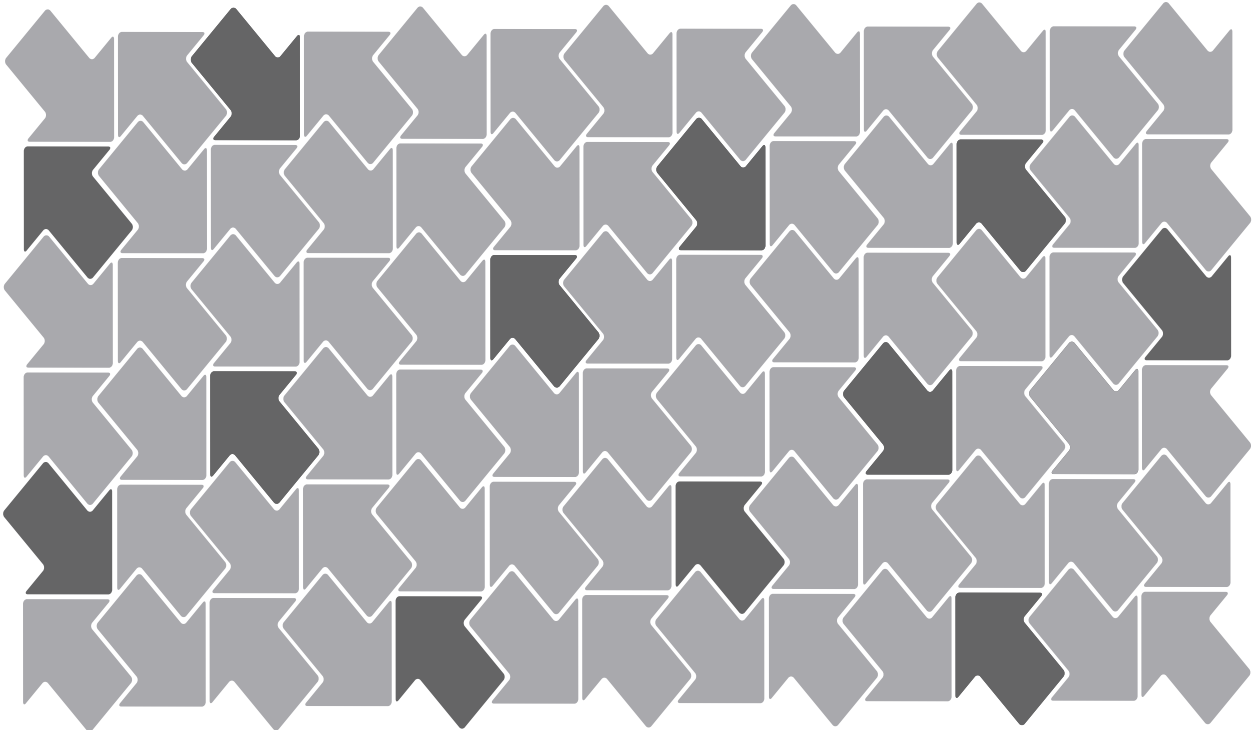


Programming API Programming Guide

Version 1.0



Programming API Programming Guide
Revision: 20060630
Item: SDK-ENG-Q206-183

You can find the most up-to-date technical documentation at:

<http://www.vmware.com/support/pubs>

The VMware Web site also provides the latest product updates.

If you have comments about this documentation, submit your feedback to:

docfeedback@vmware.com

© 2006 VMware, Inc. All rights reserved. Protected by one or more of U.S. Patent Nos. 6,397,242, 6,496,847, 6,704,925, 6,711,672, 6,725,289, 6,735,601, 6,785,886, 6,789,156, 6,795,966, 6,880,022, 6,961,941, 6,961,806 and 6,944,699; patents pending.

VMware, the VMware “boxes” logo and design, Virtual SMP and VMotion are registered trademarks or trademarks of VMware, Inc. in the United States and/or other jurisdictions. All other marks and names mentioned herein may be trademarks of their respective companies.

