents an object to the API, this document often uses the terms "handle" and "object" interchangeably.

There are several handle types, but a few of the key types are:

- Virtual Machine — A single virtual machine, which may or may not be powered on.
- Host — A single host computer, either the local host or a remote host.
- Job — An object used in managing asynchronous operations.
- Snapshot — A single snapshot of a virtual machine.
Reference Counting

Handles are reference counted, so you must call a “release” function on the handle when you are
done with it. This means that a handle will remain valid until you call the \texttt{Vix\_ReleaseHandle()} function. The \texttt{Vix\_ReleaseHandle()} function releases any type of handle:

\begin{verbatim}
void Vix_ReleaseHandle(VixHandle handle);
\end{verbatim}

For example, consider the following code to open and release a handle:

\begin{verbatim}
VixHandle handle1;
handle1 = MyOpenVMWrapper(... various parameters...);
// handle1 is assigned a unique integer value.

// Now you can perform various operations on handle1.

Vix_ReleaseHandle(handle1);
// handle1 has been released and should no longer be used.
\end{verbatim}

Handle Independence

Generally, you can have any number of handles active at one time. Each handle represents a different
object, and handles can be created and destroyed independently. Consider the following example:

\begin{verbatim}
VixHandle handle1;
VixHandle handle2;

handle1 = MyOpenVMWrapper(... parameters for virtual machine 1...);
// handle1 is assigned a unique value -- for example, 11.

handle2 = MyOpenVMWrapper(... parameters for virtual machine 2...);
// handle2 is assigned a unique value -- for example, 12.

// Now you can perform various operations on handle1 or handle2.

Vix_ReleaseHandle(handle1);
// handle1 has been released and should no longer be used.

// You can still perform operations on handle 2.

Vix_ReleaseHandle(handle2);
// handle2 has been released and should no longer be used.
\end{verbatim}
Deleting Handles

Clients must still call the "release" function even when the data stored in that handle has been deleted by another function. For example, to delete a virtual machine, you first call a "delete" function on the virtual machine's handle, and then you call the "release" function on the handle itself. This was done to avoid confusion over which functions do and do not release handles; only the "release" function can release a handle.

The client application is responsible for calling the "release" function on every handle. In the case of releasing a handle after deleting a virtual machine, the "delete" function updates the internal handle state, so most functions except the "release" function will recognize that the virtual machine has been deleted and immediately return an error.

Opening Handles

If you open a handle to the same virtual machine twice, without releasing the first handle, then the "open" function will return the same handle both times. The handle reference count has been incremented each time the "open" function returned, so you must call the "release" function twice — once for each time you opened the handle. Consider the following example:

```c
VixHandle handle1;
VixHandle handle2;

handle1 = MyOpenVMWrapper(...various parameters...);
// handle1 is assigned a unique value.
// handle1 has reference count 1.

// Now you can perform various operations on handle1.

handle2 = MyOpenVMWrapper(...the SAME parameters identifying the SAME virtual machine...);
// handle2 has the same value as handle1.
// handle1 has reference count 2.
ASSERT(handle1 == handle2);
Vix_ReleaseHandle(handle1);
// handle1 has reference count 1.

// The handle still can be safely used here.

Vix_ReleaseHandle(handle1);
// handle1 has been released and should no longer be used.
```

In general, every function that returns a handle increments the handle's reference count. This means a client application should call the release function once for every time the handle has been returned by a Vix function.
**Internal Handle References**

Some handles keep an internal reference to another handle. For example, a device keeps a reference to its virtual machine, and a virtual machine keeps references to all of its devices. As a result of these internal references, some handles may not be deleted when the client is done with them. This should not impact the client, since the internal reference counting is always correctly maintained. When an internal reference keeps a handle open, the client receive the same handle value when it opens the same object again after releasing its handle.

Host objects are an exception to this rule. Since disconnecting the host unloads the entire Vix server state, you should not call `Vix_ReleaseHandle()` on any handle after you have called `VixHost_Disconnect()`.
Error Codes

All synchronous Vix functions return an error code, which is an integer value defined by a global type. Asynchronous Vix functions (described below) report an error code when the function eventually completes.

Some Vix functions may return an internally defined error code value, such as an integer value that is not part of the public type. This is unusual; it indicates an internal error that cannot be translated to a public error code.

The error code type is `VixError`. It is defined in the public Vix header. Error codes are listed in the reference guide.

A Vix error is a 64-bit value. A value of `VIX_OK` indicates success, but if there is an error then several bit regions in the 64-bit value may be set. The least significant 16-bits will be set to the error code described for Foundry errors. More significant bit fields may be set to other values.

Error Code Functions

Vx provides the following defines for working with error codes:

- `VIX_ERROR_CODE(err)`
  Use this to mask off bit fields not used by the Vix API.

```c
VixError err;
err = VixJob_GetError(jobHandle);
if (VIX_E_FILE_NOT_FOUND == VIX_ERROR_CODE(err)) {
  // Handle error case...
}
```

- `VIX_SUCCEEDED(err)`
  Use this to test for the absence of an error.

```c
VixError err;
err = VixJob_GetError(jobHandle);
if (VIX_SUCCEEDED(err)) {
  // Handle success case...
}
```

- `VIX_FAILED(err)`
  Use this to test for the presence of an error.

```c
VixError err;
err = VixJob_GetError(jobHandle);
if (VIX_FAILED(err)) {
```
// Handle failure case...
}

**Error Code Bit Masking**

If you prefer to do your own bit masking on error codes, here are some examples:

```c
VixError err;

err = VixJob_GetError(jobHandle);

// CORRECT!
// This is legal. Success is always indicated with VIX_OK.
if (VIX_OK == err) {
    // Handle success case...
}

// CORRECT!
// This is legal. Success is always indicated with VIX_OK.
// Anything else is an error.
if (VIX_OK != err) {
    // Handle error case...
}

// WRONG!
// If an error code is not VIX_OK, then several bit fields may be set.
if (VIX_E_FILE_NOT_FOUND == err) {
    // This will not work...
}

// CORRECT!
// If an error code is not VIX_OK, then the least significant two bytes
// will be the error.
if (VIX_E_FILE_NOT_FOUND == (err & 0x000000FF)) {
    // Handle error case...
}
```
Multithreading

The Vix library is intended for use by multithreaded clients. Vix shared objects are managed by the Vix library to avoid conflicts between threads. Clients need only be responsible for protecting user-defined shared data.

