Virtuozzo 7

Virtualization SDK: Programmer's Guide

March 23, 2017

Parallels International GmbH
Vordergasse 59
8200 Schaffhausen
Switzerland
Tel: + 41 52 632 0411
Fax: + 41 52 672 2010
www.virtuozzo.com

Contents

CHAPTER 1

Getting Started

In This Chapter

Overview

Virtuozzo Virtualization SDK is a development kit used to create and integrate custom software solutions with Virtuozzo virtualization products. The SDK provides cross-platform C and Python APIs.

Virtuozzo Virtualization SDK comprises the following components:

- C header files.
- Dynamic libraries.
- Python package for developing applications in Python.
- Virtuozzo command line tools (`prlctl, prlsrvctl`).
- Virtualization SDK Programmer's Guide (this document).
- C API Reference Guide.
- Python API Reference Guide.
- Command Line Reference Guide.

System Requirements

Linux Development

When developing applications to run on Linux, system requirements are as follows:

Hardware Requirements

- Intel-compatible x86 (32-bit) or x64 (64-bit) processor.
- Ethernet network adapter.

Software Requirements

- Red Hat Enterprise Linux 7.x
- Red Hat Enterprise Linux 6.x
- Red Hat Enterprise Linux 5.x
- CentOS 7.x
- CentOS 6.x
- CentOS 5.x
- 32-bit or 64-bit Python 2.5, 2.6 or 2.7 for the Virtuozzo Python API.

Windows Development

When developing applications to run on Windows, system requirements are as follows:

Hardware Requirements

- Intel-compatible x86 (32-bit) or x64 (64-bit) processor.
- Ethernet or Wi-Fi network adapter.

Software Requirements

- Windows 2000 or later.
- 32-bit Python 2.5, 2.6 or 2.7 for the Virtuozzo Python API.

Network Requirements

Virtuozzo is listening for incoming connections on port 64000. When developing a remote application, make sure that this port is not blocked by a firewall.

CHAPTER 2

Virtuozzo C API Concepts

In This Chapter

Compiling Applications

The following topics describe how to compile applications that use Virtuozzo C API on a specific operating system.

Compiling Applications on Linux

The following is a sample make file that can be used to compile applications on Linux:

```
# set the appropriate path to the SDK headers
SDK_INSTALL_PATH=/usr

OBJS = SdkWrap.o main.o
CXX = g++
CXXFLAGS = -I$(SDK_INSTALL_PATH)/include/parallels-virtualization-sdk
LDFLAGS = -ldl

# Set the current folder name
TARGET = Example

all : $(TARGET)

$(TARGET) : $(OBJS)
	$(CXX) -o $@ $(LDFLAGS) $(OBJS)

main.o : main.cpp
	$(CXX) -c -o $@ $(CXXFLAGS) main.cpp

SdkWrap.o : $(SDK_INSTALL_PATH)/share/parallels-virtualization-
sdk/helpers/SdkWrap/SdkWrap.cpp
	$(CXX) -c -o $@ $(CXXFLAGS) $(SDK_INSTALL_PATH)/share/parallels-virtualization-
sdk/helpers/SdkWrap/SdkWrap.cpp

clean:
```

```
	@rm -f $(TARGET) $(OBJS)

.PHONY : all clean
```

Compiling Application on Windows

The following steps describe how to set up a project in Microsoft Visual Studio:

1. Create a project of your choice in a usual way and open the project **Property Pages** windows.
2. In the **C/C++ -> General -> Additional Include Directories** section, add the path to the `Include` and the `Helpers\SdkWrap` sub-directories located in the main directory where you have the SDK installed.
3. Add the following files from the `Helpers\SdkWrap` subdirectory to the project:

   `SdkWrap.h`

   `SdkWrap.cpp`

   These are the helper files that provide a set of methods for loading and unloading dynamic libraries. You can use the included source code file to customize this functionality if you wish.
4. Add the following `#include` directive to your program:

   ```
   #include "SdkWrap.h"
   ```

   The standard libraries used by the samples provided in this guide are:

   ```
   #include <stdio.h>
   #include <stdlib.h>
   #include <string.h>
   ```

Handles

The Virtuozzo C API is a set of functions that operate on objects. Objects are not accessed directly. Instead, references to these objects are used. These references are known as *handles*.

Handle Types

`PRL_HANDLE` is the only handle type used in the C API. It is a pointer to an integer and it is defined in `PrlTypes.h`.

`PRL_HANDLE` can reference any type of object within the API. The type of object that `PRL_HANDLE` references determines the `PRL_HANDLE` type. A list of handle types can be found in the `PRL_HANDLE_TYPE` enumeration in `PrlEnums.h`.

A handles' type can be extracted using the `PrlHandle_GetType` function. A string representation of the handle type can then be obtained using the `handle_type_to_string` function.

Obtaining a Handle

A handle is usually obtained by calling a function belonging to another handle, which we may call a "parent". For example, a `PHT_VIRTUAL_MACHINE` handle is obtained by calling a function that belongs to the `PHT_SERVER` handle. A virtual device handle (e.g. `PHT_VIRTUAL_DEV_HARD_DISK`) is obtained by calling a function that belongs to the `PHT_VIRTUAL_MACHINE` handle, and so forth. The **Virtuozzo C API Reference** guide contains a description of every available handle and explains how each particular handle type can be obtained. The examples in this guide also demonstrate how to obtain handles of different types.

Freeing a Handle

Virtuozzo API handles are reference counted. Each handle contains a count of the number of references to it held by other objects. A handle stays in memory for as long as the reference count is greater than zero. A program is responsible for freeing any handles that are no longer needed. A handle can be freed using the `PrlHandle_Free` function. The function decreases the reference count by one. When the count reaches zero, the object is destroyed. Failing to free a handle after it has been used will result in a memory leak.

Multithreading

Virtuozzo API handles are thread safe. They can be used in multiple threads at the same time. To maintain the proper reference counting, the count should be increased each time a handle is passed to another thread by calling the `PrlHandle_AddRef` function. If this is not done, freeing a handle in one thread may destroy it while other threads are still using it.

Example

The following code snippet demonstrates how to obtain a handle, how to determine its type, and how to free it when it's no longer needed. The code is a part of the bigger example that demonstrates how to log in to Virtuozzo (the full example is provided later in this guide).

```
PRL_HANDLE hServer = PRL_INVALID_HANDLE;
PRL_RESULT ret;

ret = PrlSrv_Create(&hServer);
if (PRL_FAILED(ret))
{
    fprintf(stderr, "PrlSvr_Create failed, error: %s",
            prl_result_to_string(ret));
    return PRL_ERR_FAILURE;
}

// Determine the type of the hServer handle.
PRL_HANDLE_TYPE nHandleType;
PrlHandle_GetType(hServer, &nHandleType);
printf("Handle type: %s\n",
    handle_type_to_string(nHandleType));

// Free the handle when it is no longer needed.
```

```
PrlHandle_Free(hServer);
```

Synchronous Functions

The Virtuozzo C API provides synchronous and asynchronous functions. Synchronous functions run in the same thread as the caller. When a synchronous function is called it completes executing before returning control to the caller. Synchronous functions return `PRL_RESULT`, which is a integer indicating success or failure of the operation. Consider the `PrlSrv_Create` function. The purpose of this function is to obtain a handle of type `PHT_SERVER`. The handle is required to access most of the functionality within the Virtuozzo C API. The syntax of `PrlSrv_Create` is as follows:

```
PRL_RESULT PrlSrv_Create(
    PRL_HANDLE_PTR handle
);
```

The following is an example of the `PrlSrv_Create` function call:

```
// Declare a handle variable.
PRL_HANDLE hServer = PRL_INVALID_HANDLE;

// Call the PrlSrv_Create to obtain the handle.
PRL_RESULT res = PrlSrv_Create(&hServer);

// Examine the function return code.
// PRL_FAILED is a macro that evaluates a variable of type PRL_RESULT.
// A return value of True indicates success; False indicates failure.
if (PRL_FAILED(res))
{
    printf("PrlSrv_Create returned error: %s\n",
        prl_result_to_string(res));
    exit(ret);
}
```

Asynchronous Functions

An asynchronous operation is executed in its own thread. An asynchronous function that started the operation returns to the caller immediately without waiting for the operation to complete. The results of the operation can be verified later when needed. Asynchronous functions return `PRL_HANDLE`, which is a pointer to an integer and is a handle of type `PHT_JOB`. The handle is used as a reference to the asynchronous job executed in the background. The general procedure for calling an asynchronous function is as follows:

1. Register an event handler (callback function).
2. Call an asynchronous function.
3. Analyze the results of events (jobs) within the callback function.
4. Handle the appropriate event in the callback function.
5. Un-register the event handler when it is no longer needed.

The Callback Function (Event Handler)

Asynchronous functions return data to the caller by means of an *event handler* (or *callback function*). The callback function could be called at any time, depending on how long the asynchronous function takes to complete. The callback function must have a specific signature. The prototype can be found in `PrlApi.h` and is as follows:

```
typedef PRL_METHOD_PTR(PRL_EVENT_HANDLER_PTR) (
            PRL_HANDLE hEvent,
            PRL_VOID_PTR data
            );
```

The following is an example of the callback function implementation:

```
static PRL_RESULT OurCallback(PRL_HANDLE handle, void *pData)
{
    // Event handler code...

    // You must always release the handle before exiting.
    PrlHandle_Free(handle);
}
```

A handle received by the callback function can be of type `PHT_EVENT` or `PHT_JOB`. The type can be determined using the `PrlHandle_GetType` function. The `PHT_EVENT` type indicates that the callback was called by a system event. If the type is `PHT_JOB` then the callback was called by an asynchronous job started by the program.

To handle system events within a callback function:

1. Get the event type using `PrlEvent_GetType`.
2. Examine the event type. If it is relevant, a handle of type `PHT_EVENT_PARAMETER` can be extracted using `PrlEvent_GetParam`.
3. Convert the `PHT_EVENT_PARAMETER` handle to the appropriate handle type using `PrlEvtPrm_ToHandle`.

To handle jobs within a callback function:

1. Get the job type using `PrlJob_GetType`. A job type can be used to identify the function that started the job and to determine the type of the result it contains. For example, a job of type `PJOC_SRV_GET_VM_LIST` is started by `PrlSrv_GetVmList` function call, which returns a list of virtual servers.
2. Examine the job type. If it is relevant, proceed to the next step.
3. Get the job return code using `PrlJob_GetRetCode`. If it doesn't contain an error, proceed to the next step.
4. Get the result (a handle of type `PHT_RESULT)` from the job handle using `PrlJob_GetResult`.
5. Get a handle to the result using `PrlResult_GetParam`. Note that some functions return a list (ie. there can be more than a single parameter in the result). For example,

`PrlSrv_GetVmList` returns a list of available virtual servers. In such cases, use `PrlResult_GetParamCount` and `PrlResult_GetParamByIndex`.

6 Implement code to use the handle obtained in step 5.

Note: You must always free the handle that was passed to the callback function before exiting, regardless of whether you actually used it or not. Failure to do so will result in a memory leak.

The following skeleton code demonstrates implementation of the above steps. In this example, the objective is to handle events of type `PET_DSP_EVT_HOST_STATISTICS_UPDATED` that are generated by a call to function `PrlSrv_SubscribeToHostStatistics`, and to obtain the result from a job of type `PJOC_SRV_GET_VM_LIST`.

```
static PRL_RESULT OurCallbackFunction(PRL_HANDLE hHandle, PRL_VOID_PTR pUserData)
{
    PRL_JOB_OPERATION_CODE nJobType = PJOC_UNKNOWN; // job type
    PRL_HANDLE_TYPE nHandleType = PHT_ERROR; // handle type
    PRL_HANDLE hVm = PRL_INVALID_HANDLE; // virtual server handle
    PRL_HANDLE hParam = PRL_INVALID_HANDLE; // event parameter
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE; // job result
    PRL_UINT32 nParamsCount = -1; // parameter count
    PRL_UINT32 nParamIndex = -1; // parameter index
    PRL_RESULT err = PRL_ERR_UNINITIALIZED; // error

    // Check the type of the received handle.
    PrlHandle_GetType(hHandle, &nHandleType);

    if (nHandleType == PHT_EVENT) // Event handle
    {
        PRL_EVENT_TYPE EventType;
        PrlEvent_GetType(hHandle, &EventType);

        // Check if the event type is a statistics update.
        if (EventType == PET_DSP_EVT_HOST_STATISTICS_UPDATED)
        {
            // Get handle to PHT_EVENT_PARAMETER.
            PRL_HANDLE hEventParameters = PRL_INVALID_HANDLE;
            PrlEvent_GetParam(hHandle, 0, &hEventParameters);

            // Get handle to PHT_SYSTEM_STATISTICS.
            PRL_HANDLE hServerStatistics = PRL_INVALID_HANDLE;
            PrlEvtPrm_ToHandle(hEventParameters, &hServerStatistics);

            // Code goes here to extract the statistics data
            // using hServerStatistics.

            PrlHandle_Free(hServerStatistics);
            PrlHandle_Free(hEventParameters);
        }
    }
    else if (nHandleType == PHT_JOB) // Job handle
    {
        // Get the job type.
        PrlJob_GetOpCode(hHandle, &nJobType);

        // Check if the job type is PJOC_SRV_GET_VM_LIST.
        if (nJobType == PJOC_SRV_GET_VM_LIST)
        {
            // Check the job return code.
```

```
            PRL_RESULT nJobRetCode;
            PrlJob_GetRetCode(hHandle, &nJobRetCode);
            if (PRL_FAILED(nJobRetCode))
            {
                fprintf(stderr, "[B]%.8X: %s\n", nJobRetCode,
                prl_result_to_string(nJobRetCode));
                PrlHandle_Free(hHandle);
                return nJobRetCode;
            }

            err = PrlJob_GetResult(hHandle, &hJobResult);

            // if (err != PRL_ERR_SUCCESS), process the error here.

            // Determine the number of parameters in the result.
            PrlResult_GetParamsCount(hJobResult, &nParamsCount);

            // Iterate through the parameter list.
            for(nParamIndex = 0; nParamIndex < nParamsCount ; nParamIndex++)
            {
                // Obtain a virtual server handle (PHT_VIRTUAL_MACHINE).
                PrlResult_GetParamByIndex(hJobResult, nParamIndex, &hVm);

                // Code goes here to obtain virtual server info from hVm.

                // Free the handle when done using it.
                PrlHandle_Free(hVm);
            }
            PrlHandle_Free(hJobResult);
        }
    }

    PrlHandle_Free(hHandle);
    return PRL_ERR_SUCCESS;
}
```

Registering / Unregistering an Event Handler

The `PrlSrv_RegEventHandler` function is used to register an event handler, `PrlSrv_UnregEventHandler` is used to unregister an event handler.