VMware, Inc.
3145 Porter Drive
Palo Alto, CA 94304
www.vmware.com

Contents

Chapter 1 Introduction to the Programming API Programming Guide	1
Organization of This Chapter	1
About the Programming API	1
Compatibility	2
32-Bit Client Support	2
Using the Programming API Documentation	2
The Document Set	2
Using the Programming Guide	2
Installing the Programming API	3
Compiling a Client on a Microsoft Windows Host with VMware Server Installed	3
Compiling a Client on a Linux Host with VMware Server Installed	4
Compiling a Client on a Microsoft Windows Host Without VMware Server Installed	4
Compiling a Client on a Linux Host Without VMware Server Installed	5
Chapter 2 Programming API Concepts	7
Objects and Handles	7
Reference Counting on Handles	8
Handle Independence	8
Deleting Handles	9
Opening Handles	9
Internal Handle References	10
Error Codes	10
Error Code Functions	10
Error Code Bit Masking	11
Multithreading	12
Thread Safety of Handles	12
Performance Implications	12
Handle Properties	12
GetProperties() Function	13

Property Lists	14
Asynchronous Operations and Job Objects	14
Signaling a Job Object	15
Polling the Job Object for Completion	15
Using the Job Object to Block Calls	16
Retrieving Results from Job Object Properties	17
Callback Functions	18
Using a Callback Function	18
Callback Events	20
Event Models	21
Multi-threaded Event Model	21
Single-threaded Event Model	21
Using the Event Pump	21
Polling for Completion in a Single-Threaded Client	22
Using a Callback Function in a Single-Threaded Client	24
Local and Remote Host Handles	26
Chapter 3 Using the Programming API	27
Connecting to a Host	27
Connecting to a Specified Host	28
Connecting to the Local Host as a Specified User	29
Connecting to the Local Host as the Current User	29
Registering and Unregistering Virtual Machines	30
Registering a Virtual Machine	31
Unregistering a Virtual Machine	31
Getting a Handle to a Virtual Machine	32
Starting or Resuming a Virtual Machine	33
Starting a Virtual Machine from a Powered-Off State	33
Resuming a Suspended Virtual Machine	34
Installing VMware Tools in a Virtual Machine	36
Virtual Machine Guest Operations	37
VMware Tools	37
Authentication	37
Powering Off or Suspending a Virtual Machine	38
Choosing Whether to Power Off or Suspend	38
Powering Off a Virtual Machine	39
Suspending a Virtual Machine	40
Importing a Legacy Virtual Machine	40

Upgrading Virtual Hardware	41
Glossary	43
Revision History	47
Index	49

CHAPTER 1 Introduction to the Programming API Programming Guide

The *Programming 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 Programming API”](#) on page 1 describes the purpose for the Programming API.
- [“Using the Programming API Documentation”](#) on page 2 describes the Programming API document set.
- [“Installing the Programming API”](#) on page 3 explains the basic installation requirements for creating and running Programming API clients.

About the Programming API

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

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

The Programming API runs on the Microsoft Windows and Linux platforms. This release supports clients written in C.

Compatibility

This release of the Programming API is compatible with VMware Server 1.0.

32-Bit Client Support

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

Using the Programming API Documentation

The Document Set

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

- Programming API Reference Guide
- Programming 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 Programming API Programming Guide”](#) on page 1 describes the purpose, documentation, and installation of the Programming API.
- [“Programming API Concepts”](#) on page 7 explains the fundamental concepts necessary for using the Programming API.
- [“Using the Programming API”](#) on page 27 describes how to perform tasks using the Programming API.
- [“Glossary”](#) on page 43 lists some VMware terms commonly used in describing the Programming API.

Installing the Programming API

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

- On the VMware Server host machine, you do not need to install any additional Programming API components. The server-side components of the Programming API are already present in VMware Server. When you installed VMware Server, the server components for the API were installed automatically.
- To use the Programming API on a client machine without VMware Server installed, 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 copy these files to a client machine and use them as described in the section [“Compiling a Client on a Microsoft Windows Host Without VMware Server Installed”](#) or [“Compiling a Client on a Linux Host Without VMware Server Installed”](#) below.

Compiling a Client on a Microsoft 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:

```
#include "vix.h"
... client code here ...
```
- 2 Set the include path in your development environment to include:

```
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:

```
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:

```
#include "vix.h"
... client code here ...
```

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

```
gcc -I/usr/include/vmware-vix -o myprogram main.c \
/usr/lib/libvmware-vix.so.0
```

- 3 Run your client program from any directory on the client machine.

Compiling a Client on a Microsoft Windows Host Without VMware Server Installed

To compile your client code on a Microsoft Windows machine without VMware Server installed, you need the following files:

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

These files are available from 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 directory containing the files you copied.

- 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.