Thread Safety of Handles

All Vix objects are thread safe so they may be used from several threads at the same time.

Vix objects are not directly modified by client code. The value of a handle identifies the object when passed to a function that works with the object. Certain functions cause object properties to be modified during execution. For example, the function `VixVM_PowerOn()` modifies the `VIX_PROPERTY_VM_IS_RUNNING` of the virtual machine handle.

The Vix library handles locking of objects when they are modified. As a result, client code does not need to be concerned with the protection of objects when they are used from multiple threads.

Performance Implications

Locking shared objects implies some performance degradation in certain situations. For example, when you power on a virtual machine, the Vix library needs to modify both the virtual machine handle and the host’s list of running virtual machines. If two clients power on two virtual machines at the same time, the Vix library needs to lock the list of running virtual machines on behalf of one client, causing a small delay for the second client.
Handle Properties

The Vix API defines a set of properties for each type of handle. A property is a typed name/value pair; a type name is a unique integer ID and the type may be one of the following:

- 32-bit Integer
- 64-bit Integer
- String
- Boolean
- Handle

Vix defines a different set of properties for each handle type. For example, a virtual machine handle has a string property that stores the file pathname of the virtual machine, but a job handle does not. If a property is defined for a particular handle type, however, then all handles of that type will have some value for that property all the time. For example, every virtual machine has a property that stores its file pathname, whether the virtual machine is powered on or not, or whether it is stored on the local host or not. The complete set of handle properties is defined in the reference guide.

Properties are the main mechanism for reading and writing both the persistent configuration state and the runtime status of handles. Properties allow Vix to be language independent and backward compatible.

Any property that is defined for a handle may be read at any time, but only some properties may be written. For example, properties such as the power state of a virtual machine are read-only. To change the power state of a virtual machine, you call functions such as VixVM_PowerOn() rather than set property values.

The GetProperties() Function

Vix provides one function that can get properties from any handle. This function has a varargs signature, which means you can use it to retrieve any number of properties by passing in sufficient parameters. The argument list must be terminated with a special property, VIX_PROPERTY_NONE.

```c
VixError Vix_GetProperties(VixHandle handle,
                                int firstPropertyID,
                                ...);
```

Here is an example of retrieving properties from a virtual machine handle:

```c
VixError err;
VixHandle handle1;
int vmPowerState;
char vmVmxPathName;
```
handle1 = MyOpenVMWrapper(...various parameters...);
// handle1 is assigned a unique value.
// handle1 has reference count 1.

err = Vix_GetProperties(handle1,
                        VIX_PROPERTY_VM_VMX_PATHNAME,
                        &vmVmxPathName,
                        VIX_VM_POWER_STATE_PROPERTY,
                        &vmPowerState,
                        VIX_PROPERTY_NONE);

Vix_ReleaseHandle(handle1);
// handle1 has been released and should no longer be used.
Property Lists

Vix defines a special runtime data structure, the property list, as a convenient way to store properties and pass them as arguments. Property lists are runtime-only data structures, and they behave as Vix objects: you can reference a property list with a handle and you can pass the handle to functions such as `Vix_GetProperties()`. Many Vix functions take a property list as a parameter, since this allows future versions of the function to take new optional parameters while remaining compatible with older code.

The set of properties you can retrieve from a handle depends on the handle type. For example, you cannot retrieve the virtual machine file pathname from a host handle. However, you can store any property in a property list that you pass as a function argument. The function ignores properties that are irrelevant.

Property lists are especially useful in asynchronous callback functions. Vix stores arguments for the callback function in a property list. Here is an example of a callback function that retrieves a value from a property list:

```c
void MyFunction(VixHandle propertyListHandle)
{
    char *url = NULL;

    err = Vix_GetProperties(propertyListHandle,
                            VIX_PROPERTY_VM_VMX_PATHNAME,
                            &url,
                            VIX_PROPERTY_NONE);

    if (VIX_OK != err) {
        ...Handle the error...
    }

    Vix_FreeBuffer(url);
}
```
Asynchronous Operations and Job Handles

Many Vix functions are synchronous, which makes them easy to use. Some functions, however, such as powering on a virtual machine, are asynchronous. These asynchronous functions either implement time-consuming operations or interact with the persistent virtual machine state, which may be on a remote host.

All asynchronous Vix functions allocate and return a "job" handle, which is a Vix object that represents the execution of an asynchronous function. A job handle may be used to signal when the asynchronous operation has completed, and also to retrieve the results of a completed asynchronous function. A job has a single "completed" state, which indicates when the job has finished. Additionally, a job may have several result values that are set when the job has completed. These different types of values are used differently: the completed state reports when the job is done and the result values are information returned by a completed job. Different kinds of jobs have different return values.

A new job handle is created for each active asynchronous call. For example, if you invoke three asynchronous calls then they all may complete at different times and they each have a different job handle. Vix always creates a job handle for every asynchronous call; even if you do not use it you are still responsible for releasing it.