Note: When an event handler is registered, it will receive all of the events/jobs regardless of their origin. It is the responsibility of the program to identify the type of the event and to handle each one accordingly.

```
// Register an event handler.
ReturnDataClass rd; // some user-defined class.
PrlSrv_RegEventHandler(hServer, OurCallbackFunction, &rd);

// Make a call to an asynchronous function here.
// OurCallbackFunction will be called by the background thread
// as soon as the job is completed, and code within
// OurCallbackFunction can populate the ReturnDataClass instance.
// For example, we can make the following call here:

hJob = PrlSrv_GetVmList(hServer);
PrlHandle_Free(hJob);

// Please note that we still have to obtain the
// job object (hJob above) and free it; otherwise
```

```
// we will have memory leaks.

// Unregister the event handler when it is no longer needed.
PrlSrv_UnregEventHandler(hServer, OurCallbackFunction, &rd);
```

Calling Asynchronous Functions Synchronously

It is possible to call an asynchronous function synchronously by using the `PrlJob_Wait` function. The function takes two parameters: a `PHT_JOB` handle and a timeout value in milliseconds. Once you call the function, the main thread will be suspended and the function will wait for the asynchronous job to complete. The function will return when the job is completed or when timeout value is reached, whichever comes first. The following code snippet illustrates how to call an asynchronous function `PrlServer_Login` synchronously:

```
// Log in (PrlSrv_Login is asynchronous).
PRL_HANDLE hJob = PrlSrv_Login(
        hServer,
        szHostnameOrIpAddress,
        szUsername,
        szPassword,
        0,
        0,
        0,
        PSL_LOW_SECURITY);

// Wait for a maximum of 10 seconds for
// asynchronous function PrlSrv_Login to complete.
ret = PrlJob_Wait(hJob, 10000);
if (PRL_FAILED(ret))
{
    fprintf(stderr, "PrlJob_Wait for PrlSrv_Login returned with error: %s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);
    return -1;
}

// Analyse the result of the PrlServer_Login call.
PRL_RESULT nJobResult;
ret = PrlJob_GetRetCode(hJob, &nJobResult);
if (PRL_FAILED(nJobResult))
{
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);
    printf("Login job returned with error: %s\n",
        prl_result_to_string(nJobResult));
    return -1;
}
else
{
    printf("login successfully performed\n");
}
```

Strings as Return Values

Sting values in the Virtuozzo C API are received by passing a `char` pointer to a function which populates it with data. It is the responsibility of the caller to allocate the memory required to receive the value, and to free it when it is no longer needed. Since in most cases we don't know the string size in advance, we have to either allocate a chunk of memory large enough for any possible value or to determine the exact required size. To determine the required buffer size, the following two approaches can be used:

1. Calling the same function twice: first, to obtain the required buffer size, and second, to receive the actual string value. To get the required buffer size, call the function passing a null pointer as a value of the output parameter, and pass 0 (zero) as a value of the variable that is used to specify the buffer size. The function will calculate the required size and will populate the variable with the correct value, which you can use to initialize a variable that will receive the string. You can then call the function again to get the actual string value.
2. It is also possible to use a static buffer. If the length of the buffer is large enough, you will simply receive the result. If the length is too small, a function will fail with the `PRL_ERR_BUFFER_OVERRUN` error but it will populate the "buffer_size" variable with the required size value. You can then allocate the memory using the received value and call the function again to get the results.

Consider the following function:

```
PRL_RESULT PrlVmCfg_GetName(
    PRL_HANDLE hVmCfg,
    PRL_STR sVmName,
    PRL_UINT32_PTR pnVmNameBufLength
);
```

The `PrlVmCfg_GetName` function above is a typical Virtuozzo API function that returns a string value (in this case, the name of a virtual server). The `hVmCfg` parameter is a handle to an object containing the virtual server configuration information. The `sVmName` parameter is a `char` pointer. It is used as output that receives the virtual server name. The variable must be initialized on the program side with enough memory allocated for the expected string. The size of the buffer must be specified using the `pnVmNameBufLength` variable.

The following example demonstrates how to call the function using the first approach:

```
PRL_RESULT ret;
PRL_UINT32 nBufSize = 0;

// Get the required buffer size.
ret = PrlVmCfg_GetName(hVmCfg, 0, &nBufSize);

// Allocate the memory.
PRL_STR pBuf = (PRL_STR)malloc(sizeof(PRL_CHAR) * nBufSize);

// Get the virtual server name.
ret = PrlVmCfg_GetName(hVmCfg, pBuf, &nBufSize);

printf("VM name: %s\n", pBuf);
```

```
// Deallocate the memory.
free(pBuf);
```

The following example uses the second approach. To test the buffer-overrun scenario, set the `sVmName` array size to some small number.

```
#define MY_STR_BUF_SIZE 1024

PRL_RESULT ret;
char sVmName[MY_STR_BUF_SIZE];
PRL_UINT32 nBufSize = MY_STR_BUF_SIZE;

// Get the virtual server name.
ret = PrlVmCfg_GetName(hVmCfg, sVmName, &nBufSize);

// Check for errors.
if (PRL_SUCCEEDED(ret))
{
    // Everything's OK, print the server name.
    printf("VM name: %s\n", sVmName);
}
else if (ret == PRL_ERR_BUFFER_OVERRUN)
{
    // The sVmName array size is too small.
    // Get the required size, allocate the memory,
    // and try getting the VM name again.

    PRL_UINT32 nSize = 0;
    PRL_STR pBuf;

    // Get the required buffer size.
    ret = PrlVmCfg_GetName(hVmCfg, 0, &nSize);

    // Allocate the memory.
    pBuf = (PRL_STR)malloc(sizeof(PRL_CHAR) * nSize);

    // Get the virtual server name.
    ret = PrlVmCfg_GetName(hVmCfg, pBuf, &nSize);

    printf("VM name: %s\n", pBuf);

    // Dallocate the memory.
    free(pBuf);
}
```

Error Handling

Synchronous Functions

All synchronous Virtuozzo C API functions return **PRL_RESULT**, which is an integer indicating success or failure of the operation.

Error Codes for Asynchronous Functions

All asynchronous functions return `PRL_HANDLE`. The error code (return value) in this case can be extracted with `PrlJob_GetRetCode` after the asynchronous job has finished.

Analyzing Return Values

The C API provides the following macros to work with error codes:

`PRL_FAILED`	Returns True if the return value indicates failure, or False if the return value indicates success.
`PRL_SUCCEEDED`	Returns True if the return value indicates success, or False if the return value indicates failure.
`prl_result_to_string`	Returns a string representation of the error code.

The following code snippet attempts to create a directory on the host and analyzes the return value (error code) of asynchronous function `PrlSrv_CreateDir`.

```
// Attempt to create directory /tmp/TestDir on the host.
char *szRemoteDir = "/tmp/TestDir";
hJob = PrlSrv_FsCreateDir(hServer, szRemoteDir);

// Wait for a maximum of 5 seconds for asynchronous
// function PrlSrv_FsCreateDir to complete.
PRL_RESULT resWaitForCreateDir = PrlJob_Wait(hJob, 5000);
if (PRL_FAILED(resWaitForCreateDir))
{
    fprintf(stderr, "PrlJob_Wait for PrlSvr_FsCreateDir failed with error: %s\n",
        prl_result_to_string(resWaitForCreateDir));
    PrlHandle_Free(hJob);
    return -1;
}

// Extract the asynchronous function return code.
PrlJob_GetRetCode(hJob, &nJobResult);
if (PRL_FAILED(nJobResult))
{
    fprintf(stderr, "Error creating directory %s. Error returned: %s\n",
        szRemoteDir, prl_result_to_string(nJobResult));
    PrlHandle_Free(hJob);
    return -1;
}

PrlHandle_Free( hJob );
printf( "Remote directory %s was successfully created.\n", szRemoteDir );
```

Descriptive Error Strings

Descriptive error messages can sometimes be obtained using the `PrlJob_GetError` function. This function will return a handle to an object of type `PHT_EVENT`. In cases where `PrlJob_GetError` is unable to return error information, `PrlApi_GetResultDescription` can be used. Although it is possible to avoid using `PrlJob_GetError` and use `PrlJob_GetResultDescription` instead, it is recommended to first use `PrlJob_GetError`,

and if this doesn't return additional descriptive error information then use `PrlApi_GetResultDescription`. The reason is that sometimes errors contain dynamic parameters. The following example demonstrates how to obtain descriptive error information:

```
PrlJob_GetRetCode(hJob, &nJobResult);

    PRL_CHAR szErrBuff[1024];
    PRL_UINT32 nErrBuffSize = sizeof(szErrBuff);
    PRL_HANDLE hError = PRL_INVALID_HANDLE;
    PRL_RESULT ret = PrlJob_GetError(hJob, &hError);

    // Check if additional error information is available.
    if (PRL_SUCCEEDED(ret))    // Additional error information is available.
    {
        // Additional error information is available.
        ret = PrlEvent_GetErrString(hError, PRL_FALSE, PRL_FALSE, szErrBuff,
&nErrBuffSize);
        if (PRL_FAILED(ret))
        {
            printf("PrlEvent_GetErrString returned error: %.8x %s\n",
                ret, prl_result_to_string(ret));
        }
        else
        {
            // Extra error information is available, display it.
            printf("Error returned: %.8x %s\n", nJobResult,
prl_result_to_string(nJobResult));
            printf("Descriptive error: %s\n", szErrBuff);
        }
    }
    else
    {
        // No additional error information available, so use
PrlApi_GetResultDescription.
        ret = PrlApi_GetResultDescription(nJobResult, PRL_FALSE, PRL_FALSE, szErrBuff,
&nErrBuffSize);
        if (PRL_FAILED(ret))
        {
            printf("PrlApi_GetResultDescription returned error: %s\n",
                prl_result_to_string(ret));
        }
        else
        {
            printf("Error returned: %.8x %s\n", nJobResult,
prl_result_to_string(nJobResult));
            printf("Descriptive error: %s\n", szErrBuff);
        }
    }
    // Free handles, return the error code.
    PrlHandle_Free(hJob);
    PrlHandle_Free(hError);
    return nJobResult;
}
```

CHAPTER 3

Virtuozzo C API by Example

In This Chapter

Obtaining Server Handle and Logging In

The following steps are required in any program using the Virtuozzo C API:

1. Load the Virtuozzo Virtualization SDK library using the `SdkWrap_Load` function.
2. Initialize the API using the `PrlApi_InitEx` function.
3. Create a Server handle using the `PrlSrv_Create` function. The function returns a `PHT_SERVER` handle identifying the Virtuozzo host.
4. Call `PrlSrv_LoginLocal` or `PrlSrv_Login` to login to Virtuozzo. The former is used when the program is running on Virtuozzo. The latter is used to log in to a remote server.

To end a session with a Virtuozzo host, the following steps must be performed before exiting the application:

1. Call `PrlSrv_Logoff` to log off.
2. Free the Server handle using `PrlHandle_Free`.
3. Call `PrlApi_Deinit` to de-initialize the library.
4. Call `SdkWrap_Unload` to unload the API.

Example

The following sample functions demonstrates how to perform the steps described above.

```
// Intializes the SDK library and
// logs in to the Virtuozzo host locally.
// Obtains a handle of type PHT_SERVER identifying
// the Virtuozzo host.

PRL_RESULT LoginLocal(PRL_HANDLE &hServer)
```

```
{
    // Variables for handles.
    PRL_HANDLE hJob = PRL_INVALID_HANDLE; // job handle
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE; // job result

    // Variables for return codes.
    PRL_RESULT err = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Use the correct dynamic library depending on the platform.
    #ifdef _WIN_
    #define SDK_LIB_NAME "prl_sdk.dll"
    #elif defined(_LIN_)
    #define SDK_LIB_NAME "libprl_sdk.so"
    #endif

    // Load SDK library.
    if (PRL_FAILED(SdkWrap_Load(SDK_LIB_NAME)) &&
        PRL_FAILED(SdkWrap_Load("./" SDK_LIB_NAME)))
    {
        fprintf( stderr, "Failed to load " SDK_LIB_NAME "\n" );
        return -1;
    }

    // Initialize the API.
    err = PrlApi_InitEx(PARALLELS_API_VER, PAM_SERVER, 0, 0);

    if (PRL_FAILED(err))
    {
        fprintf(stderr, "PrlApi_InitEx returned with error: %s.\n",
            prl_result_to_string(err));
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Create a server handle (PHT_SERVER).
    err = PrlSrv_Create(&hServer);
    if (PRL_FAILED(err))
    {
        fprintf(stderr, "PrlSvr_Create failed, error: %s",
            prl_result_to_string(err));
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Log in (asynchronous call).
    hJob = PrlSrv_LoginLocal(hServer, NULL, NULL, PSL_NORMAL_SECURITY);

    // Wait for a maximum of 10 seconds for
    // the job to complete.
    err = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(err))
    {
        fprintf(stderr,
            "PrlJob_Wait for PrlSrv_Login returned with error: %s\n",
            prl_result_to_string(err));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
```

```
        SdkWrap_Unload();
        return -1;
    }

    // Analyze the result of PrlSrv_Login.
    err = PrlJob_GetRetCode(hJob, &nJobReturnCode);

    // First, check PrlJob_GetRetCode success/failure.
    if (PRL_FAILED(err))
    {
        fprintf(stderr, "PrlJob_GetRetCode returned with error: %s\n",
            prl_result_to_string(err));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Now check the Login operation success/failure.
    if (PRL_FAILED(nJobReturnCode))
    {
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        printf("Login job returned with error: %s\n",
            prl_result_to_string(nJobReturnCode));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }
    else
    {
        printf( "Login was successful.\n" );
    }

    return 0;
}


// Log off and deinitializes the SDK library.
//
PRL_RESULT LogOff(PRL_HANDLE &hServer)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;

    PRL_RESULT err = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Log off.
    hJob = PrlSrv_Logoff(hServer);
    err = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(err))
    {
        fprintf(stderr, "PrlJob_Wait for PrlSrv_Logoff returned error: %s\n",
            prl_result_to_string(err));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
```

```
        SdkWrap_Unload();
        return -1;
    }

    // Get the Logoff operation return code.
    err = PrlJob_GetRetCode(hJob, &nJobReturnCode);

    // Check the PrlJob_GetRetCode success/failure.
    if (PRL_FAILED(err))
    {
        fprintf(stderr, "PrlJob_GetRetCode failed for PrlSrv_Logoff with error: %s\n",
            prl_result_to_string(err));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Report success or failure of PrlSrv_Logoff.
    if (PRL_FAILED(nJobReturnCode))
    {
        fprintf(stderr, "PrlSrv_Logoff failed with error: %s\n",
            prl_result_to_string(nJobReturnCode));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }
    else
    {
        printf( "Logoff was successful.\n" );
    }

    // Free handles that are no longer required.
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);

    // De-initialize the Virtuozzo API, and unload the SDK.
    PrlApi_Deinit();
    SdkWrap_Unload();

    return 0;
}
```

Host Operations

This chapter describes the common tasks that can be performed on the host computer.