Compiling a Client on a Linux Host Without VMware Server Installed

To compile your client code on a Linux machine without VMware Server installed, you need the files `vix.h`, `vm_basic_types.h`, and `libvix.so`. These files are available from 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/libvmware-vix.so.0`
- `/usr/lib/vmware/lib/libcrypto.so.0.9.7/libcrypto.so.0.9.7`
- `/usr/lib/vmware/lib/libssl.so.0.9.7/libssl.so.0.9.7`

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:

```
#include "vix.h"
... client code here ...
```
- 3 Compile your client code to link in `libvmware-vix.so`:

```
gcc -o myprogram main.c libvmware-vix.so.0
```
- 4 Run your client program from the directory containing the library files you copied.

CHAPTER 2 **Programming API Concepts**

This chapter contains the following topics:

- [“Objects and Handles”](#) on page 7
- [“Reference Counting on Handles”](#) on page 8
- [“Error Codes”](#) on page 10
- [“Multithreading”](#) on page 12
- [“Handle Properties”](#) on page 12
- [“Property Lists”](#) on page 14
- [“Asynchronous Operations and Job Objects”](#) on page 14
- [“Callback Functions”](#) on page 18
- [“Event Models”](#) on page 21
- [“Using the Event Pump”](#) on page 21
- [“Local and Remote Host Handles”](#) on page 26

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 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 might or might 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 on Handles

Handles are reference counted, so you must call a “release” function on the handle when you are done using it. A handle remains valid until you call the `Vix_ReleaseHandle()` function. The `Vix_ReleaseHandle()` function releases any type of handle:

```
void Vix_ReleaseHandle(VixHandle handle);
```

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

```
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.
```

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:

```
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.
```

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 design avoids 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, the open function will return the same handle both times. The handle reference count is incremented each time the open function returned, so you must call the release function twice, once for each time you open the handle. Consider the following example:

```
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 might not be deleted when the client is done with them. This situation should not impact the client, because the internal reference counting is always correctly maintained. When an internal reference keeps a handle open, the client receives the same handle value when it opens the same object again after releasing its handle.

Host objects are an exception to this rule. Because disconnecting the host unloads the entire Vix server state, do 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 can 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 *Programming API 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 might be set. The least significant 16 bits are set to the error code described for Vix errors. More significant bit fields might be set to other values.

Error Code Functions

Vix 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.

```
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.

```
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.

```
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:

```
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 (all zeroes).
// Anything else is an error.
if (VIX_OK != err) {
    // Handle error case...
}

// WRONG!
// If an error code is not VIX_OK, 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, the least significant 16 bits
// will be the Vix error code.
if (VIX_E_FILE_NOT_FOUND == (err & 0xFFFF)) {
```

```
// Handle error case...  
}
```

Multithreading

The Vix library is intended for use by multi-threaded 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 path name 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 always have some value for that property. For example, every virtual machine has a property that stores its file path name, 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 *Programming API 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 can be read at any time, but only some properties can 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.

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`.

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

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

```
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_PROPERTY_VM_POWER_STATE,
                      &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, because 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 path name 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.

```
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 Objects

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 can 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 operation. A job handle can 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 properties that are set when the job has completed. These result properties contain information returned by a completed job. Different kinds of jobs have different return values.

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

Signaling a Job Object

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

- Poll the job, by calling a non-blocking function that checks whether it has been signaled. This is described in [“Polling the Job Object for Completion”](#) on page 15.
- Block until the job is signaled, by calling the `VixJob_Wait()` function. This is described in [“Using the Job Object to Block Calls”](#) on page 16.
- Register a callback function that is called when the job is signaled. This is described in [“Using a Callback Function”](#) on page 18.

A typical asynchronous call looks similar to `VixVM_Open()`, and there are several common patterns shared by all asynchronous Vix calls. The following code example shows the signature of `VixVM_Open()` for reference.

```
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.

Polling the Job Object for Completion

The job object tracks the status of an asynchronous operation. You can interrogate the completion status of a job object using the function `VixJob_CheckCompletion()`. This is a non-blocking function that returns a Boolean value representing the completion state of the asynchronous operation. The following example shows the use of `VixJob_CheckCompletion()` in a polling loop.