Signaling a Job Handle

Currently, a client application may use several mechanisms to detect when a job handle has been signaled:

- Poll the job, by calling a non-blocking function that checks whether it is signaled.
- Block until the job is signaled, by calling the VixJob_Wait() function.
- Register a procedure that is called when the job is signaled.

A typical asynchronous call looks similar to VixVM_Open(), and there are several common patterns shared by all asynchronous Vix calls.

```
VixHandle VixVM_Open(VixHandle hostHandle,
                    const char *vmxFilePathName,
                    VixEventProc *callbackProc,
                    void *clientData);
```

This function creates a Vix job object and returns a VixHandle for this new job object. The caller is responsible for releasing this job object, even if the job object is not used for anything else.
Using the Job Object in a Blocking Fashion

The job object allows a client to block until an asynchronous call completes. This effectively turns the asynchronous call into a synchronous call. Here is an example of how this can be done with `VixVM_Open()`.

```c
VixHandle hostHandle = VIX_INVALID_HANDLE;
VixHandle jobHandle = VIX_INVALID_HANDLE;
VixHandle vmHandle = VIX_INVALID_HANDLE;

jobHandle = VixHost_Connect(VIX_API_VERSION,
                           VIX_SERVICEPROVIDER_VMWARE_SERVER,
                           "myMachine.myCompany.com", // hostName
                           0, // hostPort
                           "myUserName", // username
                           "myPassword", // password
                           0, // options
                           VIX_INVALID_HANDLE, // propertyListHandle
                           NULL, // callbackProc
                           NULL); // clientData

err = VixJob_Wait(jobHandle,
                  VIX_PROPERTY_JOB_RESULT_HANDLE,
                  &hostHandle,
                  VIX_PROPERTY_NONE);
Vix_ReleaseHandle(jobHandle);
if (VIX_OK != err) {
    goto abort;
}

jobHandle = VixVM_Open(hostHandle,
                       "c:\Virtual Machines\vm1\win2000.vmx",
                       NULL, // callbackProc,
                       NULL); // clientData

err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
Vix_ReleaseHandle(jobHandle);

err = Vix_GetProperties(jobHandle,
                        VIX_PROPERTY_JOB_RESULT_HANDLE,
                        &vmHandle,
                        VIX_PROPERTY_NONE);
```

Retrieving Results from Job Object Properties

The job object can also be used to retrieve results from an asynchronous operation once the asynchronous operation has completed. You can get these properties by calling...
Vix_GetProperties() on the job handle. For convenience, you can also extract these properties with the VixJob_Wait() function.

VixHandle hostHandle = VIX_INVALID_HANDLE;
VixHandle jobHandle = VIX_INVALID_HANDLE;
VixHandle vmHandle = VIX_INVALID_HANDLE;

jobHandle = VixHost_Connect(VIX_API_VERSION,
    VIX_SERVICEPROVIDER_VMWARE_SERVER,
    "myMachine.myCompany.com", // hostName
    0, // hostPort
    "myUserName", // username
    "myPassword", // password
    0, // options
    VIX_INVALID_HANDLE, // propertyListHandle
    NULL, // callbackProc
    NULL); // clientData

er = VixJob_Wait(jobHandle,
    VIX_PROPERTY_JOB_RESULT_HANDLE,
    &hostHandle,
    VIX_PROPERTY_NONE);
Vix_ReleaseHandle(jobHandle);
if (VIX_OK != err) {
    goto abort;
}

jobHandle = VixVM_Open(hostHandle,
    "c:\Virtual Machines\vm1\win2000.vmx",
    NULL, // callbackProc,
    NULL); // clientData

er = VixJob_Wait(jobHandle,
    VIX_PROPERTY_JOB_RESULT_HANDLE,
    &vmHandle,
    VIX_PROPERTY_NONE);
Vix_ReleaseHandle(jobHandle);

For simplicity, most of the examples in this document use the simple approach of calling VixJob_Wait() and requesting the desired properties at the same time. For better performance and responsiveness, applications may want to use callback procedures.
Callback Functions

All asynchronous Vix functions, such as \texttt{VixVM\_Open()}, include a parameter for a callback procedure and a parameter that will be passed to the callback procedure. These parameters are optional, so a caller may pass \texttt{NULL} for either. The prototype of this callback procedure parameter is:

\begin{verbatim}
typedef void (VixEventProc)(VixHandle handle,
    VixEventType eventType,
    VixHandle moreEventInfo,
    void *clientData);
\end{verbatim}

Using a Callback Function

If the caller provides a callback procedure, then that procedure is registered with the job object and will be invoked when the job object is signaled. For example, if a caller passes a callback procedure to \texttt{VixVM\_Open()}, then that callback procedure will be invoked when the virtual machine has been opened. This could happen either before or after \texttt{VixVM\_Open()} returns. It also may happen on any thread.

This mechanism allows the Vix function to complete asynchronously, so the application should not call \texttt{VixJob\_Wait()} when using a callback function.