Obtaining Host Configuration Information

The Virtuozzo C API provides a set of functions to retrieve detailed information about a host computer. This includes:

- CPU(s) - number of, mode, model, speed.

- Devices - disk drives, network interfaces, ports, sound.
- Operating system - type, version, etc.
- Memory (RAM) size.

To retrieve this information, first obtain a handle of type `PHT_SERVER_CONFIG` and then use its functions to get information about a particular resource. The following sample function demonstrates how it is accomplished. The function accepts the `hServer` parameter which is a server handle. For the example on how to obtain a server handle, see **Obtaining Server Handle and Logging In** (p. 20).

```
PRL_RESULT GetHostConfig(PRL_HANDLE hServer)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;
    PRL_HANDLE hHostConfig = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // An asynchronous call that obtains a handle
    // of type PHT_SERVER_CONFIG.
    hJob = PrlSrv_GetSrvConfig(hServer);

    // Wait for the job to complete.
    ret = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        return -1;
    }

    // Analyze the result of PrlSrv_GetSrvConfig.
    ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        return -1;
    }
    // Get the job return code.
    if (PRL_FAILED(nJobReturnCode))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        return -1;
    }

    // Get job result.
    ret = PrlJob_GetResult(hJob, &hJobResult);
    PrlHandle_Free(hJob);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Get the PHT_SERVER_CONFIG handle.
```

```
    ret = PrlResult_GetParam(hJobResult, &hHostConfig);
    PrlHandle_Free(hJobResult);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Obtain the individual configuration setting.
    printf("\nHost Configuration Information: \n\n");

    // Get CPU count.
    PRL_UINT32 nCPUcount = 0;
    ret = PrlSrvCfg_GetCpuCount(hHostConfig, &nCPUcount);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "Error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hHostConfig);
        return -1;
    }

    printf("CPUs: %d\n", nCPUcount);

    // Get host OS type.
    PRL_HOST_OS_TYPE nHostOsType;
    ret = PrlSrvCfg_GetHostOsType(hHostConfig, &nHostOsType);

    // if (PRL_FAILED(ret)) { handle the error... }

    printf("OS Type: %d\n", nHostOsType);

    // Get host RAM size.
    PRL_UINT32 nHostRamSize;
    ret = PrlSrvCfg_GetHostRamSize(hHostConfig, &nHostRamSize);

    // if (PRL_FAILED(ret)) { handle the error... }

    printf("RAM: %d MB\n", nHostRamSize);

    // Get the network adapter info.
    // First get the net adapter count.
    PRL_UINT32 nNetAdaptersCount = 0;
    ret = PrlSrvCfg_GetNetAdaptersCount(hHostConfig,
                                &nNetAdaptersCount);
    // if (PRL_FAILED(ret)) { handle the error... }

    // Now iterate through the list and get the info
    // about each adapter.
    printf("\n");
    for (PRL_UINT32 i = 0; i < nNetAdaptersCount; ++i)
    {
        printf("Net Adapter %d\n", i+1);

        // Obtains a handle of type PHT_HW_NET_ADAPTER.
        PRL_HANDLE phDevice = PRL_INVALID_HANDLE;
        ret = PrlSrvCfg_GetNetAdapter(hHostConfig, i, &phDevice);

        // Get adapter type (physical, virtual).
        PRL_HW_INFO_NET_ADAPTER_TYPE nNetAdapterType;
        ret = PrlSrvCfgNet_GetNetAdapterType(phDevice,
```

```
                                        &nNetAdapterType);
        printf("Type: %d\n", nNetAdapterType);

        // Get system adapter index.
        PRL_UINT32 nIndex = 0;
        ret = PrlSrvCfgNet_GetSysIndex(phDevice, &nIndex);
        printf("Index: %d\n\n", nIndex);
    }

    PrlHandle_Free(hHostConfig);

    return 0;
}
```

Managing Files in Host OS

The following file management operations can be performed using the Virtuozzo C API on the host server:

- Obtaining a directory listing.
- Creating directories.
- Automatically generate unique names for new file system entries.
- Rename file system entries.
- Delete file system entries.

The file management functionality can be accessed through the `PHT_SERVER` handle. The file management functions are prefixed with "PrlSrv_Fs".

Obtaining the host OS directory listing

The directory listing is obtained using the `PrlSrv_FsGetDirEntries` function. The function returns a handle of type `PHT_REMOTE_FILESYSTEM_INFO` containing the information about the specified file system entry and its immediate child entries (if any). The child entries are returned as a list of handles of type `PHT_REMOTE_FILESYSTEM_ENTRY` which is included in the `PHT_REMOTE_FILESYSTEM_INFO` object. The sample function below demonstrates how to obtain a listing for the specified directory. On initial call, the function obtains a list of child entries (files and sub-directories) for the specified directory and is then called recursively for each file system entry returned. On completion, the entire directory tree will be displayed on the screen.

```
// Obtains the entire directory tree in the host OS
// starting at the specified path.
// The "levels" parameter specifies how many levels should the
// function traverse down the directory tree.
PRL_RESULT GetHostDirList(PRL_HANDLE hServer, PRL_CONST_STR path, int levels)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;
    PRL_HANDLE hParentDirectory = PRL_INVALID_HANDLE;
    PRL_HANDLE hChildElement = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;
```

```
// Get directory list from the host.
// The second parameter specifies the absolute
// path for which to get the directory listing.
hJob = PrlSrv_FsGetDirEntries(hServer, path);

// Wait for the job to complete.
ret = PrlJob_Wait(hJob, 1000);
if (PRL_FAILED(ret))
{
    // Handle the error...
    return -1;
}

// Analyze the result of PrlSrv_FsGetDirEntries.
ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hJob);
    return -1;
}
// Check the job return code.
if (PRL_FAILED(nJobReturnCode))
{
    // Handle the error...
    PrlHandle_Free(hJob);
    return -1;
}

// Get job result.
ret = PrlJob_GetResult(hJob, &hJobResult);
PrlHandle_Free(hJob);
if (PRL_FAILED(ret))
{
    // Handle the error...
    return -1;
}

// Get a handle to the parent directory.
// This is the directory that we specified in the
// PrlSrv_FsGetDirEntries call above.
ret = PrlResult_GetParam(hJobResult, &hParentDirectory);
PrlHandle_Free(hJobResult);
if (PRL_FAILED(ret))
{
    // Handle the error...
    return -1;
}

// Get parameter count (the number of child entries).
PRL_UINT32 nParamCount = 0;
ret = PrlFsInfo_GetChildEntriesCount(hParentDirectory, &nParamCount);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hJob);
    return -1;
}

// Iterate through the list obtaining
```

```
    // a handle of type PHT_REMOTE_FILESYSTEM_ENTRY
    // for each child element of the parent directory.
    for (PRL_UINT32 i = 0; i < nParamCount; ++i)
    {
        // Get a handle to the child element.
        ret = PrlFsInfo_GetChildEntry(hParentDirectory, i, &hChildElement);
        if (PRL_FAILED(ret))
        {
            // Handle the error...
            continue;
        }

        // Get the filesystem element name.
        PRL_CHAR sBuf[1024];
        PRL_UINT32 nBufSize = sizeof(sBuf);
        ret = PrlFsEntry_GetAbsolutePath(hChildElement, sBuf, &nBufSize);
        if (PRL_FAILED(ret))
        {
            // Handle the error...
            PrlHandle_Free(hChildElement);
            continue;
        }

        printf("%s\n", sBuf);
        PrlHandle_Free(hChildElement);

        // Recursive call. Obtains directory listing for
        // the entry returned in this iteration.
        if (levels > 0 || levels <= -1)
        {
            int count = levels - 1;
            GetHostDirList(hServer, sBuf, count);
        }
    }
    PrlHandle_Free(hParentDirectory);
    PrlHandle_Free(hJob);

    return PRL_ERR_SUCCESS;
}
```

Obtaining Problem Report

If you are experiencing a problem with a virtual server, you can obtain a problem report from the Virtuozzo host. The report can then be sent to the technical support for evaluation. A problem report contains technical data about your Virtuozzo installation, log data, and other technical details that can be used to determine the source of the problem and to develop a solution. The following example demonstrates how to obtain the report.

```
PRL_RESULT GetProblemReport(PRL_HANDLE hServer)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Get problem report from the host.
    hJob = PrlSrv_GetProblemReport(hServer);
```

```
    // Wait for the job to complete.
    ret = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Analyze the result of PrlSrv_GetProblemReport.
    ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        return -1;
    }
    // Check the job return code.
    if (PRL_FAILED(nJobReturnCode))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        return -1;
    }

    // Get job result.
    ret = PrlJob_GetResult(hJob, &hJobResult);
    PrlHandle_Free(hJob);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Get the string containing the report data.
    // First, get the required buffer size.
    PRL_UINT32 nBufSize = 0;
    ret = PrlResult_GetParamAsString(hJobResult, NULL, &nBufSize);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJobResult);
        return -1;
    }

    // Second, initialize the buffer and get the report.
    char* sReportData =(PRL_STR)malloc(nBufSize);
    ret = PrlResult_GetParamAsString(hJobResult, sReportData, &nBufSize);

    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJobResult);
        return ret;
    }


    printf("%s", sReportData);
    PrlHandle_Free(hJobResult);
    free(sReportData);

    return 0;
```

```
}
```

Virtual Server Operations

This chapter describes the common tasks that can be performed on virtual servers.

Virtuozzo Virtual Machines vs. Virtuozzo Containers

With Virtuozzo, you have a choice of creating and running two types of virtual servers: Virtuozzo Virtual Machines and Virtuozzo Containers. While the two server types use different virtualization technologies (Hypervisor and Operating System Virtualization respectively), the Virtuozzo C API can be used to manage both server types transparently. This means that the same set of handles and functions (with some exceptions) is used to perform operations on both Virtual Machines and Containers.

The following list provides an overview of the specifics and differences when using the API to manage Virtual Machines or Containers:

- When obtaining the list of the available virtual servers with the `PrlSrv_GetVmList` function, the result set will contain the servers of both types. When iterating through the list and obtaining an individual server configuration data, you can determine the server type using the `PrlVmCfg_GetVmType` function. The function accepts a handle of type `PHT_VM_CONFIGURATION` and an [out] pointer of type `PRL_VM_TYPE`, which is an enumeration with `PVT_VM` (Virtual Machine) and `PVT_CT` (Container) enumerators. Once you know the server type, you can perform the desired server management tasks accordingly.
- The majority of handles and functions operate on both virtual server types. A function call and input/output parameters don't change. Some handles and functions are Virtual Machine specific or Container specific. In the C API Reference documentation such handles and functions have a *Virtuozzo Virtual Machines only* or *Virtuozzo Containers only* note at the beginning of a handle or function description. If a handle or function description doesn't specify the server type, it means that it can operate on both Virtual Machines and Containers.

Listing Available Virtual Servers

Any virtual server operation begins with obtaining a handle of type `PHT_VIRTUAL_MACHINE` identifying the virtual server. Once a handle identifying a virtual server is obtained, its functions can be used to perform a full range of virtual server operations. This sections describes how to obtain a list of handles, each identifying an individual virtual server.

The steps that must be performed to obtain the virtual server list are:

1. Log in to the Virtuozzo host and obtain a handle of type `PHT_SERVER`. See **Obtaining Server Handle and Logging In** (p. 20) for more info and code samples.
2. Call `PrlSrv_GetVmList`. This is an asynchronous function that returns a handle of type `PHT_JOB`.

3. Call `PrlJob_GetResults` passing the `PHT_JOB` object obtained in step 2. This function returns a handle of type `PHT_RESULT` containing the virtual server list.
4. Free the job handle using `PrlHandle_Free` as it is no longer needed.
5. Call `PrlResult_GetParamsCount` to determine the number of virtual servers contained in the `PHT_RESULT` object.
6. Call the `PrlResult_GetParamByIndex` function in a loop passing an index from 0 to the total virtual server count. The function obtains a handle of type `PHT_VIRTUAL_MACHINE` containing information about an individual virtual server.
7. Use functions of the `PHT_VIRTUAL_MACHINE` object to obtain the virtual server information. For example, use `PrlVmCfg_GetName` to obtain the virtual server name.
8. Free the virtual server handle using `PrlHandle_Free`.
9. Free the result handle using `PrlHandle_Free`.