```
Bool openVMWithPolling(const VixHandle hostHandle,
```

```

        const char *vmName,
        VixHandle *vmHandle)
{
    VixError err = VIX_OK;
    VixHandle jobHandle = VIX_INVALID_HANDLE;

    if (!vmHandle) {
        return FALSE;
    }
    *vmHandle = VIX_INVALID_HANDLE;

    // Start asynchronous operation.
    jobHandle = VixVM_Open(hostHandle,
        vmName,
        NULL, // callbackProc,
        NULL); // clientData

    // Poll the job object for completion of asynchronous operation.
    for ( Bool completed = FALSE; completed != TRUE; ) {
        sleep(1);
        err = VixJob_CheckCompletion(jobHandle, &completed);
        if (VIX_OK != err) {
            Vix_ReleaseHandle(jobHandle);
            return FALSE;
        }
        err = Vix_GetProperties(jobHandle,
            VIX_PROPERTY_JOB_RESULT_HANDLE,
            vmHandle,
            VIX_PROPERTY_NONE);
        if (VIX_OK != err) {
            Vix_ReleaseHandle(jobHandle);
            *vmHandle = VIX_INVALID_HANDLE;
            return FALSE;
        }
    }
    // end "for"
    Vix_ReleaseHandle(jobHandle);
    return TRUE;
    // Caller must release vmHandle.
}

```

Using the Job Object to Block Calls

The job object allows a client to block until an asynchronous call completes. This achieves the same result as if the asynchronous call were a synchronous call. Here is an example of how this can be done, using 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

// Wait for completion of operation.
err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    goto abort;
}

// Release handle when done.
Vix_ReleaseHandle(jobHandle);

```

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. The following example shows how this is done.

```

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

// Wait for completion of operation.
err = VixJob_Wait(jobHandle, VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    goto abort;
}

// Collect desired result of operation.

```

```

err = Vix_GetProperties(jobHandle,
                      VIX_PROPERTY_JOB_RESULT_HANDLE,
                      &hostHandle,
                      VIX_PROPERTY_NONE);

// Release handle when done.
Vix_ReleaseHandle(jobHandle);

```

For convenience, you can also extract properties from the job object with the `VixJob_Wait()` function. The following example shows how this is done.

```

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

// Wait for completion of operation. Collect result handle.
err = VixJob_Wait(jobHandle,
                  VIX_PROPERTY_JOB_RESULT_HANDLE,
                  &vmHandle,
                  VIX_PROPERTY_NONE);

// Release handle when done.
Vix_ReleaseHandle(jobHandle);

```

For simplicity, most of the examples in this document use the approach of calling `VixJob_Wait()` and requesting the desired properties at the same time. For better performance and responsiveness, applications might want to use callback functions.

Callback Functions

All asynchronous Vix functions, such as `VixVM_Open()`, include a parameter for a callback procedure and a parameter that is passed to the callback procedure. These parameters are optional, so a caller can pass `NULL` for either. The prototype of this callback procedure parameter is:

```

typedef void (VixEventProc)(VixHandle handle,
                           VixEventType eventType,
                           VixHandle moreEventInfo,
                           void *clientData);

```

Using a Callback Function

If the caller provides a callback procedure, that procedure is registered with the job object and is invoked when the job object is signaled. For example, if a caller passes a callback procedure to `VixVM_Open()`, that callback procedure is invoked when the

virtual machine has been opened. This situation could happen either before or after `VixVM_Open()` returns. It also can happen on any thread.

This mechanism allows the Vix function to complete asynchronously, so the application should not call `VixJob_Wait()` when using a callback function.

When a callback procedure is invoked, it is passed the `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.

```

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

    /*
     * Ignore progress callbacks. Check only for final signal.
     */
    if (VIX_EVENTTYPE_CALLBACK_SIGNALLED != eventType) {
        return;
    }

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

    Vix_ReleaseHandle(jobHandle);

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

int main()
{
    VixError err = VIX_OK;
    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

// Block for host connection to complete.
err = VixJob_Wait(jobHandle,
                 VIX_PROPERTY_JOB_RESULT_HANDLE,
                 &hostHandle,
                 VIX_PROPERTY_NONE);
Vix_ReleaseHandle(jobHandle);
if (VIX_OK != err) {
    goto abort;
}

// Use callback function to capture completion of virtual machine open.
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.
 */
abort:
    Vix_ReleaseHandle(jobHandle);
}

```

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()`.

Event Models

The Vix API provides the flexibility to handle events in different ways, depending on the needs of the client. The “event pump” mechanism allows you to process asynchronous operations in single-threaded clients. This section discusses both ways to manage asynchronous operations.