When a callback procedure is invoked, it is passed the \texttt{clientData} parameter that was passed in the original call to the asynchronous function. This allows a callback procedure to associate some context with an outstanding asynchronous call.

\begin{verbatim}
void myCallback(VixHandle jobHandle,
    VixEventType eventType,
    VixHandle moreEventInfo,
    void *clientData)
{
    VixError err;
    VixError asyncErr;
    VixHandle vmHandle = VIX_INVALID_HANDLE;

    /*
    * Ignore progress callbacks.
    */
    if (VIX_EVENTTYPE\_CALLBACK\_SIGNALED != eventType) {
        return;
    }

    err = Vix\_GetProperties(jobHandle,
        VIX\_PROPERTY\_JOB\_RESULT\_HANDLE,

    VIX\_PROPERTY\_JOB\_EXIT\_STATE,

    VIX\_PROPERTY\_JOB\_RESULT\_INFO,

    VIX\_PROPERTY\_JOB\_RESULT\_HANDLE);

    if (err) {
        asyncErr = Vix\_GetErrorString(err);
        printf("Error: \%s\n", asyncErr);
        Vix\_FreeString(asyncErr);
    }

    // Perform callback
    myCallbackProc(jobHandle, eventType,
        moreEventInfo, clientData);

    return;
}
\end{verbatim}
Vix_ReleaseHandle(jobHandle);

if (VIX_OK != asyncErr) {
    /*
     * The open failed.
     */
}
}

int main()
{
    VixHandle hostHandle = VIX_INVALID_HANDLE;
    VixHandle jobHandle = VIX_INVALID_HANDLE;
    VixHandle vmHandle = VIX_INVALID_HANDLE;
    char *contextData = "Hello, Vix";

    jobHandle = VixHost_Connect(VIX_API_VERSION,
                                VIX_SERVICEPROVIDER_VMWARE_SERVER,
                                "myMachine.myCompany.com", // hostName
                                0, // hostPort
                                "myUserName", // username
                                "myPassword", // password
                                0, // options
                                VIX_INVALID_HANDLE, // propertyListHandle
                                NULL, // callbackProc
                                NULL); // clientData

    err = VixJob_Wait(jobHandle,
                      VIX_PROPERTY_JOB_RESULT_HANDLE,
                      &hostHandle,
                      VIX_PROPERTY_NONE);
    Vix_ReleaseHandle(jobHandle);
    if (VIX_OK != err) {
        goto abort;
    }

    jobHandle = VixVM_Open(hostHandle,
                            "c:\\Virtual Machines\\vm1\\win2000.vmx",
                            myCallback,
                            contextData);

    /*
* Do something, like pump a message pump.
* Later, myCallback will be invoked on another thread.
*/
}

Callback Events

Note that a callback might be called several times, for several different reasons. For example, it might be called for periodic progress updates. The eventType parameter indicates why the callback is being called. The supported event types are:

- **VIX_EVENTTYPE_CALLBACK_SIGNALLED** — This event indicates that the asynchronous action has completed, whether successfully or not.
- **VIX_EVENTTYPE_JOB_PROGRESS** — This event may be passed several times to report progress on an asynchronous action.
- **VIX_EVENTTYPE_FIND_ITEM** — This event is used by VixHost_FindItems().
- **VIX_EVENTTYPE_HOST_INITIALIZED** — This event is used by VixHost_Connect().
It is also possible to manage event timing in the client using the ‘event pump’ mechanism. This mechanism allows the client to call `Vix_PumpEvents()` to poll for asynchronous events at a convenient time. This mechanism is useful when it is desirable to minimize interruptions of the client application’s main thread.

To use the event pump, two things are required:

- Open the host with a flag indicating that events are to be collected on a single user-defined thread rather than a pool of threads with callback functions. The flag is a value of `VIX_HOSTOPTION_USE_EVENT_PUMP` passed in the `options` parameter to the `VixHost_Connect()` function.

```c
VixHandle hostHandle = VIX_INVALID_HANDLE;
VixHandle jobHandle = VIX_INVALID_HANDLE;
VixError err;
jobHandle = VixHost_Connect(VIX_API_VERSION,
    VIX_SERVICEPROVIDER_VMWARE_SERVER,
    NULL, // hostName
    0, // hostPort
    NULL, // userName
    NULL, // password,
    VIX_HOSTOPTION_USE_EVENT_PUMP, // options
    VIX_INVALID_HANDLE, // propertyListHandle
    NULL, // callbackProc
    NULL); // clientData
err = VixJob_Wait(jobHandle,
    VIXPROPERTY_JOB_RESULT_HANDLE,
    &hostHandle,
    VIXPROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
```

- Poll for events at appropriate points in the main thread. This is done by calling `Vix_PumpEvents()` to request the next event.

```c
• Poll for events at appropriate points in the main thread. This is done by calling `Vix_PumpEvents()` to request the next event.
```
Local and Remote Host Handles

Vix defines a “host” object — identified by a handle — that allows access to virtual machines. Typically, a Vix application begins by getting a handle to a host object, and then passing that host handle to other functions. A host handle may represent either a local or remote machine.

Vix provides a single API function to open any host. To open an arbitrary host, the function requires a machine name and password authentication.

Once you have a host object, you will pass it as a parameter to several Vix functions, such as the function that opens a handle to a virtual machine. When the client completes all operations, it releases the host handle with a call to `VixHost_Disconect()` . All handles and all state associated with that host are erased by the call.

For more information on getting a host handle, refer to Connecting to a Host on page 32.
Using the C API

This chapter explains how to use the Server API for common virtual machine management tasks. The following tasks are described:

- Connecting to a Host on page 32
- Registering and Unregistering Virtual Machines on page 36
- Getting a Handle to a Virtual Machine on page 38
- Starting Or Resuming a Virtual Machine on page 39
- Installing VMware Tools in a Virtual Machine on page 42
- Virtual Machine Guest Operations on page 44
- Powering Off or Suspending a Virtual Machine on page 46
Connecting to a Host

To work with a virtual machine, you first have to connect to the host where the virtual machine is stored. The Server API allows you to connect in three different ways, using the `VixHost_Connect()` function:

- You can connect to an arbitrary host using the name of the host machine, along with a user name and password identifying a valid user account on the machine. Refer to Connecting to a Specified Host on page 32.

- You can connect as a specified user on the local host (the host on which your client runs) when you provide a NULL value in the `hostName` argument, but you do specify the user name and password of a valid use account on the local host. Refer to Connecting to the Local Host As a Specified User on page 33.