The following sample function implements the steps described above.

```
PRL_RESULT GetVmList(PRL_HANDLE hServer)
{
    // Variables for handles.
    PRL_HANDLE hJob = PRL_INVALID_HANDLE; // job handle
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE; // job result

    // Variables for return codes.
    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Get the list of the available virtual servers.
    hJob = PrlSrv_GetVmList(hServer);

    // Wait for a maximum of 10 seconds for PrlSrv_GetVmList.
    ret = PrlJob_Wait(hJob, 10000);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr,
            "PrlJob_Wait for PrlSrv_GetVmList returned with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        return ret;
    }

    // Check the results of PrlSrv_GetVmList.
    ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlJob_GetRetCode returned with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        return ret;
    }

    if (PRL_FAILED(nJobReturnCode))
    {
        fprintf(stderr, "PrlSrv_GetVmList returned with error: %s\n",
            prl_result_to_string(ret));
```

```
        PrlHandle_Free(hJob);
        return ret;
    }

    // Get the results of PrlSrv_GetVmList.
    ret = PrlJob_GetResult(hJob, &hJobResult);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlJob_GetResult returned with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        return ret;
    }

    // Handle to the result object is available,
    // job handle is no longer needed, so free it.
    PrlHandle_Free(hJob);

    // Iterate through the results (list of virtual servers returned).
    PRL_UINT32 nParamsCount = 0;
    ret = PrlResult_GetParamsCount(hJobResult, &nParamsCount);

    printf("\nVirtual Servers:\n");

    for (PRL_UINT32 i = 0; i < nParamsCount; ++i)
    {
        PRL_HANDLE hVm = PRL_INVALID_HANDLE; // virtual server handle

        // Get a handle to the result at index i.
        PrlResult_GetParamByIndex(hJobResult, i, &hVm);

        // Now that we have a handle of type PHT_VIRTUAL_MACHINE,
        // we can use its functions to retrieve or to modify the
        // virtual server information.
        // As an example, we will get the virtual server name.
        char szVmNameReturned[1024];
        PRL_UINT32 nBufSize = sizeof(szVmNameReturned);

        ret = PrlVmCfg_GetName(hVm, szVmNameReturned, &nBufSize);

        if (PRL_FAILED(ret))
        {
            printf("PrlVmCfg_GetName returned with error (%s)\n",
                prl_result_to_string(ret));
        }
        else
        {
            printf(" (%d) %s\n\n", i+1, szVmNameReturned);
        }

        // Free the virtual server handle.
        PrlHandle_Free(hVm);
    }

    return PRL_ERR_SUCCESS;
}
```

Searching for Virtual Servers by Name

This section contains am example of how to obtain a handle of type `PHT_VIRTUAL_MACHINE` identifying the virtual server using the virtual server name as a search parameter. We will use the sample as a helper function in the later section of this guide that demonstrate how to perform operations on virtual servers. The sample is based on the code provided in **Listing Available Virtual Servers** (p. 30).

```
// Obtains a handle of type PHT_VIRTUAL_MACHINE using the
// virtual server name as a search parameter.
// Parameters
//  hServer: A handle of type PHT_SERVER.
//  sVmName: The name of the virtual server.
//  hVm: [out] A handle of type PHT_VIRTUAL_MACHINE
//             identifying the virtual server.
PRL_RESULT GetVmByName(PRL_HANDLE hServer, PRL_STR sVmName, PRL_HANDLE &hVm)
{
    PRL_HANDLE hResult = PRL_INVALID_HANDLE;
    PRL_RESULT nJobResult = PRL_INVALID_HANDLE;

    // Get a list of available virtual servers.
    PRL_HANDLE hJob = PrlSrv_GetVmList(hServer);

    PRL_RESULT ret = PrlJob_Wait(hJob, 10000);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        return ret;
    }

    // Check the results of PrlSrv_GetVmList.
    ret = PrlJob_GetRetCode(hJob, &nJobResult);
    if (PRL_FAILED(nJobResult))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        return ret;
    }

    // Get the results of PrlSrv_GetVmList.
    ret = PrlJob_GetResult(hJob, &hResult);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        return ret;
    }

    PrlHandle_Free(hJob);

    // Iteratre through the results (list of virtual servers returned).
    PRL_UINT32 nParamsCount = 0;
    ret = PrlResult_GetParamsCount(hResult, &nParamsCount);
```

```
    for (PRL_UINT32 i = 0; i < nParamsCount; ++i)
    {
        // Get a handle to result i.
        PrlResult_GetParamByIndex(hResult, i, &hVm);

        // Get the name of the virtual server for result i.
        char vm_name[1024];
        PRL_UINT32 nBufSize = sizeof(vm_name);

        ret = PrlVmCfg_GetName(hVm, vm_name, &nBufSize);

        if (PRL_FAILED(ret))
        {
            // Handle the error...
            return PRL_ERR_FAILURE;
        }

        // If the name of the virtual server at this index is equal to sVmName,
        // then this is the handle we need.
        if (strcmp(sVmName, vm_name) == 0)
        {
            PrlHandle_Free(hResult);
            return PRL_ERR_SUCCESS;
        }

        // It's not the virtual server being searched for, so free the handle to it.
        PrlHandle_Free(hVm);
    }

    // The specified virtual server was not found.
    PrlHandle_Free(hResult);

    return PRL_ERR_NO_DATA;
}
```

Obtaining Virtual Server Configuration Information

The virtual server configuration information is obtained using functions of the `PHT_VM_CONFIGURATION` object. The functions are prefixed with `PrlVmCfg_`. To use the functions, a handle of type `PHT_VM_CONFIGURATION` must first be obtained from the virtual server object (a handle of type `PHT_VIRTUAL_MACHINE`) using the `PrlVm_GetConfig` function. The following example shows how to obtain the virtual server name, guest operating system type and version, RAM size, HDD size, and CPU count. To obtain the virtual server handle (`hVm` input parameter), use the helper function described in **Searching for Virtual Servers by Name** (p. 33).

```
PRL_RESULT GetVmConfig(PRL_HANDLE hVm)
{
    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;

    // Obtain the PHT_VM_CONFIGURATION handle.
    PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;

    ret = PrlVm_GetConfig(hVm, &hVmCfg);

    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return ret;
```

```
    }

    // Get the virtual server name.
    PRL_STR szVmName;
    PRL_UINT32 nVmNameSize = 0;

    // Call once with NULL (PRL_INVALID_HANDLE) to get the size
    // of the buffer to allocate for the VM name.
    PrlVmCfg_GetName(hVmCfg, PRL_INVALID_HANDLE, &nVmNameSize);

    // Allocate memory for the VM name.
    szVmName = (PRL_STR)malloc(nVmNameSize);

    // Get the virtual server name.
    PrlVmCfg_GetName(hVmCfg, szVmName, &nVmNameSize);
    printf("Virtual server name: %s\n", szVmName);

    free(szVmName);

    // Get the OS type.
    PRL_UINT32 nOsType = 0;
    PrlVmCfg_GetOsType(hVmCfg, &nOsType);

    char* sOsTypeName;
    switch (nOsType)
    {
        case PVS_GUEST_TYPE_WINDOWS:
            sOsTypeName = "Windows";
            printf("OS Type: %s\n", PVS_GUEST_TYPE_NAME_WINDOWS);
            break;
        case PVS_GUEST_TYPE_LINUX:
            printf("OS Type: %s\n", PVS_GUEST_TYPE_NAME_LINUX);
            break;
        case PVS_GUEST_TYPE_MACOS:
            printf("OS Type: %s\n", PVS_GUEST_TYPE_NAME_MACOS);
            break;
        case PVS_GUEST_TYPE_FREEBSD:
            printf("OS Type: %s\n", PVS_GUEST_TYPE_NAME_FREEBSD);
            break;
        default:
            printf("OS Type: %s: %d\n", "Other OS Type: ", nOsType);
    }

    // Get the OS version.
    PRL_UINT32 nOsVersion = 0;
    PrlVmCfg_GetOsVersion(hVmCfg, &nOsVersion);
    printf("OS Version: %s\n", PVS_GUEST_TO_STRING(nOsVersion));

    // Get RAM size.
    PRL_UINT32 nRamSize = 0;
    PrlVmCfg_GetRamSize(hVmCfg, &nRamSize);
    printf("RAM size: %dMB\n", nRamSize);

    // Get default HDD size.
    PRL_UINT32 nDefaultHddSize = 0;
    PrlVmCfg_GetDefaultHddSize(nOsVersion, &nDefaultHddSize);
    printf("Default HDD size: %dMB\n", nDefaultHddSize);

    // Get CPU count.
    PRL_UINT32 nCpuCount = 0;
    PrlVmCfg_GetCpuCount(hVmCfg, &nCpuCount);
```

```
    printf("Number of CPUs: %d\n", nCpuCount);

    return PRL_ERR_SUCCESS;
}
```

Determining Virtual Server State

To determine the current state of a virtual server, first obtain a handle to the virtual server as described in **Listing Available Virtual Servers** (p. 30). Then use the `PrlVmCfg_GetState` function to obtain a handle of type `PHT_VM_INFO` and call the `PrlVmInfo_GetState` function to obtain the state information. The function returns the virtual server state as an enumerator from the `VIRTUAL_MACHINE_STATE` enumeration that defines every possible state and transition applicable to a virtual server. The following table lists the available states and transitions:

Enumerator	State/Transition	Description
`VMS_UNKNOWN`	State	Unknown or unsupported state.
`VMS_STOPPED`	State	Virtual server is stopped.
`VMS_STARTING`	Transition	Virtual server is starting.
`VMS_RESTORING`	Transition	Virtual server is being restored from a snapshot.
`VMS_RUNNING`	State	Virtual server is running.
`VMS_PAUSED`	State	Virtual server is paused.
`VMS_SUSPENDING`	Transition	Virtual server is going into "suspended" mode.
`VMS_STOPPING`	Transition	Virtual server is stopping.
`VMS_COMPACTING`	Transition	The Compact operation is being performed on a virtual server.
`VMS_SUSPENDED`	State	Virtual server is suspended.
`VMS_SNAPSHOTING`	Transition	A snapshot of the virtual server is being taken.
`VMS_RESETTING`	Transition	Virtual server is being reset.
`VMS_PAUSING`	Transition	Virtual server is going into the "paused" mode.
`VMS_CONTINUING`	Transition	Virtual server is being brought back up from the "paused" mode.
`VMS_MIGRATING`	Transition	Virtual server is being migrated.
`VMS_DELETING_STATE`	Transition	Virtual server is being deleted.
`VMS_RESUMING`	Transition	Virtual server is being resumed from the "suspended" mode.

The following example demonstrates how obtain state/transition information for the specified virtual server.

```
PRL_RESULT GetVMstate(PRL_HANDLE hVm)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;
    PRL_HANDLE hVmInfo = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
```

```
 PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

// Obtain the PHT_VM_CONFIGURATION handle.
PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
ret = PrlVm_GetConfig(hVm, &hVmCfg);

 // Obtain a handle of type PHT_VM_INFO containing the
 // state information. The object will also contain the
 // virtual server access rights info. We will discuss
 // this functionality later in this guide.
 hJob = PrlVm_GetState(hVmCfg);

 // Wait for the job to complete.
 ret = PrlJob_Wait(hJob, 1000);
 if (PRL_FAILED(ret))
 {
     // Handle the error...
     return -1;
 }

 // Analyze the result of PrlVm_GetState.
 ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
 if (PRL_FAILED(ret))
 {
     // Handle the error...
     PrlHandle_Free(hJob);
     return -1;
 }
 // Check the job return code.
 if (PRL_FAILED(nJobReturnCode))
 {
     // Handle the error...
     PrlHandle_Free(hJob);
     return -1;
 }

 // Get job result.
 ret = PrlJob_GetResult(hJob, &hJobResult);
 PrlHandle_Free(hJob);
 if (PRL_FAILED(ret))
 {
     // Handle the error...
     return -1;
 }

 // Get the PHT_VM_INFO handle.
 ret = PrlResult_GetParam(hJobResult, &hVmInfo);
 PrlHandle_Free(hJobResult);
 if (PRL_FAILED(ret))
 {
     // Handle the error...
     return -1;
 }

 // Get the virtual server state.
 VIRTUAL_MACHINE_STATE vm_state = VMS_UNKNOWN;
 ret = PrlVmInfo_GetState(hVmInfo, &vm_state);
 if (PRL_FAILED(ret))
 {
     // Handle the error...
     PrlHandle_Free(hVmInfo);
```

```
        return -1;
    }
    printf("Status: ");

    switch (vm_state) {
        case VMS_UNKNOWN:
            printf("Unknown state\n");
            break;
        case VMS_STOPPED:
            printf("Stopped\n");
            break;
        case VMS_STARTING:
            printf("Starting...\n");
            break;
        case VMS_RESTORING:
            printf("Restoring...\n");
            break;
        case VMS_RUNNING:
            printf("Running\n");
            break;
        case VMS_PAUSED:
            printf("Paused\n");
            break;
        case VMS_SUSPENDING:
            printf("Suspending...\n");
            break;
        case VMS_STOPPING:
            printf("Stopping...\n");
            break;
        case VMS_COMPACTING:
            printf("Compacting...\n");
            break;
        case VMS_SUSPENDED:
            printf("Suspended\n");
            break;
        default:
            printf("Unknown state\n");
    }
    printf("\n");

    PrlHandle_Free(hVmCfg);
    PrlHandle_Free(hVmInfo);

    return 0;
}
```

Starting, Stopping, Resetting Virtual Servers

Note: When stopping or resetting a virtual server, please be aware of the following important information:

Stopping a virtual server is not the same as performing a guest operating system shutdown operation. When a virtual server is stopped, it is a *cold stop* (i.e. it is the same as turning off the power to a physical computer). Any unsaved data will be lost. However, if the OS in the virtual server supports ACPI (Advanced Configuration and Power Interface) then you can set the second parameter of the `PrlVm_Stop` function to `PRL_FALSE` in which case, the ACPI will be used and the server will be properly shut down.

Resetting a virtual server is not the same as performing a guest operating system restart operation. It is the same as pressing the "Reset" button on a physical box. Any unsaved data will be lost.

The following sample function demonstrates how start, stop, and reset a virtual server.

```
PRL_RESULT StartStopResetVm(PRL_HANDLE hVm, VIRTUAL_MACHINE_STATE action)
{
    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    if (action == VMS_RUNNING)
    {
        // Start the virtual server.
        hJob = PrlVm_Start(hVm);
        printf("Starting the virtual server... \n");
    }
    else if (action == VMS_STOPPED)
    {
        // Stop the virtual server.
        hJob = PrlVm_Stop(hVm, PRL_TRUE);
        printf("Stopping the virtual server... \n");
    }
    else if (action == VMS_RESETTING)
    {
        // Reset the virtual server.
        hJob = PrlVm_Reset(hVm);
        printf("Resetting the virtual server... \n");
    }
    else
    {
        printf ("Invalid action type specified \n");
        return PRL_ERR_FAILURE;
    }

    PrlJob_Wait(hJob, 10000);
    PrlJob_GetRetCode(hJob, &nJobReturnCode);

    if (PRL_FAILED(nJobReturnCode))
    {
        printf ("Error: %s\n", prl_result_to_string(nJobReturnCode));
        PrlHandle_Free(hJob);
        return PRL_ERR_FAILURE;
    }

    return PRL_ERR_SUCCESS;
}
```

Suspending and Pausing Virtual Servers

Suspending a Virtual Server

When a virtual server is suspended, the information about its state is stored in non-volatile memory. A suspended virtual server can resume operating in the same state it was in at the point it was placed into a suspended state. Resuming a virtual server from a suspended state is quicker than starting a virtual server from a stopped state.

To suspend a virtual server, obtain a handle to the virtual server, then call `PrlVm_Suspend`.