Multi-threaded Event Model

The multi-threaded event model is the default model for Vix clients. Its use is described in detail in the section above called “[Callback Functions](#)” on page 18. This model is the easiest to use, if your client code is thread-safe.

Using this model, the Vix library creates worker threads as needed to process asynchronous operations in parallel. Callback functions are invoked on the worker threads under the control of the Vix library.

Single-threaded Event Model

Using the single-threaded model, all asynchronous processing and all event reporting is deferred until the thread calls `Vix_PumpEvents()`. Each call to `Vix_PumpEvents()` results in the completion of one step of an asynchronous operation. At appropriate times during the operation, control is transferred to the callback routine, if one was specified in the call to the asynchronous function that began the operation.

The single-threaded event model is useful for clients that are not thread safe, or that depend on libraries that are not thread-safe. All processing is done in a single thread. You have to write extra code for a polling loop, but you need not be concerned with synchronizing multiple threads.

Using the Event Pump

To use the event pump, two things are required:

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

```
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,
                 VIX_PROPERTY_JOB_RESULT_HANDLE,
                 &hostHandle,
                 VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}

```

- Call `Vix_PumpEvents()` at appropriate times to advance processing of the asynchronous operation.

```
Vix_PumpEvents(hostHandle, VIX_PUMPEVENTOPTION_NONE);
```

- Check the results of the asynchronous operation, either by polling or by callback. The polling method is explained in [“Polling for Completion in a Single-Threaded Client”](#) on page 22. The callback method is explained in [“Using a Callback Function in a Single-Threaded Client”](#) on page 24.

Polling for Completion in a Single-Threaded Client

Single-threaded clients must share the thread between the main logic of the program and the processing that takes place as a result of calling an asynchronous function. In a single-threaded client, the processing can only take place during a call to `Vix_PumpEvents()`.

That fact implies that the single-threaded client cannot call `VixJob_Wait()`. In a multi-threaded client, a call to `VixJob_Wait()` returns when the asynchronous operation is completed by another thread. However, in a single-threaded client, `VixJob_Wait()` never returns because there is no processing happening on any thread during the call to `VixJob_Wait()`.

One solution is to poll for completion using `VixJob_CheckCompletion()` instead of calling `VixJob_Wait()`. You alternate pumping with polling, as in the following example.

```

// Function to poll for completion.
// Returns TRUE if successful; otherwise returns FALSE.
Bool CheckCompletion(VixHandle hostHandle,
                   VixHandle jobHandle,
                   int timeOut)
{
    Bool completed = FALSE;

```

```

VixError err = VIX_OK;
int secondCount = 0;

while (!completed && secondCount < timeOut) {
    Vix_PumpEvents(hostHandle, VIX_PUMPEVENTOPTION_NONE);
    err = VixJob_CheckCompletion(jobHandle, &completed);
    if (VIX_OK != err) {
        return FALSE;
    }
    secondCount++;
    sleep(1);
}
if ((!completed)
&& (secondCount == ONE_MINUTE)) {
    return FALSE;
}
return TRUE;
}

// Part of virtual machine processing below.
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

// Don't wait; test the virtual machine power state immediately.
VixPowerState powerState = 0;
err = Vix_GetProperties(vmHandle,
                      VIX_PROPERTY_VM_POWER_STATE,
                      &powerState,
                      VIX_PROPERTY_NONE);

if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
// powerState may be VIX_POWERSTATE_SUSPENDING.
// Now wait for completion.
if (CheckCompletion(hostHandle, jobHandle, 30)) {
    // Now the powerState should be ready.
    err = Vix_GetProperties(vmHandle,
                          VIX_PROPERTY_VM_POWER_STATE,
                          &powerState,
                          VIX_PROPERTY_NONE);

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

```

```

        goto abort;
    }
} // CheckCompletion

```

Using a Callback Function in a Single-Threaded Client

Single-threaded clients also have the option to use callback functions that handle completion. Because the callback has to share the single thread, it cannot happen asynchronously. As with the polling method of completion described in [“Polling for Completion in a Single-Threaded Client”](#) on page 22, the client cannot call `VixJob_Wait()`.

In the single-threaded client, processing takes place only during calls to `Vix_PumpEvents()`. Each call completes one step of the operation. If a callback was specified in the call to the asynchronous function that began the operation, the Vix library invokes the callback function during one of the calls to `Vix_PumpEvents()`.