- You can connect to the local host as the current user by passing NULL to all three identifying parameters: `hostName`, `userName`, and `password`. Refer to Connecting to the Local Host As the Current User on page 34.

In all cases, you can specify the port number while allowing other function arguments to take default values. The port number defaults to 902, which is correct for most installations of VMware Server. Some installations may be configured to use a different port number, if port 902 is already in use on the host.

`VixHost_Connect()` is an asynchronous function, so the function call must be completed with a call-back function or a call to `VixJob_Wait()`.

Connecting to a Specified Host

This option allows you to access any networked host machine from a single client machine. You can also specify the name of the local host and connect in the same manner as with a remote host.

```c
#include "vix.h"

int main(int argc, char * argv[])
{
    VixHandle hostHandle = VIX_INVALID_HANDLE;
    VixHandle jobHandle = VIX_INVALID_HANDLE;
    VixError err;

    // Connect to specified host.
    jobHandle = VixHost_Connect(VIX_API_VERSION,
                                VIX_SERVICEPROVIDER_VMWARE_SERVER,
                                argv[1], // hostName
                                0, // hostPort
                                argv[2], // userName
                                argv[3], // password,
```
CHAPTER 3 Using the C API

Connecting to the Local Host As a Specified User

This option allows you to run your client on any host machine that has a given user account, or you can choose a user account when you run the script. For example, you could create a client script that a system administrator could use on any machine to shut down all virtual machines owned by a given user.

```c
#include "vix.h"

int main(int argc, char * argv[])
{
    VixHandle hostHandle = VIX_INVALID_HANDLE;
    VixHandle jobHandle = VIX_INVALID_HANDLE;
    VixError err;

    // Connect as specified user on local host.
    jobHandle = VixHost_Connect(VIX_API_VERSION,
                                VIX_SERVICEPROVIDER_VMWARE_SERVER,
                                NULL, // hostName
                                0, // hostPort
                                argv[1], // userName
                                argv[2], // password,
                                0, // options
                                VIX_INVALID_HANDLE, // propertyListHandle
                                NULL, // callbackProc
                                NULL); // clientData
    err = VixJob_Wait(jobHandle,
                      VIX_PROPERTY_JOB_RESULT_HANDLE,
                      &hostHandle,
                      VIX_PROPERTY_NONE);
    if (VIX_OK != err) {
        // Handle the error...
        goto abort;
    }
    Vix_ReleaseHandle(jobHandle);
    // Other code goes here...

    abort:
    Vix_ReleaseHandle(jobHandle);
    Vix_ReleaseHandle(hostHandle);
}
```

0, // options
VIX_INVALID_HANDLE, // propertyListHandle
NULL, // callbackProc
NULL); // clientData

err = VixJob_Wait(jobHandle,
                  VIX_PROPERTY_JOB_RESULT_HANDLE,
                  &hostHandle,
                  VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
Connecting to the Local Host As the Current User

This option maximizes portability. You can run the client on any host machine, without limiting yourself to a particular user account and without having to specify a user account at run time. However, you need to be sure that the current user has appropriate permissions to use one or more virtual machines on the local host.

```c
#include "vix.h"
int main(int argc, char * argv[]) {
    VixHandle hostHandle = VIX_INVALID_HANDLE;
    VixHandle jobHandle = VIX_INVALID_HANDLE;
    VixError err;

    // Connect as current user on local host.
    jobHandle = VixHost_Connect(VIX_API_VERSION,
                               VIX_SERVICEPROVIDER_VMWARE_SERVER,
                               NULL, // hostName
                               0, // hostPort
                               NULL, // userName
                               NULL, // password,
                               0, // options
                               VIX_INVALID_HANDLE, // propertyListHandle
                               NULL, // callbackProc
                               NULL); // clientData
    err = VixJob_Wait(jobHandle,
                      VIX_PROPERTY_JOB_RESULT_HANDLE,
                      &hostHandle,
                      VIX_PROPERTY_NONE);
    if (VIX_OK != err) {
        // Handle the error...
    }
    goto abort;
}
```
goto abort;
}
Vix_ReleaseHandle(jobHandle);
// Other code goes here...

abort:
    Vix_ReleaseHandle(jobHandle);
    Vix_ReleaseHandle(hostHandle);
Registering and Unregistering Virtual Machines

VMware Server requires that a virtual machine be present in its “registry” before using the virtual machine. The virtual machine registry is a list of virtual machines that the local host is able to run. Conversely, if you remove a virtual machine from VMware Server’s registry, it is not available for any kind of virtual machine operations. This is a convenient way to prevent virtual machines from being modified while they are not in regular use. You can also keep an unregistered virtual machine as a “gold” standard from which you can make copies as needed.

Registering or unregistering a virtual machine does not require a handle to the virtual machine, but it does require a path to the virtual machine’s configuration (.vmx) file. The client must also provide a handle to the host machine where the virtual machine is to be registered or unregistered.

VixHost_RegisterVM() and VixHost_UnregisterVM() are asynchronous functions, so the function call must be completed with a call-back function or a call to VixJob_Wait().

Registering a Virtual Machine

To register a virtual machine:

1. Connect to the host on which the virtual machine is located. Refer to Connecting to a Host on page 32.
2. Use the host handle with VixHost_RegisterVM() to register the virtual machine by its path name in the host’s file system.