The following example will suspend a virtual server called *Windows XP - 01*.

```
const char *szVmName = "Windows XP - 01";

// Get a handle to virtual server with name szVmName.
PRL_HANDLE hVm = GetVmByName((char*)szVmName, hServer);
if (hVm == PRL_INVALID_HANDLE)
{
    fprintf(stderr, "Virtual server \"%s\" was not found.\n", szVmName);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    exit(-1);
}

PRL_RESULT nJobResult;
PRL_HANDLE hJob = PrlVm_Suspend(hVm);
PRL_RESULT ret = PrlJob_Wait(hJob, 1000);
if (PRL_FAIL(ret))
{
    fprintf(stderr, "PrlJob_Wait for PrlVm_Suspend failed. Error: %s",
        prl_result_to_string(ret));
    PrlHandle_Free(hServer);
    PrlHandle_Free(hJob);
    PrlApi_Deinit();
    SdkWrap_Unload();
    exit(-1);
}
PrlJob_GetRetCode(hJob, &nJobResult);
if (PRL_FAILED(nJobResult))
{
    fprintf(stderr, "PrlVm_Suspend failed with error: %s\n",
        prl_result_to_string(nJobResult));
    PrlHandle_Free(hVm);
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}
```

A suspended virtual server can be stopped completely (placed into a "stopped" state) using the `PrlVm_DropSuspendedState` function.

Pausing a Virtual Server

Pausing a virtual server will pause execution of the virtual server. This can be achieved using `PrlVm_Pause`. `PrlVm_Pause` takes two parameters: a handle to the virtual server, and a boolean value indicating if ACPI should be used. The above example could be modified to pause a virtual server by replacing the line:

```
PRL_HANDLE hJob = PrlVm_Suspend(hVm);
```

with:

```
PRL_HANDLE hJob = PrlVm_Pause(hVm, PRL_FALSE);
```

It would also be necessary to change the error messages accordingly.

Resuming / Continuing a Virtual Server

A suspended or paused virtual server can be restarted using `PrlVm_Start`. Alternatively, `PrlVm_Resume` can be used to resume execution of a suspended virtual server.

Dropping Suspended State

A suspended virtual server can be shut down using `PrlVm_DropSuspendedState`. If this is used, any unsaved data will be lost.

Creating New Virtual Server

The first step in creating a new virtual server is to create a blank virtual server and register it with the Virtuozzo host. A blank virtual server is the equivalent of a hardware box with no operating system installed on the hard drive. Once a blank virtual server is created and registered, it can be powered on and an operating system can be installed on it.

In this section, we will discuss how to create a typical virtual server for a particular OS type using a sample configuration. By using this approach, you can easily create a virtual server without knowing all of the little details about configuring a virtual server for a particular operating system type.

The steps involved in creating a typical virtual server are:

1. Obtain a new handle of type `PHT_VIRTUAL_MACHINE` using the `PrlSrv_CreateVm` function. The handle will identify our new virtual server.
2. Obtain a handle of type `PHT_VM_CONFIGURATION` by calling the `PrlVm_GetConfig` function. The handle is used for manipulating virtual server configuration settings.
3. Set the default configuration parameters based on the version of the OS that you will later install in the virtual server. This step is performed using the `PrlVmCfg_SetDefaultConfig` function. You supply the version of the target OS, and the function will generate the appropriate configuration parameters automatically. The OS version parameter value is specified using predefined macros. The names of the macros are prefixed with `PVS_GUEST_VER_`. You can find the macro definitions in the **C API Reference** guide or in the `PrlOses.h` file. In addition to the OS information, the `PrlVmCfg_SetDefaultConfig` function allows to specify the physical host configuration which will be used to connect the virtual devices inside a virtual server to their physical counterparts. The devices include floppy disk drive, CD drive, serial and parallel ports, sound card, etc. To connect the available host devices, obtain a handle of type `PHT_SERVER_CONFIG` (physical host configuration) using the `PrlSrv_GetSrvConfig` function. The handle should then be passed to `PrlVmCfg_SetDefaultConfig` together with OS information and other parameters. If you don't want to connect the devices, set the `hSrvConfig` parameter to `PRL_INVALID_HANDLE`.
4. Choose a name for the new virtual server and set it using the `PrlVmCfg_SetName` function.

5 Modify some of the default configuration parameters if needed. For example, you may want to modify the hard disk image type and size, the amount of memory available to the server, and the networking options. You will have to obtain an appropriate handle for the type of the parameter that you would like to modify and call one of its functions to perform the modification. The code sample below shows how to modify some of the default values.

6 Create and register the new server using the `PrlVm_Reg` function. This step will create the necessary virtual server files on the host and register the server with the Virtuozzo host. The directory containing the virtual server files will have the same name as the virtual server name. The directory will be created in the default location for this Virtuozzo host. If you would like to create the virtual server directory in a different location, you may specify the desired parent directory name and path.

The following sample demonstrates how to create a new virtual server. The sample assumes that the client program has already obtained a server object handle (`hServer`) and performed the login operation.

```
PRL_HANDLE hVm = PRL_INVALID_HANDLE;
PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
PRL_HANDLE hResult = PRL_INVALID_HANDLE;
PRL_RESULT nJobRetCode;
PRL_RESULT ret;

// Obtain a new virtual server handle.
ret = PrlSrv_CreateVm(hServer, &hVm);
if (PRL_FAILED(ret))
{
    // Error handling goes here...
    return ret;
}

// Get the host config info.
hJob = PrlSrv_GetSrvConfig(hServer);
ret = PrlJob_Wait(hJob, 10000);

// Check the return code of PrlSrv_GetSrvConfig.
PrlJob_GetRetCode(hJob, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
{
    fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hVm);
    return nJobRetCode;
}

// Get a handle to the object containing the result of PrlSrv_GetSrvConfig,
// and then get a hosts configuration handle from it.
ret = PrlJob_GetResult(hJob, &hResult);
PRL_HANDLE hSrvCfg = PRL_INVALID_HANDLE;
PrlResult_GetParam(hResult, &hSrvCfg);

// Free job and result handles.
PrlHandle_Free(hJob);
PrlHandle_Free(hResult);

// Now that we have the host configuration data,
// we can set the default configuration for the new virtual server.
ret = PrlVm_GetConfig(hVm, &hVmCfg);
```

```
ret = PrlVmCfg_SetDefaultConfig(
                hVmCfg,                 // VM config handle.
                hSrvCfg,                // Host config data.
                PVS_GUEST_VER_WIN_2003, // Target OS version.
                PRL_TRUE);              // Create and connect devices.

if (PRL_FAILED(ret))
{
    fprintf(stderr, "Error: %s\n", prl_result_to_string(ret));
    PrlHandle_Free(hSrvCfg);
    PrlHandle_Free(hVmCfg);
    PrlHandle_Free(hVm);
    return ret;
}

PrlHandle_Free(hSrvCfg);

// Set the virtual server name.
ret = PrlVmCfg_SetName(hVmCfg, "My Windows Server 2003");

// The following two calls demonstrate how to modify
// some of the default values of the virtual server configuration.
// These calls are optional. You may remove them to use the default values.
//

// Set RAM size for the server to 256 MB.
ret = PrlVmCfg_SetRamSize(hVmCfg, 256);

// Set virtual hard disk size to 20 GB.
// First, get the handle to the hard disk object using the
// PrlVmCfg_GetHardDisk function. The index of 0 is used
// because the default configuration has just one virtual hard disk.
// After that, use the handle to set the disk size.
PRL_HANDLE hHDD = PRL_INVALID_HANDLE;
ret = PrlVmCfg_GetHardDisk(hVmCfg, 0, &hHDD);
ret = PrlVmDevHd_SetDiskSize(hHDD, 20000);

// Create and register the server with Virtuozzo.
// This is an asynchronous call. Returns a job handle.
hJob = PrlVm_Reg(hVm,        // VM handle.
                 "",         // VM root directory (using default).
                 PRL_TRUE);  // Using non-interactive mode.

// Wait for the operation to complete.
ret = PrlJob_Wait(hJob, 10000);

// Check the return code of PrlVm_Reg.
PrlJob_GetRetCode(hJob, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
{
    fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hVmCfg);
    PrlHandle_Free(hVm);
    return nJobRetCode;
}

// Delete handles.
PrlHandle_Free(hJob);
PrlHandle_Free(hVmCfg);
PrlHandle_Free(hVm);
```

Searching for Virtual Servers

A host computer may have virtual servers on its hard drive that are not currently registered with the Virtuozzo. This can happen when a virtual server is removed from Virtuozzo registry but its files are kept on the drive, or when a virtual server files are manually copied to the drive from another computer. Virtuozzo C API provides the `PrlSrv_StartSearchVms` function that can be used to find such virtual servers on the specified host at the specified location on the hard drive. The function accepts a string containing a directory name and path and searches the directory and all its subdirectories for unregistered virtual servers. It then returns a list of `PHT_FOUND_VM_INFO` handles, each containing information about an individual virtual server that it finds. You can then decide whether you want to keep the server as-is, register it, or remove it from the hard drive.

Since the search operation may take a long time (depending on the size of the specified directory tree), the `PrlSrv_StartSearchVms` function should be executed using the callback functionality (see **Asynchronous Methods** (p. 11) for details) The callback function will be called for every virtual server found and a single instance of the `PHT_FOUND_VM_INFO` handle will be passed to it. A callback function can receive two types of objects: jobs (`PHT_JOB`) and events (`PHT_EVENT`). In this instance, the information is passed to the callback function as an event of type `PET_DSP_EVT_FOUND_LOST_VM_CONFIG`. To following steps are involved in processing the event inside the callback function:

1. Determine the type of the event using the `PrlHandle_GetType` function. If it is `PET_DSP_EVT_FOUND_LOST_VM_CONFIG` then the data passed to the callback function contains information about an unregistered virtual server. If not, then the event was generated by some other function and contains the data relevant to that function.
2. Use the `PrlEvent_GetParam` function to obtain a handle of type `PHT_EVENT_PARAMETER` (this is a standard event processing step).
3. Use the `PrlEvtPrm_ToHandle` function to obtain a handle of type `PHT_FOUND_VM_INFO` containing the virtual server information.
4. Use functions of the `PHT_FOUND_VM_INFO` object to determine the location of the virtual server files, the virtual server name, guest OS version, and some other information.

The following is an implementation of the steps above:

```
static PRL_RESULT callback(PRL_HANDLE hEvent, PRL_VOID_PTR pUserData)
{
    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_HANDLE_TYPE nHandleType;

    PrlHandle_GetType(hEvent, &nHandleType);

    // If this is a job, release the handle and exit.
    // Normally, we would process this, if we were after
    // a job.
    if (nHandleType == PHT_JOB)
    {
        PrlHandle_Free(hEvent);
        return 0;
    }
```

```
    // If it's not a job, then it is an event (PHT_EVENT).
    // Get the type of the event received.
    PRL_EVENT_TYPE eventType;
    PrlEvent_GetType(hEvent, &eventType);

    // Check the event type. If it's what we are looking for, process it.
    if (eventType == PET_DSP_EVT_FOUND_LOST_VM_CONFIG)
    {
        PRL_UINT32 nParamsCount = 0;

        // this will receive the event parameter handle.
        PRL_HANDLE hParam = PRL_INVALID_HANDLE;

        // The PrlEvent_GetParam function obtains a handle of type
        // PHT_EVENT_PARAMETER.
        ret = PrlEvent_GetParam(hEvent, 0, &hParam);
        if (PRL_FAILED(ret))
        {
            fprintf(stderr, "[4]%.8X: %s\n", ret,
                prl_result_to_string(ret));
            PrlHandle_Free(hParam);
            PrlHandle_Free(hEvent);
            return ret;
        }

        PRL_HANDLE hFoundVmInfo = PRL_INVALID_HANDLE;
        ret = PrlEvtPrm_ToHandle(hParam, &hFoundVmInfo);
        if (PRL_FAILED(ret))
        {
            fprintf(stderr, "[9]%.8X: %s\n", ret,
                prl_result_to_string(ret));
            PrlHandle_Free(hParam);
            PrlHandle_Free(hEvent);
            return ret;
        }

        // Get the virtual server name.
        PRL_CHAR sName[1024];
        PRL_UINT32 nBufSize = sizeof(sName);
        ret = PrlFoundVmInfo_GetName(hFoundVmInfo, sName, &nBufSize);
        printf("VM name: %s\n", sName);

        // Get the name and path of the virtual server directory.
        PRL_CHAR sPath[1024];
        nBufSize = sizeof(sPath);
        ret = PrlFoundVmInfo_GetConfigPath(hFoundVmInfo, sPath, &nBufSize);
        printf("Path: %s\n\n", sPath);

        PrlHandle_Free(hFoundVmInfo);
        PrlHandle_Free(hEvent);
        return 0;
    }
    // The received event handler MUST be freed.
    PrlHandle_Free(hEvent);
}
```

To begin the search operation, place the following code into your main program:

```
PRL_HANDLE hJob = PRL_INVALID_HANDLE;
PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;
```

```
// Register the event handler.
PrlSrv_RegEventHandler(hServer, &callback, NULL);

// Create a string list object and populate it
// with the name and path of the directory to search.
PRL_HANDLE hStringList = PRL_INVALID_HANDLE;
ret = PrlApi_CreateStringsList(&hStringList );
ret = PrlStrList_AddItem(hStringList, "/Users/Shared/Parallels/");

// Begin the search operation.
hJob = PrlSrv_StartSearchVms(hServer, hStringList);
PrlHandle_Free(hJob);
```

In order for the callback function to be called, your program should have a global loop (the program never exits on its own). The callback function will be called as soon as the first virtual server is found. If there are no unregistered virtual servers in the specified directory tree, then the function will never be called as a `PET_DSP_EVT_FOUND_LOST_VM_CONFIG` event (it will still be called at least once as a result of the started job and will receive the job object but this and possibly other callback invocations are irrelevant in the context of this example).