The code to use this method looks like the following example. The main function initiates a host connection operation (specifying a callback function) and then enters a loop to pump events. When the operation has completed, the callback function returns a handle to the host (or an invalid value in case of failure).

```

Bool handleCallbackDone = FALSE;

void handleCallback(VixHandle jobHandle,
                   VixEventType eventType,
                   VixHandle moreEventInfo,
                   void *clientData)
{
    /*
     * This function is invoked on completion of an asynchronous operation.
     * If the operation succeeded, this function returns a handle
     * that resulted from the asynchronous operation.
     */
    VixError err;
    VixError asyncErr;
    VixHandle resultHandle = VIX_INVALID_HANDLE;

    /*
     * Ignore progress callbacks. Check only for final signal.
     */
    if (VIX_EVENTTYPE_CALLBACK_SIGNALLED != eventType) {
        return;
    }

    err = Vix_GetProperties(jobHandle,
                          VIX_PROPERTY_JOB_RESULT_HANDLE,
                          &resultHandle,

```

```

        VIX_PROPERTY_JOB_RESULT_ERROR_CODE,
        &asyncErr,
        VIX_PROPERTY_NONE);
// Release handle when done.
Vix_ReleaseHandle(jobHandle);
if (VIX_OK != err) {
    // Failed to get properties. Bail.
    *clientData = VIX_INVALID_HANDLE;
    handleCallbackDone = TRUE;
    return;
}

if (VIX_OK == asyncErr) {
    *clientData = (void) resultHandle;
} else {
    // The asynchronous operation failed. Bail.
    *clientData = VIX_INVALID_HANDLE;
}
handleCallbackDone = TRUE;
} // end handleCallback

int main()
{
    VixHandle hostHandle = VIX_INVALID_HANDLE;
    VixHandle jobHandle = 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
                               handleCallback,
                               (void *) &hostHandle);

    // Release handle, if not using it.
    Vix_ReleaseHandle(jobHandle).

    // Pump events and wait for callback.
    handleCallbackDone = FALSE;
    while (!handleCallbackDone) {
        Vix_PumpEvents();
    }
    if (VIX_INVALID_HANDLE == hostHandle) {
        printf(":-(\n");
    } else {
        printf(":-)\n");
    }
}

```

```
    }  
} // end main
```

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 can 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.

When you have a host object, you 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_Disconnect()`. All handles and all states 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 27.

CHAPTER 3 Using the Programming API

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

- [“Connecting to a Host”](#) on page 27
- [“Registering and Unregistering Virtual Machines”](#) on page 30
- [“Getting a Handle to a Virtual Machine”](#) on page 32
- [“Starting or Resuming a Virtual Machine”](#) on page 33
- [“Installing VMware Tools in a Virtual Machine”](#) on page 36
- [“Virtual Machine Guest Operations”](#) on page 37
- [“Powering Off or Suspending a Virtual Machine”](#) on page 38
- [“Importing a Legacy Virtual Machine”](#) on page 40

Connecting to a Host

To work with a virtual machine, you must first 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 28.
- 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 user account on the local host. Refer to [“Connecting to the Local Host as a Specified User”](#) on page 29.
- 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 29.

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 might 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 callback 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.

```
#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,
                               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);
}
```

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.

```
#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);
}
```

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 must be sure that the current user has appropriate permissions to use one or more virtual machines on the local host.

```

#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;
    }
    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 “inventory” before using the virtual machine. The virtual machine inventory 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 inventory, it is not available for any 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 callback 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 27.
- 2 Use the host handle with `VixHost_RegisterVM()` to register the virtual machine by its path name in the host's file system.

```
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 27.
- 2 Make sure the virtual machine is powered off. Refer to [“Powering Off or Suspending a Virtual Machine”](#) on page 38.

If the virtual machine is suspended, first resume it, then power it off. Refer to [“Starting or Resuming a Virtual Machine”](#) on page 33.

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

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

Getting a Handle to a Virtual Machine

Most 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 callback function or a call to `VixJob_Wait()`.

To get a handle to a virtual machine:

- 1 Supply a handle to the host on which the virtual machine is located. Refer to [“Connecting to a Host”](#) on page 27.
- 2 Ensure that the virtual machine is registered on the host. Refer to [“Registering and Unregistering Virtual Machines”](#) on page 30.
- 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.

```
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:

- To start the guest operating system in a virtual machine previously in a powered-off state.
- To resume execution of a guest operating system in a suspended virtual machine.

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

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

For most purposes (especially if you want to do operations in the guest operating system) you must wait for VMware Tools to become active in the guest. Until the VMware Tools utility is ready, no guest operations using the Programming API can complete. Refer to [“Installing VMware Tools in a Virtual Machine”](#) on page 36 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 27.
- 2 Get a handle to the virtual machine. Refer to [“Getting a Handle to a Virtual Machine”](#) on page 32.

- 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.