   ```c
   VixHandle jobHandle = VIX_INVALID_HANDLE;
   char vmxFilePath[] = "c:\Virtual Machines\vm1\win2000.vmx";
   VixError err;
   // Add virtual machine to host’s inventory.
   jobHandle = VixHost_RegisterVM(hostHandle,
       vmxFilePath,
       NULL, // callbackProc,
       NULL); // clientData
   err = VixJob_Wait(jobHandle,
       VIX_PROPERTY_NONE);
   if (VIX_OK != err) {
       // Handle the error...
       goto abort;
   }
   Vix_ReleaseHandle(jobHandle);
   ```

Unregistering a virtual machine

To unregister a virtual machine:
1. Connect to the host on which the virtual machine is located. Refer to Connecting to a Host on page 32.

2. Make sure the virtual machine is powered off. Refer to Powering Off or Suspending a Virtual Machine on page 46.
   If the virtual machine is suspended, first resume it, then power it off. Refer to Starting Or Resuming a Virtual Machine on page 39.

3. Use the host handle with `VixHost_UnregisterVM()` to unregister the virtual machine by its path name in the host’s file system.

   ```c
   VixHandle jobHandle = VIX_INVALID_HANDLE;
   char vmxFilePath[] = "c:\\Virtual Machines\\vm1\\win2000.vmx";
   VixError err;

   // Remove virtual machine from host’s inventory.
   jobHandle = VixHost_UnregisterVM(hostHandle,
                                     vmxFilePath,
                                     NULL, // callbackProc,
                                     NULL); // clientData

   err = VixJob_Wait(jobHandle,
                     VIXPROPERTY_NONE);

   if (VIX_OK != err) {
     // Handle the error...
     goto abort;
   }

   Vix_ReleaseHandle(jobHandle);
   ```
Getting a Handle to a Virtual Machine

The majority of virtual machine operations require a handle to identify the virtual machine. The `VixVM_Open()` function converts a path name to a handle, which you can use in subsequent function calls. `VixVM_Open()` is an asynchronous function, so the function call must be completed with a call-back function or a call to `VixJob_Wait()`.

To get a handle to a virtual machine:

1. You need to supply a handle to the host on which the virtual machine is located. Refer to Connecting to a Host on page 32.
2. The virtual machine must be registered on the host. Refer to Registering and Unregistering Virtual Machines on page 36.
3. Use the host handle and the virtual machine's path name in the host's file system to open the virtual machine with `VixVM_Open()`.
4. Retrieve the virtual machine handle from the job object.

```c
VixHandle jobHandle = VIX_INVALID_HANDLE;
char vmxFilePath[] = "c:\Virtual Machines\vm1\win2000.vmx";
VixError err;

// Open virtual machine and get a handle.
jobHandle = VixVM_Open(hostHandle,
                        vmxFilePath,
                        NULL, // callbackProc
                        NULL); // clientData
err = VixJob_Wait(jobHandle,
                  VIX_PROPERTY_JOB_RESULT_HANDLE,
                  &vmHandle,
                  VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
Vix_ReleaseHandle(jobHandle);
// ...Use vmHandle in subsequent code...
```
Starting Or Resuming a Virtual Machine

The `VixVM_PowerOn()` function has two purposes:

- Start bootup of the guest operating system in a virtual machine previously in a powered-off state.
- Resume execution of a guest operating system in a suspended virtual machine.

There is no separate Resume function in the C API. The `VixVM_PowerOn()` function must be used instead.

`VixVM_PowerOn()` is an asynchronous function, so the function call must be completed with a call-back function or a call to `VixJob_Wait()`.

For most purposes — and especially if you want to do operations in the guest operating system — you need to wait for VMware Tools to become active in the guest. Until the VMware Tools utility is ready, no guest operations using the C API can complete. Refer to Installing VMware Tools in a Virtual Machine on page 42 for more information about VMware Tools.

The `VixVM_WaitForToolsInGuest()` function allows you to wait until the VMware Tools utility is responding. There is a timeout parameter available for the function. You should generally set the timeout somewhat longer than the typical time it takes the guest operating system to finish booting.

Starting a Virtual Machine from a Powered-Off State

To start a virtual machine:

1. Connect to the host on which the virtual machine is located. Refer to Connecting to a Host on page 32.
2. Get a handle to the virtual machine. Refer to Getting a Handle to a Virtual Machine on page 38.
3. Use the virtual machine handle in a call to `VixVM_PowerOn()` to start up the virtual machine.
4. Wait for VMware Tools to become active in the guest operating system, using a call to `VixVM_WaitForToolsInGuest()`. Check the job object to determine whether the wait timed out.

```c
VixError err = VIX_OK;
VixHandle jobHandle = VIX_INVALID_HANDLE;

// Start up virtual machine.
jobHandle = VixVM_PowerOn(vmHandle,
    0, // powerOnOptions,
    VIX_INVALID_HANDLE, // propertyListHandle,
    NULL, // callbackProc,
```
err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
Vix_ReleaseHandle(jobHandle);

// Wait for VMware Tools to be active.
jobHandle = VixVM_WaitForToolsInGuest(VixHandle vmHandle,
                                        120, // timeoutInSeconds,
                                        NULL, // callbackProc,
                                        NULL); // clientData

err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
Vix_ReleaseHandle(jobHandle);

### Resuming a Suspended Virtual Machine

To resume a suspended virtual machine:

1. Connect to the host on which the virtual machine is located. Refer to Connecting to a Host on page 32.
2. Get a handle to the virtual machine. Refer to Getting a Handle to a Virtual Machine on page 38.
3. Use the virtual machine handle in a call to VixVM_PowerOn() to start up the virtual machine.
4. Generally, VMware Tools was already active in the guest operating system when the virtual machine was suspended, so you may not need to use the VixVM_WaitForToolsInGuest() function.