Receiving the search results synchronously

It is also possible to use this function synchronously using the `PrlJob_Wait` function. In this case, the information is returned as a list of `PHT_FOUND_VM_INFO` objects contained in the job object returned by the `PrlSrv_StartSearchVms` function. The following example demonstrates how to call the function and to process results synchronously.

```
PRL_RESULT SearchVMsSample(PRL_HANDLE hServer)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;
    PRL_HANDLE hFoundVmInfo = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Create a string list object and populate it
    // with the name and path of the directory to search.
    PRL_HANDLE hStringList = PRL_INVALID_HANDLE;
    PrlApi_CreateStringsList(&hStringList );
    PrlStrList_AddItem(hStringList, "/Users/Shared/Parallels/");

    // Begin the search operation.
    hJob = PrlSrv_StartSearchVms(hServer, hStringList);

    // Wait for the job to complete.
    ret = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Analyze the result of PrlSrv_StartSearchVms.
    ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
```

```
        // Handle the error...
        PrlHandle_Free(hJob);
        return -1;
    }
    // Check the job return code.
    if (PRL_FAILED(nJobReturnCode))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        return -1;
    }

    // Get job result.
    ret = PrlJob_GetResult(hJob, &hJobResult);
    PrlHandle_Free(hJob);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Iterate through the returned list obtaining a
    // handle of type PHT_FOUND_VM_INFO in each iteration containing
    // the information about an individual virtual server.
    PRL_UINT32 nIndex, nCount;
    PrlResult_GetParamsCount(hJobResult, &nCount);
    for(nIndex = 0; nIndex < nCount ; nIndex++)
    {
        PrlResult_GetParamByIndex(hJobResult, nIndex, &hFoundVmInfo);

        // Get the virtual server name.
        PRL_CHAR sName[1024];
        PRL_UINT32 nBufSize = sizeof(sName);
        ret = PrlFoundVmInfo_GetName(hFoundVmInfo, sName, &nBufSize);
        printf("VM name: %s\n", sName);

        // Get the name and path of the virtual server directory.
        PRL_CHAR sPath[1024];
        nBufSize = sizeof(sPath);
        ret = PrlFoundVmInfo_GetConfigPath(hFoundVmInfo, sPath, &nBufSize);
        printf("Path: %s\n\n", sPath);

        PrlHandle_Free(hFoundVmInfo);
    }
    PrlHandle_Free(hJobResult);
    PrlHandle_Free(hStringList);
}
```

Adding an Existing Virtual Server

A host may have virtual servers that are not registered with Virtuozzo. This can happen if a virtual server was previously removed from the Virtuozzo registry or if the virtual server files were manually copied from a different location. If you know the location of such a virtual server, you can easily register it with Virtuozzo using the `PrlSrv_RegisterVm` function. The function accepts a server handle, name and path of the directory containing the virtual server files, and registers the server.

Note: The `PrlSrv_RegisterVm` function does NOT generate new MAC addresses for the virtual network adapters that already exist in the virtual server. If the server is a copy of another virtual server

then you should set new MAC addresses for its network adapters after you register it. The example at the end of this section demonstrates how this can be accomplished.

The following sample function demonstrates how to add an existing virtual server to Virtuozzo. The function takes a handle of type PHT_SERVER and a string specifying the name and path of the virtual server directory. It registers the virtual server with Virtuozzo and then modifies the MAC address of every virtual network adapter installed in it.

```
PRL_RESULT RegisterExistingVM(PRL_HANDLE hServer, PRL_CONST_STR sVmDirectory)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobRetCode = PRL_ERR_UNINITIALIZED;

    // Register the virtual server.
    hJob = PrlSrv_RegisterVm(
        hServer,
        sVmDirectory,
        PRL_TRUE);  // Using non-interactive mode.

    ret = PrlJob_Wait(hJob, 10000);

    // Check the return code of PrlSrv_RegisterVm.
    PrlJob_GetRetCode(hJob, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        printf("PrlSrv_RegisterVm returned error: %s\n",
            prl_result_to_string(nJobRetCode));
        PrlHandle_Free(hJob);
        return -1;
    }

    // Obtain the virtual server handle from the job object.
    // We will use the handle later to modify the virtual server
    // configuration.
    PRL_HANDLE hVm = PRL_INVALID_HANDLE;
    ret = PrlJob_GetResult(hJob, &hJobResult);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return ret;
    }

    ret = PrlResult_GetParam(hJobResult, &hVm);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return ret;
    }

    PrlHandle_Free(hJob);
    PrlHandle_Free(hJobResult);

    printf("Virtual server '%s' was successfully registered.",
            sVmDirectory);

    // The following code will generate and set a new MAC address
    // for every virtual network adapter that exists in the virtual server.
```

```
// This step is optional and should normally be performed when the virtual
// server is a copy of another virtual server.

PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;

// Begin the virtual server editing operation.
hJobBeginEdit = PrlVm_BeginEdit(hVm);
ret = PrlJob_Wait(hJobBeginEdit, 10000);
PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
{
    fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
    PrlHandle_Free(hJobBeginEdit);
    return nJobRetCode;
}

// Obtain a handle of type PHT_VM_CONFIGURATION containing the
// virtual server configuration data.
PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
ret = PrlVm_GetConfig(hVm, &hVmCfg);
if (PRL_FAILED(ret))
{
    // Handle the error...
    return ret;
}

// Determine the number of the network adapters
// installed in the server.
PRL_UINT32 nCount = PRL_ERR_UNINITIALIZED;
ret = PrlVmCfg_GetNetAdaptersCount(hVmCfg, &nCount);
if (PRL_FAILED(ret))
{
    // Handle the error...
    return ret;
}

// Itereate through the adapter list.
PRL_HANDLE hNetAdapter = PRL_INVALID_HANDLE;
for (PRL_UINT32 i = 0; i < nCount; ++i)
{
    ret = PrlVmCfg_GetNetAdapter(hVmCfg, i, &hNetAdapter);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return ret;
    }

    // Automatically generate new MAC address for the current adapter.
    // The address will be updated in the configuration object.
    ret = PrlVmDevNet_GenerateMacAddr(hNetAdapter);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return ret;
    }
}

// Commit the changes to the virtual server.
hJobCommit = PrlVm_Commit(hVm);
```

```
    // Check the results of the commit operation.
    ret = PrlJob_Wait(hJobCommit, 10000);
    PrlJob_GetRetCode(hJobCommit, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        fprintf(stderr, "Commit error: %s\n", prl_result_to_string(nJobRetCode));
        PrlHandle_Free(hJobCommit);
        return nJobRetCode;
    }

    PrlHandle_Free(hNetAdapter);
    PrlHandle_Free(hVm);
    PrlHandle_Free(hVmCfg);
    PrlHandle_Free(hJobCommit);
    PrlHandle_Free(hJobBeginEdit);

    return 0;
}
```

Cloning Virtual Servers

A new virtual server can also be created by cloning an existing virtual server. The server will be created as exact copy of the source virtual server and will be automatically registered with Virtuozzo. The cloning operation is performed using the `PrlVm_Clone` function. The following parameters must be specified when cloning a virtual server:

1. A valid handle of type `PHT_VIRTUAL_MACHINE` containing information about the source virtual server.
2. A unique name for the new virtual server (the name is NOT generated automatically).
3. The name of the directory where the virtual server files should be created (or an empty string to create the files in the default directory).
4. A boolean value specifying whether to create the new server as a valid virtual server or as a template. `PRL_TRUE` indicates to create a template. `PRL_FALSE` indicates to create a virtual server.

The source virtual server must be registered with Virtuozzo before it can be cloned.

The following sample function demonstrates how to clone an existing virtual server. When testing a function, the `hVm` parameter must contain a valid handle of type `PHT_VIRTUAL_MACHINE` (the source virtual server to clone). On completion, the new virtual server should appear in the list of registered virtual servers.

```
PRL_RESULT CloneVmSample(PRL_HANDLE hVm)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;
    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;

    // Declare and populate variables that
    // will be used as input parameters
    // in the function that clones a VM.

    // Virtual server name.
    // Get the name of the original VM and use
```

```
    // it in the new virtual server name. You can
    // use any name that you like of course.
    char vm_name[1024];
    PRL_UINT32 nBufSize = sizeof(vm_name);
    PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
    ret = PrlVm_GetConfig(hVm, &hVmCfg);
    ret = PrlVmCfg_GetName(hVmCfg, vm_name, &nBufSize);
    char new_vm_name[1024] = "Clone of ";
    strcat(new_vm_name, vm_name);

    // Name of the target directory on the
    // host.
    // Empty string indicates that the default
    // directory should be used.
    PRL_CHAR_PTR new_vm_root_path = "";

    // Virtual server or template?
    // The cloning functionality allows to create
    // a new virtual server or a new template.
    // True indicates to create a template.
    // False indicates to create a virtual server.
    // We are creating a virtual server.
    PRL_BOOL bCreateTemplate = PRL_FALSE;

    // Begin the cloning operation.
    hJob = PrlVm_Clone(hVm, new_vm_name, new_vm_root_path, bCreateTemplate);
    // Wait for the job to complete.
    ret = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        printf("Error: (%s)\n",
                prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hVmCfg);
        return -1;
    }

    // Analyze the result of PrlVm_Clone.
    ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        PrlHandle_Free(hVmCfg);
        return -1;
    }
    // Check the job return code.
    if (PRL_FAILED(nJobReturnCode))
    {
        // Handle the error...
        printf("Error: (%s)\n",
                prl_result_to_string(nJobReturnCode));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hVmCfg);
        return -1;
    }
    PrlHandle_Free(hJob);
    PrlHandle_Free(hVmCfg);
    return 0;
}
```

Deleting Virtual Servers

If a virtual server is no longer needed, it can be removed. There are two options for removing a virtual server:

1. Un-register the virtual server using `PrlVm_Unreg`. This will remove the virtual server from the list of the virtual servers registered with Virtuozzo. Once a virtual server has been unregistered it is not possible to use it. The directory containing the virtual server files will remain on the hard drive of the host computer, and the virtual server can later be re-registered and used.
2. Delete the virtual server using `PrlVm_Delete`. The virtual server will be unregistered, and the directory (or just some of its files that you can specify) will be deleted.

The following example demonstrates unregistering a virtual server. Note that this example uses a function called `GetVmByName` that can be found in the **Listing Available Virtual Servers** section (p. 30).

```
const char *szVmName = "Windows XP - 02";

// Get a handle to virtual server with name szVmName.
PRL_HANDLE hVm = GetVmByName((char*)szVmName, hServer);
if (hVm == PRL_INVALID_HANDLE)
{
    fprintf(stderr, "VM \"%s\"was not found.\n", szVmName);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

// Unregister a virtual server.
PRL_HANDLE hJob = PrlVm_Unreg(hVm);
PRL_RESULT ret = PrlJob_Wait(hJob, 10000);
if (PRL_FAILED(ret))
{
    printf("PrlJob_Wait failed for PrlVm_Unreg. Error returned: %s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hVm);
    PrlHandle_Free(hJob);
    return -1;
}

PrlJob_GetRetCode(hJob, &nJobResult);
if (PRL_FAILED(nJobResult))
{
    printf("PrlVm_Unreg failed. Error returned: %s\n",
        prl_result_to_string(nJobResult));
    PrlHandle_Free(hVm);
    PrlHandle_Free(hJob);
    return -1;
}
```

The following example demonstrates deleting a virtual server and deleting `config.pvs` within the virtual server directory:

```
// Delete a virtual server and a specified file.
PRL_HANDLE hDeviceList = PRL_INVALID_HANDLE;
```

```
PrlApi_CreateStringsList(&hDeviceList);
PrlStrList_AddItem(hDeviceList, "/Users/Shared/Parallels/WinXP02/config.pvs");
hJob = PrlVm_Delete(hVm, hDeviceList);
PrlHandle_Free(hDeviceList);
ret = PrlJob_Wait(hJob, 10000);
if (PRL_FAILED(ret))
{
    printf("PrlJob_Wait failed for PrlVm_Unreg. Error returned: %s\n",
prl_result_to_string(ret));
    PrlHandle_Free(hVm);
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

PrlJob_GetRetCode(hJob, &nJobResult);
if (PRL_FAILED(nJobResult))
{
    printf("PrlVm_Delete failed. Error returned: %s\n",
prl_result_to_string(nJobResult));
    PrlHandle_Free(hVm);
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}
```

To delete the virtual server and the virtual server directory (all files belonging to the virtual server), omit the line:

```
PrlStrList_AddItem(hDeviceList, "/Users/Shared/Parallels/WinXP02/config.pvs");
```

from the above example. **Note that this operation is irreversible**.

Modifying Virtual Server Configuration

The Virtuozzo C API provides a complete set of functions to modify the configuration parameters of an existing virtual server. You can find the list of the available functions in the **Virtuozzo C API Reference** guide by looking at the `PHT_VM_CONFIGURATION` group. Most of the get/set functions in the group allow to obtain and modify the virtual server configuration parameters. Some parameters are handled as objects and require extra steps in getting or setting them. The following subsections describe how to modify the most common configuration parameters and provide code samples. The samples assume that:

- you've already obtained a handle to the server object and logged on to Virtuozzo.
- you've already obtained a handle to the virtual server that you would like to modify.

Note: All operations on virtual server devices (adding, modifying, removing) must be performed on a stopped virtual server. An attempt to modify the device configuration on a running server will result in error.

PrlVm_BeginEdit and PrlVm_Commit Functions

All virtual server configuration changes must begin with the `PrlVm_BeginEdit` and end with the `PrlVm_Commit` call. These two functions are used to detect collisions with other clients trying to modify the configuration settings of the same virtual server.

When `PrlVm_BeginEdit` is called, the Virtuozzo timestamps the beginning of a configuration change(s) operation. It does not lock the server, so other clients can make changes to the same virtual server at the same time. The function will also automatically update your local virtual server object with the current virtual server configuration information. This is done in order to ensure that your local object contains the changes that might have have happened since you obtained the virtual server handle.

When you are done making the changes, you must call the `PrlVm_Commit` function. The first thing that the function will do is verify that the virtual server configuration has not been modified by other client(s) since you called the `PrlVm_BeginEdit` function. If it has been, your changes will be rejected and `PrlVm_Commit` will return with error. In such a case, you will have to reapply your changes. In order to do that, you will have to get the latest configuration using the `PrlVm_GetConfig` function, compare your changes with the latest changes, and make a decision about merging them. Please note that `PrlVm_GetConfig` function will update the configuration data in your current virtual server object and will overwrite all existing data, including the changes that you've made to it. Furthermore, the `PrlVm_BeginEdit` function will also overwrite all existing data (see above). If you don't want to loose your data, save it locally before calling `PrlVm_GetConfig` or `PrlVm_BeginEdit`.

The following example demonstrates how to use the `PrlVm_BeginEdit` and `PrlVm_Commit` functions:

```
PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
PRL_RESULT nJobRetCode = PRL_INVALID_HANDLE;

// Timestamps the beginning of the "transaction".
// Updates the hVm object with current configuration data.
hJobBeginEdit = PrlVm_BeginEdit(hVm);
ret = PrlJob_Wait(hJobBeginEdit, 10000);
PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
{
    fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
    PrlHandle_Free(hJobBeginEdit);
    return nJobRetCode;
}

// The code modifying configuration parameters goes here...

// Commits the changes to the virtual server.
hJobCommit = PrlVm_Commit(hVm);

// Check the results of the commit operation.
ret = PrlJob_Wait(hJobCommit, 10000);
PrlJob_GetRetCode(hJobCommit, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
```

```
{
    fprintf(stderr, "Commit error: %s\n", prl_result_to_string(nJobRetCode));
    PrlHandle_Free(hJobCommit);
    return nJobRetCode;
}
```

Obtaining PHT_VM_CONFIGURATION handle

Before you can use any of the virtual server configuration management functions, you have to obtain a handle of type `PHT_VM_CONFIGURATION`. The handle is obtained from the virtual server object as shown in the following example:

```
PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
ret = PrlVm_GetConfig(hVm, &hVmCfg);
```

Once you have the handle, you can use its functions to manipulate the virtual server configuration settings. As usual, don't forget to free the handle when it is no longer needed.