```

VixError err = VIX_OK;
VixHandle jobHandle = VIX_INVALID_HANDLE;

// Power on the 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);

// 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 timeout...
}
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 27.
- 2 Get a handle to the virtual machine.
Refer to [“Getting a Handle to a Virtual Machine”](#) on page 32.
- 3 Use the virtual machine handle in a call to `VixVM_PowerOn()` to power on the virtual machine.

- 4 (Optional) Use the `VixVM_WaitForToolsInGuest()` function to activate VMware Tools.

Generally, VMware Tools was already active in the guest operating system when the virtual machine was suspended.

```

VixError err = VIX_OK;
VixHandle jobHandle = VIX_INVALID_HANDLE;
int toolsState = 0;

// 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);

// Check if VMware Tools active.
err = Vix_GetProperties(vmHandle,
    VIX_PROPERTY_VM_TOOLS_STATE,
    &toolsState,
    VIX_PROPERTY_NONE);
if (VIX_OK != err) {
    // Handle the error...
    goto abort;
}
Vix_ReleaseHandle(jobHandle);

if (VIX_TOOLSSTATE_RUNNING != toolsState) {
    // 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 timeout...
    }
    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 `VixVM_RunProgramInGuest()`. You need to install the VMware Tools before calling guest functions in the Programming API.

To begin the Tools installation process, you call the Vix function `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 starts automatically. Connect a console window to the virtual machine and use the mouse to navigate the installer wizard.

The `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, VMware recommends that you run the Tools installation on your virtual machines to make sure you are working with the latest version.

`VixVM_InstallToolsInGuest()` is an asynchronous function, so the function call must be completed with a callback function or a call to `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 27.
- 2 Get a handle to the virtual machine. Refer to [“Getting a Handle to a Virtual Machine”](#) on page 32.
- 3 Power on the virtual machine. Refer to [“Starting or Resuming a Virtual Machine”](#) on page 33.
- 4 Use the virtual machine handle to call `VixVM_InstallToolsInGuest()`.
- 5 Using a console connected to the virtual machine, do one of the following:
 - For a Microsoft 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.

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

```

// Mount virtual CD-ROM with VMware Tools installer.
jobHandle = VixVM_InstallTools(vmHandle,
    0, // options
    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 36 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. The following functions require client authentication:

- `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 attempts to authenticate a second time, and the second attempt fails.
- 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 true only 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 27.
- 2 Get a handle to the virtual machine. Refer to [“Getting a Handle to a Virtual Machine”](#) on page 32.
- 3 Power on the virtual machine and wait for VMware Tools to become active. Refer to [“Starting or Resuming a Virtual Machine”](#) on page 33.
- 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

You can stop virtual machine execution in two ways. 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. When you resume operation, the virtual machine continues exactly from the point of suspension.

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 callback 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 27.
- 2 Get a handle to the virtual machine. Refer to [“Getting a Handle to a Virtual Machine”](#) on page 32.
- 3 Use the virtual machine handle in a call to `VixVM_PowerOff()`.
- 4 Check the power state of the virtual machine.

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

// Power off the virtual machine.
jobHandle = VixVM_PowerOff(vmHandle,
    0, // powerOffOptions,
    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_POWERSTATE_POWERED_OFF != 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);
}
```

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 27.
- 2 Get a handle to the virtual machine. Refer to [“Getting a Handle to a Virtual Machine”](#) on page 32.
- 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, // powerOffOptions,
                        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 that 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);
}
```

Importing a Legacy Virtual Machine

The Programming API allows you to run some legacy virtual machines, with some loss of functionality that depends on the virtual hardware version. A legacy virtual machine is one created by an earlier version of VMware software.

The properties of a virtual machine include a virtual hardware version that determine the capabilities of the virtual machine. VMware products generally support the current virtual hardware version and the previous virtual hardware version. Virtual machines older than the previous virtual hardware version are best upgraded, as they are not supported and they may fail to run.

This release of VMware Server supports virtual machines created with Workstation 4 and GSX Server 3. Virtual machines created with Workstation 3 or GSX Server 2 should be upgraded before using them with this release.