VixError err = VIX_OK;
VixHandle jobHandle = VIX_INVALID_HANDLE;

// Resume suspended virtual machine.
jobHandle = VixVM_PowerOn(vmHandle,
                           0, // powerOnOptions,
                           VIX_INVALID_HANDLE, // propertyListHandle,
                           NULL, // callbackProc,
                           NULL); // clientData

err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
goto abort;
}
Vix_ReleaseHandle(jobHandle);
Installing VMware Tools in a Virtual Machine

VMware Tools is a suite of utilities and drivers that enhances the performance and functionality of your guest operating system. For instance, VMware Tools provides a mouse driver and an SVGA driver for the virtual machine.

VMware Tools also implements some of the functionality required for doing Vix functions that affect the guest operating system, such as \texttt{VixVM\_RunProgramInGuest()}. You need to install the VMware Tools before calling guest functions in the C API.

To begin the Tools installation process, you can call the Vix function \texttt{VixVM\_InstallToolsInGuest()}. This function mounts a CD image containing the VMware Tools installer. If the guest operating system has the autorun feature enabled, the installer will start automatically. Connect a console window to the virtual machine and use the mouse to navigate the installer wizard.

The \texttt{VixVM\_InstallToolsInGuest()} function is also useful for updating the version of VMware Tools installed in a virtual machine. When a newer version of the API is available, it’s a good idea to run the Tools installation on your virtual machines to make sure you are working with the very latest version.

\texttt{VixVM\_InstallToolsInGuest()} is an asynchronous function, so the function call must be completed with a call-back function or a call to \texttt{VixJob\_Wait()}. To install VMware Tools:

1. Connect to the host on which the virtual machine is located. Refer to Connecting to a Host on page 32.
2. Get a handle to the virtual machine. Refer to Getting a Handle to a Virtual Machine on page 38.
4. Use the virtual machine handle to call \texttt{VixVM\_InstallToolsInGuest()}.  
5. Using a console connected to the virtual machine, do one of the following:
   • For a Windows guest, use the VMware Tools installer as described in the VMware Server manual.
   • For a Linux guest, follow the instructions in the VMware Server manual to install the VMware Tools package.

\begin{verbatim}
VixError err = VIX_OK;
VixHandle jobHandle = VIX_INVALID_HANDLE;

jobHandle = VixVM\_InstallTools(vmHandle,
    0, // options

\end{verbatim}
NULL, // commandLineArgs
NULL, // callbackProc
NULL); // clientData
err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
Vix_ReleaseHandle(jobHandle);
Virtual Machine Guest Operations

VMware Tools
All functions that execute in a guest operating system require VMware Tools to be installed and running. Refer to Installing VMware Tools in a Virtual Machine on page 42 for information on installing VMware Tools.

Authentication
All functions that modify a file system in the guest operating system or that execute code in a guest operating system require the client to authenticate as a user account known to the guest. This includes the following functions:
- VixVM_RunProgramInGuest ()
- VixVM_CopyFileFromHostToGuest ()
- VixVM_CopyFileFromGuestToHost ()

A client uses VixVM_LoginInGuest () to authenticate with the guest operating system. Once authenticated, the client may continue to perform guest operations until one of the following takes place:
- The client terminates, which ends the session.
- The client fails another attempt to authenticate.
- The virtual machine shuts down.

If a virtual machine is suspended while a client is authenticated with the guest operating system, the client may resume guest operations after resuming virtual machine execution. However, this is only true as long as the client continues running while the virtual machine is suspended.

To authenticate with the guest operating system:
1. Connect to the host on which the virtual machine is located. Refer to Connecting to a Host on page 32.
2. Get a handle to the virtual machine. Refer to Getting a Handle to a Virtual Machine on page 38.
3. Power on the virtual machine and wait for VMware Tools to become active. Refer to Starting Or Resuming a Virtual Machine on page 39.
4. Use the virtual machine handle, the user name, and the user password, to call VixVM_LoginInGuest ()

VixError err = VIX_OK;
VixHandle jobHandle = VIX_INVALID_HANDLE;
// Authenticate with guest operating system.
jobHandle = VixVM_LoginInGuest(vmHandle,
    "Administrator", // userName
    "secret", // password
0, // options
NULL, // callbackProc
NULL); // clientData

err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
Vix_ReleaseHandle(jobHandle);
Powering Off or Suspending a Virtual Machine

Choosing Whether To Power Off or Suspend

There are two ways to stop virtual machine execution. Whether you choose to power off or to suspend a virtual machine depends on your purpose.

- **VixVM_PowerOff()** is equivalent to turning off the power switch of a physical machine. Some operations on a virtual machine, such as changing the virtual hardware configuration, require that the virtual machine be powered off.

- **VixVM_Suspend()** is similar to putting a laptop to sleep. The virtual machine stops running, but when you resume operation, the virtual machine continues exactly where it left off. The suspend operation also saves the virtual machine’s state on disk, allowing you to reboot the host or move the virtual machine to a new location without shutting down the guest operating system in the virtual machine.

Both functions are asynchronous, so either function call must be completed with a call-back function or a call to **VixJob_Wait()**.