Server Name, Description, Boot Options

The virtual server name and description modifications are simple. They are performed using a single call for each parameter:

```
// Modify VM name.
ret = PrlVm_GetConfig(hVm, &hVmCfg);
ret = PrlVmCfg_SetName(hVmCfg, "New Name1");

// Modify VM description.
ret = PrlVmCfg_SetDescription(hVmCfg, "My updated VM");
```

To modify the boot options (boot device priority), first make the `PrlVmCfg_GetBootDevCount` call to determine the number of the available devices. Then obtain a handle to each device by making the `PrlVmCfg_GetBootDev` call in a loop. To place a device at the specified position in the boot device priority list, use the `PrlBootDev_SetSequenceIndex` function passing the device handle and the index (0 - first boot device, 1 - second boot device, and so forth).

The following sample illustrates how to make the above modifications.

```
PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
PRL_RESULT nJobRetCode = PRL_INVALID_HANDLE;

// Timestamp the beginning of the transaction.
hJobBeginEdit = PrlVm_BeginEdit(hVm);
ret = PrlJob_Wait(hJobBeginEdit, 10000);
PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
{
    fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
    PrlHandle_Free(hJobBeginEdit);
    return nJobRetCode;
}

// Modify VM name.
ret = PrlVmCfg_SetName(hVmCfg, "New Name1");

// Modify VM description.
ret = PrlVmCfg_SetDescription(hVmCfg, "My updated VM");
```

```
// Modify boot options.
// Set boot device list as follows:
// 0. CD/DVD drive.
// 1. Hard disk.
// 2. Network adapter.
// 3. Floppy disk drive.
// Remove all other devices (if any) from the
// boot devices list for this VM.
// Device count.
PRL_UINT32 nDevCount;
// A handle identifying the device.
PRL_HANDLE hDevice = PRL_INVALID_HANDLE;
// Device type.
PRL_DEVICE_TYPE devType;

// Get the total number of devices.
ret = PrlVmCfg_GetBootDevCount(hVmCfg, &nDevCount);

// Iterate through the device list.
// Get a handle for each available device.
// Set an index for a device in the boot list.
for (int i = 0; i < nDevCount; ++i)
{
    ret = PrlVmCfg_GetBootDev(hVmCfg, i, &hDevice);
    ret = PrlBootDev_GetType(hDevice, &devType);

    if (devType == PDE_OPTICAL_DISK)
    {
        PrlBootDev_SetSequenceIndex(hDevice, 0);
    }
    if (devType == PDE_HARD_DISK)
    {
        PrlBootDev_SetSequenceIndex(hDevice, 1);
    }
    else if (devType == PDE_GENERIC_NETWORK_ADAPTER)
    {
        PrlBootDev_SetSequenceIndex(hDevice, 2);
    }
    else if (devType == PDE_FLOPPY_DISK)
    {
        PrlBootDev_SetSequenceIndex(hDevice, 3);
    }
    else
    {
        PrlBootDev_Remove(hDevice);
    }
}

// Commit the changes.
hJobCommit = PrlVm_Commit(hVm);

// Check the results of the commit operation.
ret = PrlJob_Wait(hJobCommit, 10000);
PrlJob_GetRetCode(hJobCommit, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
{
    fprintf(stderr, "Commit error: %s\n", prl_result_to_string(nJobRetCode));
    PrlHandle_Free(hJobCommit);
    return nJobRetCode;
}
```

RAM Size

The size of the memory available to the virtual server is obtained using the `PrlVmCfg_SetRamSize` function. The first parameter is the virtual server handle and the second parameter is the new RAM size in megabytes:

```
PrlVmCfg_SetRamSize(hVmCfg, 512);
```

Hard Disks

Modifying the size of the existing hard disk image

A virtual server may have more than one virtual hard disk. To select a disk that you would like to modify, first retrieve the list of the available disks, as shown in the following example:

```
PRL_HANDLE hHDD = PRL_INVALID_HANDLE;
PRL_UINT32 nCount;

// Get the number of disks available.
PrlVmCfg_GetHardDisksCount(hVmCfg, &nCount);

// Iterate through the list.
for (PRL_UINT32 i = 0; i < nCount; ++i)
{
    // Obtain a handle to the hard disk object.
    ret = PrlVmCfg_GetHardDisk(hVmCfg, i, &hHDD);

    // The code selecting the desired HDD goes here...
    // {
         // Modify the disk size.
         // The hard disk size is specified in megabytes.
         ret = PrlVmDevHd_SetDiskSize(hHDD, 20000);
    // }
}
```

Adding a new hard disk

In this example, we will add a hard disk to a virtual server. The following options are available:

- You may create new or use an existing image file for your new disk.
- Creating a dynamically expanding or a fixed-size disk. The expanding drive image will be initially created with a size of zero. The space for it will be allocated dynamically on as-needed basis. The space for the fixed-size disk will be allocated fully at the time of creation.
- Choosing the maximum disk size.

Creating a new image file

In the first example, we will create a new disk image and will add it to a virtual server.

```
PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
PRL_RESULT nJobRetCode = PRL_INVALID_HANDLE;

// Timestamp the beginning of the configuration changes operation.
```

```
// The hVm specifies the virtual server that we'll be editing.
//
hJobBeginEdit = PrlVm_BeginEdit(hVm);
ret = PrlJob_Wait(hJobBeginEdit, 10000);
PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
{
    fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
    PrlHandle_Free(hJobBeginEdit);
    return nJobRetCode;
}

// Create a new device handle.
// This will be our new virtual hard disk.
PRL_HANDLE hHDD = PRL_INVALID_HANDLE;
ret = PrlVmCfg_CreateVmDev(
        hVmCfg, // The target virtual server.
        PHT_VIRTUAL_DEV_HARD_DISK, // Device type.
        &hHDD); // Device handle.

// Set disk type to "expanding".
ret = PrlVmDevHd_SetDiskType(hHDD, PHD_EXPANDING_HARD_DISK);

// Set max disk size, in megabytes.
ret = PrlVmDevHd_SetDiskSize(hHDD, 32000);

// This option determines whether the image file will be splitted
// into chunks or created as a single file.
ret = PrlVmDevHd_SetSplitted(hHDD, PRL_FALSE);

// Choose and set the name for the new image file.
// We must set both the "friendly" name and the "system" name.
// For a virtual device, use the name of the new image file in both
// functions. By default, the file will be
// created in the virtual server directory. You may specify a
// full path if you want to place the file in a different
// directory.
//
ret = PrlVmDev_SetFriendlyName(hHDD, "harddisk4.hdd");
ret = PrlVmDev_SetSysName(hHDD, "harddisk4.hdd");

// Set the emulation type.
ret = PrlVmDev_SetEmulatedType(hHDD, PDT_USE_IMAGE_FILE);

// Enable the new disk on successful creation.
ret = PrlVmDev_SetEnabled(hHDD, PRL_TRUE);

// Create the new image file.
hJob = PrlVmDev_CreateImage(hHDD,
            PRL_TRUE, // Do not overwrite if the file exists.
            PRL_TRUE); // Use non-interactive mode.

// Commit the changes.
hJobCommit = PrlVm_Commit(hVm);

// Check the results of the commit operation.
ret = PrlJob_Wait(hJobCommit, 10000);
PrlJob_GetRetCode(hJobCommit, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
{
    fprintf(stderr, "Commit error: %s\n", prl_result_to_string(nJobRetCode));
```

```
    PrlHandle_Free(hJobCommit);
    return nJobRetCode;
}
```

Using an existing image file

In the next example, we will use an existing image file to add a virtual hard disk to a virtual server. The procedure is similar to the one described above, except that you don't have to specify the disk parameters and you don't have to create an image file.

```
// Timestamp the beginning of the configuration changes operation.
// The hVm specifies the virtual server that we'll be editing.
//
hJobBeginEdit = PrlVm_BeginEdit(hVm);
ret = PrlJob_Wait(hJobBeginEdit, 10000);
PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
if (PRL_FAILED(nJobRetCode))
{
    fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
    PrlHandle_Free(hJobBeginEdit);
    return nJobRetCode;
}

// Create a device handle.
PRL_HANDLE hHDD = PRL_INVALID_HANDLE;
ret = PrlVmCfg_CreateVmDev(
               hVmCfg, // Target virtual server.
               PHT_VIRTUAL_DEV_HARD_DISK, // Device type.
               &hHDD); // Device handle.

// In this example, these two functions are used
// to specify the name of the existing image file.
// By default, it will look for the file in the
// virtual server directory. If the file is located
// anywhere else, you must specify the full path here.
//
ret = PrlVmDev_SetFriendlyName(hHDD, "harddisk4.hdd");
ret = PrlVmDev_SetSysName(hHDD, "harddisk4.hdd");

// Set the emulation type.
ret = PrlVmDev_SetEmulatedType(hHDD, PDT_USE_IMAGE_FILE);

// Enable the drive on completion.
ret = PrlVmDev_SetEnabled(hHDD, PRL_TRUE);

// Commit the changes.
hJobCommit = PrlVm_Commit(hVm);
```

If the commit operation is successful, a hard disk will be added to the virtual server and will appear in the list of the available devices.

Network Adapters

When adding a network adapter to a virtual server, you first have to choose a networking mode for it. The following options are available:

- Host-only networking. A virtual server can communicate with the host and other virtual servers, but not with external networks.

- Shared networking. Uses the NAT feature. A virtual server shares the IP address with the host.
- Bridged networking. A virtual adapter in the VM is bound to a network adapter on the host. The virtual server appears as a standalone computer on the network.

Host-Only and Shared Networking

The following sample function illustrates how to add virtual network adapters using the host-only and shared networking (both types are created similarly). The steps are:

1. Call the `PrlVm_BeginEdit` function to mark the beginning of the virtual server editing operation. This step is required when modifying any of the virtual server configuration parameters.
2. Obtain a handle of type `PHT_VM_CONFIGURATION` containing the virtual server configuration information.
3. Create a new virtual device handle of type `PHT_VIRTUAL_DEV_NET_ADAPTER` (virtual network adapter) using the `PrlVmCfg_CreateVmDev` function.
4. Set the desired device emulation type (host or shared) using the `PrlVmDev_SetEmulatedType` function. Virtual network adapter emulation types are defined in the `PRL_VM_DEV_EMULATION_TYPE` enumeration.
5. The MAC address for the adapter will be generated automatically. If needed, you can set the address manually using the `PrlVmDevNet_SetMacAddress` function.
6. Call the `PrlVm_Commit` function to finalize the changes.

```
PRL_RESULT AddNetAdapterHostOrShared(PRL_HANDLE hVm)
{
    PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
    PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
    PRL_RESULT nJobRetCode = PRL_INVALID_HANDLE;
    PRL_UINT32 ret = PRL_ERR_UNINITIALIZED;

    // Timestamp the beginning of the configuration changes operation.
    // The hVm parameter specifies the target virtual server.
    hJobBeginEdit = PrlVm_BeginEdit(hVm);
    ret = PrlJob_Wait(hJobBeginEdit, 10000);
    PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
        PrlHandle_Free(hJobBeginEdit);
        return nJobRetCode;
    }

    // Obtain a handle of type PHT_VM_CONFIGURATION containing
    // the virtual server configuration information.
    ret = PrlVm_GetConfig(hVm, &hVmCfg);
    if (PRL_FAILED(ret))
    {
        // Handle the error.
    }

    // Create a virtual network adapter device handle.
```

```
    PRL_HANDLE hNet = PRL_INVALID_HANDLE;
    ret = PrlVmCfg_CreateVmDev(
            hVmCfg, // The virtual server configuration handle.
            PDE_GENERIC_NETWORK_ADAPTER, // Device type.
            &hNet); // Device handle.

    if (PRL_FAILED(ret))
    {
      // Handle the error.
    }

    // For host-only networking, set the device emulation type
    // to PDT_USE_HOST_ONLY_NETWORK, which is an enumerator from the
    // PRL_VM_DEV_EMULATION_TYPE enumeration.
    // For shared networking, set the device emulation type
    // to PDT_USE_SHARED_NETWORK, which is also an enumerator from
    // the same enumeration.
    // Un-comment one of the following lines (and comment out the other)
    // to set the the desired emulation type.
    PRL_VM_DEV_EMULATION_TYPE pdtType = PDT_USE_HOST_ONLY_NETWORK;
    //PRL_VM_DEV_EMULATION_TYPE pdtType = PDT_USE_SHARED_NETWORK;

    ret = PrlVmDev_SetEmulatedType(hNet, pdtType);
    if (PRL_FAILED(ret))
    {
        // Handle the error.
    }

    // By default, a new device is created disabled.
    // You can set the "connected" and "enabled" properties
    // as desired.
    PrlVmDev_SetConnected(hNet, PRL_TRUE);
    PrlVmDev_SetEnabled(hNet, PRL_TRUE);

    // Commit the changes.
    hJobCommit = PrlVm_Commit(hVm);
    ret = PrlJob_Wait(hJobBeginEdit, 10000);
    PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        // Handle the error.
    }

    // Release all handles.
    PrlHandle_Free(hNet);
    PrlHandle_Free(hVmCfg);
    PrlHandle_Free(hJobBeginEdit);
    PrlHandle_Free(hJobCommit);

    return PRL_ERR_SUCCESS;
}
```

Bridged Networking

Compared to host-only and shared network adapters, adding an adapter using bridged networking involves additional steps. In a bridged networking mode, you are binding the virtual adapter inside a virtual server to an adapter on the host machine. Therefore, you first have to retrieve the list of adapters from the host and select the one you would like to use. The complete procedure of creating an adapter using bridged networking is as follows:

1. Obtain a list of network adapters installed on the host. This steps is preformed using the `PrlSrvCfg_GetNetAdaptersCount`, `PrlSrvCfg_GetNetAdapter`, and `PrlSrvCfgDev_GetName` functions.
2. Begin the virtual server editing operation and create a new network adapter handle as described in the **Host-only and Shared Networking** section (p. 60).
3. Bind the new virtual network adapter to the desired host machine adapter using the `PrlVmDevNet_SetBoundAdapterName` function.
4. Finalize the changes by calling the `PrlVm_Commit` function.

You can also bind a virtual network adapter to the default adapter on the host machine. In this case, you don't have to obtain the list of adapters from the host, so you can skip step 1 (above). In step 3, instead of setting the adapter name, set its index as -1 using the `PrlVmDevNet_SetBoundAdapterIndex` function.