Virtual machines created with ESX Server may also be used with the Programming API if they are first exported to the file formats used by Workstation and GSX Server. Consult your ESX Server documentation for more information on exporting virtual machines.

Upgrading Virtual Hardware

When using legacy virtual machines, you can upgrade the virtual hardware by calling the `VixVM_UpgradeVirtualHardware()` function. This function upgrades the virtual hardware to the same level as virtual machines created with the current release.

To upgrade the virtual hardware version of your virtual machine:

- 1 Connect to the host on which the virtual machine is located. Refer to [“Connecting to a Host”](#) on page 27.
- 2 Get a handle to the virtual machine. Refer to [“Getting a Handle to a Virtual Machine”](#) on page 32.
- 3 Power off the virtual machine. Refer to [“Powering Off a Virtual Machine”](#) on page 39.
- 4 Use the virtual machine handle in a call to `VixVM_UpgradeVirtualHardware()`.

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

// Upgrade the virtual hardware.
jobHandle = VixVM_UpgradeVirtualHardware(vmHandle,
    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);
```

NOTE Once you have upgraded the virtual hardware of a legacy virtual machine, you can no longer use that virtual machine with older VMware products — those that use a previous virtual hardware version. If you have any doubts about upgrading, make a copy of the legacy virtual machine before you upgrade the virtual hardware.

Glossary

This appendix defines some of the terms used in the Programming 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 Programming 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.

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 which 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. The file 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 *Programming API Programming Guide*.

Table 5-1.

Date	Description
July 6, 2006	Release 1.0 General Availability.

Index

Numerics

32-bit client support **2**

A

asynchronous functions **10**

asynchronous operations **14, 18, 20**

authentication

 guest operating system **37**

 host operating system **27–29**

B

blocking function calls **16**

C

callback events **20**

callback functions **18, 20**

CD image, VMware Tools **36**

client, standalone **5**

compatibility, version **2**

compiling

 on Linux host **4–5**

 on Windows host **3**

configuration file **43**

connecting to hosts **27–29**

console **43**

D

DLLs **3–5**

documentation **1–2**

E

error codes **10**

functions **10**

masking **10–11**

testing **11**

VIX_OK **10**

ESX Server **41**

event pump **21**

F

functions

 asynchronous **10**

 synchronous **10**

G

gcc **5**

GetProperties() function **13**

GSX Server 2 **41**

GSX Server 3 **41**

guest operating system **36–37, 43**

H

handle independence **8**

handles

 deleting **9**

 host **26**

 internal references **10**

 key types **7**

 opening **9**

 properties **12, 17**

 releasing **8–10**

 signalling a job handle **15**

 thread safety **12**

- virtual machine **26, 32**
- header files
 - on Linux host **4–5**
 - on Windows host **3–4**
- host computer **43**
- host handles **26**
- host operating system **43**
- hosts
 - connecting **27–29**

I

- importing virtual machines **40–41**
- installation **3**
 - on Linux host **5**

J

- job handle, signaling **15**
- job objects
 - blocking function calls **16**
 - defined **14**
 - retrieving results from **17**

L

- legacy virtual machine **43**
- legacy virtual machines **40, 42**
- library files
 - on Linux host **4–5**
 - on Windows host **3–4**
- linked clone **43**

M

- masking error codes **10**
- multithreading **12**
 - and event pumps **21**
 - performance implications **12**

O

- objects **7**
- opening a virtual machine **32**

- opening hosts **27–29**

P

- powering off virtual machines **38–39**
- powering on virtual machines **33**
- property lists **14**

R

- reference counting **8, 10**
- registering virtual machines **30–31**
- releasing handles **8–10**
- resume **44**
- resuming virtual machines **34**
- revert to snapshot **44**

S

- signaling jobs **15**
- snapshot **44**
- starting virtual machines **33**
- stopping virtual machines **39**
- suspend **44**
- suspending virtual machines **38, 40**
- synchronous functions **10**

T

- thread safety **12**
- Tools **36–37**

U

- unregistering virtual machines **31**
- upgrading virtual hardware **41**

V

- virtual disk **44**
- virtual hardware **41**
- virtual hardware version **41**
- virtual machine **44**
- virtual machine configuration **44**
- virtual machine configuration file **45**
- virtual machine handle **32**

virtual machine handles **26**
VIX_OK **10**
VMware Tools **36–37, 45**
VMware Tools Service **45**

W

Workstation 3 **41**
Workstation 4 **41**