Powering Off a Virtual Machine

To power off a virtual machine:

1. Connect to the host on which the virtual machine is located. Refer to Connecting to a Host on page 32.
2. Get a handle to the virtual machine. Refer to Getting a Handle to a Virtual Machine on page 38.
3. Use the virtual machine handle in a call to **VixVM_PowerOff()**.
4. Check the power state of the virtual machine.

```c
VixError err = VIX_OK;
VixHandle jobHandle = VIX_INVALID_HANDLE;

// Power off the virtual machine.
jobHandle = VixVM_PowerOff(vmHandle,
                          0, // powerOnOptions,
                          VIX_INVALID_HANDLE, // propertyListHandle,
                          NULL, // callbackProc,
                          NULL); // clientData

err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
Vix_ReleaseHandle(jobHandle);
```
// Verify the virtual machine is off.
VixPowerState powerState = 0;
while (VIX_POWERSTATEPowered_OFF != powerState) {
    err = Vix_GetProperties(vmHandle,
        VIXPROPERTY_VMPOWER_STATE,
        &powerState,
        VIXPROPERTY_NONE);
    if (VIX_OK != err) {
        // Handle the error...
        goto abort;
    }
    sleep(1);
}

Suspending a Virtual Machine

To suspend a virtual machine:

1. Connect to the host on which the virtual machine is located. Refer to Connecting to a Host on page 32.
2. Get a handle to the virtual machine. Refer to Getting a Handle to a Virtual Machine on page 38.
3. Use the virtual machine handle in a call to VixVM_Suspend().
4. Check the power state of the virtual machine.
VixError err = VIX_OK;
VixHandle jobHandle = VIX_INVALID_HANDLE;

// Suspend the virtual machine.
jobHandle = VixVM_Suspend(vmHandle,
    0, // powerOnOptions,
    VIX_INVALID_HANDLE, // propertyListHandle,
    NULL, // callbackProc,
    NULL); // clientData
err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
Vix_ReleaseHandle(jobHandle);

// Verify the virtual machine is suspended.
VixPowerState powerState = 0;
while (VIX_POWERSTATE_SUSPENDED != powerState) {
    err = Vix_GetProperties(vmHandle,
VIX_PROPERTY_VM_POWER_STATE,
   &powerState,
   VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
sleep(1);
This appendix defines some of the terms used in the C API.

**Configuration file**
See Virtual machine configuration.

**Console**
See VMware Virtual Machine Console.

**Guest operating system**
An operating system that runs inside a virtual machine.
See also Host operating system.

**Host computer**
The physical computer on which the VMware Server software is installed. It hosts the VMware Server virtual machines.

**Host operating system**
An operating system that runs on the host machine.
See also Guest operating system.
Legacy virtual machine
A virtual machine created for use in earlier versions of VMware software. You can use and create legacy virtual machines within VMware Server, but new features are not usable.

Linked clone
A copy of the original virtual machine that must have access to the parent virtual machine’s virtual disks. The linked clone stores changes to the virtual disks in a separate set of files.
See also Full clone.

Resume
Return a virtual machine to operation from its suspended state. When you resume a suspended virtual machine, all applications are in the same state they were when the virtual machine was suspended.
See also Suspend.

Revert to snapshot
Reverting to a snapshot restores the status of the active virtual machine to its immediate parent snapshot.

Snapshot
A snapshot preserves the virtual machine just as it was when you took the snapshot — the state of the data on all the virtual machine’s disks and whether the virtual machine was powered on, powered off, or suspended. The C API lets you take a snapshot of a virtual machine at any time and revert to that snapshot at any time. You can take a snapshot when a virtual machine is powered on, powered off, or suspended.

Suspend
Save the current state of a running virtual machine. To return a suspended virtual machine to operation, use the resume feature.
See also Resume.
To return a suspended virtual machine to operation, use the resume feature.

Virtual machine
A virtualized x86 PC environment in which a guest operating system and associated application software can run. Multiple virtual machines can operate on the same host system concurrently.

Virtual disk
A virtual disk is a file or set of files that appear as a physical disk drive to a guest operating system. These files can be on the host machine or on a remote file system.
**Virtual machine configuration**
The specification of what virtual devices (such as disks and memory) are present in a virtual machine and how they are mapped to host files and devices.

**Virtual machine configuration file**
A file containing a virtual machine configuration. It is created when you create the virtual machine. It is used by VMware Server to identify and run a specific virtual machine.

**VMware Tools**
A suite of utilities and drivers that enhances the performance and functionality of your guest operating system. Key features of VMware Tools include some or all of the following, depending on your guest operating system: an SVGA driver, a mouse driver, the VMware guest operating system service, the VMware Tools control panel, and support for such features as shrinking virtual disks, time synchronization with the host, VMware Tools scripts, and connecting and disconnecting devices while the virtual machine is running.

**VMware Tools service**
One of the components installed with VMware Tools that performs various duties in the guest operating system, such as executing commands in the virtual machine, gracefully shutting down and resetting a virtual machine, sending a heartbeat to VMware Server, synchronizing the time of the guest operating system with the host operating system, and passing strings from the host operating system to the guest operating system.
Revision History

The following table lists the revision history for the *VMware C API Programming Guide*.

<table>
<thead>
<tr>
<th>Date</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>May 1, 2006</td>
<td>Beta-3 draft.</td>
</tr>
<tr>
<td>May 31, 2006</td>
<td>RC1 draft: expanded chapter on Using the API. Fixed intro chapter.</td>
</tr>
</tbody>
</table>
Index

A
asynchronous operations 22, 25, 27

B
blocking function calls 23

C
callback events 27
callback functions 25, 27
configuration file 49
console 49

E
error code functions 16
error codes 16, 17
error masking 17
event pump 28

G
GetProperties() function 19
guest operating system 49

H
handle independence 13
handle properties 19, 23
handles 12, 14, 15, 18, 22
handles, deleting 14
host computer 49
host handles 29
host operating system 49

J
job object 23
job objects 22, 23

L
legacy virtual machine 50
linked clone 50

M
multithreading 18, 28
multithreading, performance 18

O
objects 12

P
property lists 21

R
reference counting 13, 15
resume 50
revert to snapshot 50

S
signaling jobs 22
snapshot 50
suspend 50

T
thread safety 18

V
virtual disk 50
virtual machine 50
virtual machine configuration 51
virtual machine configuration file 51
VMware Tools 51
VMware Tools Service 51