The following are two sample functions that show the implementation of the steps described above. The two functions are similar except that the first one shows how to bind a virtual network adapter to a specific adapter on the host, whereas the second function shows how to bind the virtual adapter to the default host network adapter.

Example 1:

The function accepts a server handle and a virtual server handle. The server handle will be used to obtain the list of network adapters from the host.

```
PRL_RESULT AddNetAdapterBridged(PRL_HANDLE hServer, PRL_HANDLE hVm)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
    PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
    PRL_RESULT nJobRetCode = PRL_INVALID_HANDLE;
    PRL_UINT32 ret = PRL_ERR_UNINITIALIZED;

    // Obtain a list of the network adapters installed on
    // the host.
    // First, obtain a handle containing the
    // host configuration info.
    hJob = PrlSrv_GetSrvConfig(hServer);
    ret = PrlJob_Wait(hJob, 10000);

    PrlJob_GetRetCode(hJob, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        // Handle the error.
    }

    // Get job results.
    ret = PrlJob_GetResult(hJob, &hJobResult);
    if (PRL_FAILED(ret))
    {
        // Handle the error.
    }
```

```
    // server config handle.
    PRL_HANDLE hSrvCfg = PRL_INVALID_HANDLE;

    // counter.
    PRL_UINT32 nCount = PRL_INVALID_HANDLE;

    // Now obtain the actual handle containing the
    // host configuration info.
    PrlResult_GetParam(hJobResult, &hSrvCfg);

    // Get the number of the available adapters from the
    // host configuration object.
    PrlSrvCfg_GetNetAdaptersCount(hSrvCfg, &nCount);

    // Net adapter handle.
    PRL_HANDLE hHostNetAdapter = PRL_INVALID_HANDLE;
    PRL_CHAR chHostAdapterName[1024];

    // Iterate through the list of the adapters.
    for (PRL_UINT32 i = 0; i < nCount; ++i)
    {
        PrlSrvCfg_GetNetAdapter(hSrvCfg, i, &hHostNetAdapter);

        // Get adapter name.
        PRL_CHAR chName[1024];
        PRL_UINT32 nBufSize = sizeof(chName);
        ret = PrlSrvCfgDev_GetName(hHostNetAdapter, chName, &nBufSize);

        // Normally, you would iterate through the entire list
        // and select an adapter to bind the virtual network adapter to.
        // For simplicity, we will simply pick the first one and use it.
        strcpy(chHostAdapterName, chName);
        break;

    }

    // Now that we have the name of the host network adapter,
    // we can add a new virtual network adapter to the virtual server.

    // Timestamp the beginning of the configuration changes operation.
    // The hVm parameter specifies the target virtual server.
    hJobBeginEdit = PrlVm_BeginEdit(hVm);
    ret = PrlJob_Wait(hJobBeginEdit, 10000);
    PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
        PrlHandle_Free(hJobBeginEdit);
        return nJobRetCode;
    }

    // Obtain a handle of type PHT_VM_CONFIGURATION containing
    // the virtual server configuration information.
    ret = PrlVm_GetConfig(hVm, &hVmCfg);
    if (PRL_FAILED(ret))
    {
        // Handle the error.
    }

    // Create a virtual network adapter device handle.
    PRL_HANDLE hNet = PRL_INVALID_HANDLE;
```

```
    ret = PrlVmCfg_CreateVmDev(
            hVmCfg, // The virtual server configuration handle.
            PDE_GENERIC_NETWORK_ADAPTER, // Device type.
            &hNet); // Device handle.

    if (PRL_FAILED(ret))
    {
      // Handle the error.
    }

    // Set the virtual network adapter emulation type (networking type).
    // Bridged networking is set using the PDT_USE_BRIDGE_ETHERNET enumerator
    // from the PRL_VM_DEV_EMULATION_TYPE enumeration.
    ret = PrlVmDev_SetEmulatedType(hNet, PDT_USE_BRIDGE_ETHERNET);
    if (PRL_FAILED(ret))
    {
        // Handle the error.
    }

    // Set the host adapter to which this adapter should be bound.
    PrlVmDevNet_SetBoundAdapterName(hNet, chHostAdapterName);


    // By default, a new device is created disabled.
    // You can set the "connected" and "enabled" properties
    // as desired.
    PrlVmDev_SetConnected(hNet, PRL_TRUE);
    PrlVmDev_SetEnabled(hNet, PRL_TRUE);

    // Commit the changes.
    hJobCommit = PrlVm_Commit(hVm);
    ret = PrlJob_Wait(hJobBeginEdit, 10000);
    PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        // Handle the error.
    }

    // Release all handles.
    PrlHandle_Free(hNet);
    PrlHandle_Free(hVmCfg);
    PrlHandle_Free(hJobBeginEdit);
    PrlHandle_Free(hJobCommit);

    return PRL_ERR_SUCCESS;
}
```

Example 2:

This function shows how to add a virtual network adapter to a virtual server and how to bind it to the default adapter on the host.

```
PRL_RESULT AddNetAdapterBridgedDefault(PRL_HANDLE hVm)
{
    PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
    PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
    PRL_RESULT nJobRetCode = PRL_INVALID_HANDLE;
    PRL_UINT32 ret = PRL_ERR_UNINITIALIZED;

    // Timestamp the beginning of the configuration changes operation.
```

```
    // The hVm parameter specifies the target virtual server.
    hJobBeginEdit = PrlVm_BeginEdit(hVm);
    ret = PrlJob_Wait(hJobBeginEdit, 10000);
    PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        fprintf(stderr, "Error: %s\n", prl_result_to_string(nJobRetCode));
        PrlHandle_Free(hJobBeginEdit);
        return nJobRetCode;
    }

    // Obtain a handle of type PHT_VM_CONFIGURATION containing
    // the virtual server configuration information.
    ret = PrlVm_GetConfig(hVm, &hVmCfg);
    if (PRL_FAILED(ret))
    {
        // Handle the error.
    }

    // Create a virtual network adapter device handle.
    PRL_HANDLE hNet = PRL_INVALID_HANDLE;
    ret = PrlVmCfg_CreateVmDev(
            hVmCfg, // The virtual server configuration handle.
            PDE_GENERIC_NETWORK_ADAPTER, // Device type.
            &hNet); // Device handle.

    if (PRL_FAILED(ret))
    {
      // Handle the error.
    }

    // Set the virtual network adapter emulation type (networking type).
    // Bridged networking is set using the PDT_USE_BRIDGE_ETHERNET enumerator
    // from the PRL_VM_DEV_EMULATION_TYPE enumeration.
    ret = PrlVmDev_SetEmulatedType(hNet, PDT_USE_BRIDGE_ETHERNET );
    if (PRL_FAILED(ret))
    {
        // Handle the error.
    }

    // Set the host adapter index to -1. This will
    // bind the virtual adapter to the default adapter on the
    // host.
    PrlVmDevNet_SetBoundAdapterIndex(hNet, -1);

    // By default, a new device is created disabled.
    // You can set the "connected" and "enabled" properties
    // as desired.
    PrlVmDev_SetConnected(hNet, PRL_TRUE);
    PrlVmDev_SetEnabled(hNet, PRL_TRUE);

    // Commit the changes.
    hJobCommit = PrlVm_Commit(hVm);
    ret = PrlJob_Wait(hJobBeginEdit, 10000);
    PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        // Handle the error.
    }

    // Release all handles.
```

```
    PrlHandle_Free(hNet);
    PrlHandle_Free(hVmCfg);
    PrlHandle_Free(hJobBeginEdit);
    PrlHandle_Free(hJobCommit);

    return PRL_ERR_SUCCESS;
}
```

Managing User Access Rights

User authorization (determining user access rights) is performed using OS-level file access permissions. Essentially, a virtual server is a set of files that a user can read, write, and execute. When determining access rights of a user for a particular virtual server, Virtuozzo looks at the rights the user has on the virtual server files and uses this information to allow or deny privileges.

The Virtuozzo C API contains a `PHT_ACCESS_RIGHTS` object that is used to manage user access rights. A handle to it is obtained using the `PrlVmCfg_GetAccessRights` or the `PrlVmInfo_GetAccessRights` function. The difference between the two function is that `PrlVmInfo_GetAccessRights` takes an additional step: obtaining a handle of type `PHT_VM_INFO` which will also contain the virtual server state information. If user access rights is all you need, you can use the `PrlVmCfg_GetAccessRights` function.

The `PHT_ACCESS_RIGHTS` object provides an easy way of determining access rights for the currently logged in user with the `PrlAcl_IsAllowed` function. The function allows to specify one of the available virtual server tasks (defined in the `PRL_ALLOWED_VM_COMMAND` enumeration) and returns a boolean value indicating whether the user is allowed to perform the task or not. The security is enforced on the server side, so if a user tries to perform a tasks that he/she is not authorized to perform, the access will be denied. You can still use the functionality described here to determine user access rights in advance and use it in accordance with your client application logic.

An administrator of the host has full access rights to all virtual servers. A non-administrative user has full rights to the servers created by him/her and no rights to any other virtual servers by default (these servers will not even be included in the result set when the user requests a list of virtual servers from the host). The host administrator can grant virtual server access privileges to other users when needed. Currently, the privileges can be granted to all existing users only. It is not possible to set access rights for an individual user through the API. The `PrlAcl_SetAccessForOthers` function is used to set access rights. The function takes the `PHT_ACCESS_RIGHTS` object identifying the virtual server and one of the enumerators from the `PRL_VM_ACCESS_FOR_OTHERS` enumerations identifying the access level, which includes view, view and run, full access, and no access. Once again, the function sets access rights for all existing users. To determine the current access level for other users on a particular virtual server, use the `PrlAcl_GetAccessForOthers` function. For the complete set of user access management functions, see the `PHT_ACCESS_RIGHTS` handle description in the **C API Reference** guide.

The following sample function demonstrates how to set virtual server access rights and how to determine access rights on the specified virtual server for the currently logged in user. The function accepts a virtual server handle and operates on the referenced virtual server.

```
PRL_RESULT AccessRightsSample(PRL_HANDLE hVm)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hAccessRights = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Obtain a PHT_VM_CONFIGURATION handle.
    PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
    ret = PrlVm_GetConfig(hVm, &hVmCfg);

    // Obtain the access rights handle (this will be a
    // handle of type PHT_ACCESS_RIGHTS).
    ret = PrlVmCfg_GetAccessRights(hVmCfg, &hAccessRights);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hVmCfg);
        return -1;
    }

    PrlHandle_Free(hVmCfg);

    // Get the VM owner name from the access rights handle.
    PRL_CHAR sBuf[1024];
    PRL_UINT32 nBufSize = sizeof(sBuf);
    ret = PrlAcl_GetOwnerName(hAccessRights, sBuf, &nBufSize);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hAccessRights);
        return -1;
    }
    printf("Owner: %s\n", sBuf);

    // Change the virtual server access rights for other users
    // to PAO_VM_SHARED_ON_VIEW_AND_RUN, which means that the
    // users will be able to see the server in the list and to
    // run it. When this operation completes, we will use
    // PrlAcl_IsAllowed function to determine whether the user
    // is allowed to perform a particular task on the virtual
    // server.
    PRL_VM_ACCESS_FOR_OTHERS access = PAO_VM_SHARED_ON_VIEW_AND_RUN;
    ret = PrlAcl_SetAccessForOthers(hAccessRights, access);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Commit the changes.
    hJob = PrlVm_UpdateSecurity(hVm, hAccessRights);
    ret = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }
    // Analyze the result of PrlVm_UpdateSecurity.
    ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
```

```
    PrlHandle_Free(hJob);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }
    // Check the job return code.
    if (PRL_FAILED(nJobReturnCode))
    {
        // Handle the error...
        return -1;
    }

    // Determine if the current user has the right to
    // start the virtual server.
    PRL_ALLOWED_VM_COMMAND access_level = PAR_VM_START_ACCESS;
    PRL_BOOL isAllowed = PRL_FALSE;
    ret = PrlAcl_IsAllowed(hAccessRights, access_level, &isAllowed);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }
    printf("Can start: %d\n", isAllowed);

    // Determine if the current user has the right to
    // delete the specified virtual server.
    access_level = PAR_VM_DELETE_ACCESS;
    isAllowed = PRL_FALSE;
    ret = PrlAcl_IsAllowed(hAccessRights, access_level, &isAllowed);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }
    printf("Can delete: %d\n", isAllowed);

    PrlHandle_Free(hAccessRights);
    return 0;
}
```

Virtual Server Templates

Templates are virtual servers that cannot be run but can be used to create new virtual servers. Virtual server templates are not different from regular virtual servers except, as was mentioned earlier, that they cannot be run. In fact, you can convert a template to a regular virtual server at any time, just as you can convert a regular virtual server to a template.

The Virtuozzo C API allows to perform the following template-related operations:

- Obtaining a list of the available virtual server templates.
- Creating a virtual server template from scratch.
- Converting a regular virtual server to a template.
- Converting a template to a regular virtual server.
- Creating a new virtual server from a template.

The following subsections describes each operation in detail and provide code examples.

Listing Available Templates

A list of virtual servers and virtual server templates are obtained from the server using the same `PrlSrv_GetVmList` function. A template is identified by calling the `PrlVmCfg_IsTemplate` function which returns a boolean value indicating whether the specified virtual server handle contains information about a regular virtual or a handle. The value of `PRL_TRUE` indicates that the server is a template. The value of `PRL_FALSE` indicates that the server is a regular virtual server. The following sample is identical to the sample provided in the **Listing Available Virtual Servers** (p. 30) section with the exception that it was modified to display only the lists of templates on the screen:

```
PRL_RESULT GetTemplateList(const PRL_HANDLE &hServer)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Get a list of the available virtual servers.
    hJob = PrlSrv_GetVmList(hServer);

    // Wait for a maximum of 10 seconds for PrlSrv_GetVmList.
    ret = PrlJob_Wait(hJob, 10000);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr,
            "PrlJob_Wait for PrlSrv_GetVmList returned with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        return ret;
    }

    // Check the results of PrlSrv_GetVmList.
    ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlJob_GetRetCode returned with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        return ret;
    }

    if (PRL_FAILED(nJobReturnCode))
    {
        fprintf(stderr, "PrlSrv_GetVmList returned with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        return ret;
    }

    // Get the results of PrlSrv_GetVmList.
    ret = PrlJob_GetResult(hJob, &hJobResult);
    if (PRL_FAILED(ret))
    {
```

```
        fprintf(stderr, "PrlJob_GetResult returned with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        return ret;
    }

    // Handle to result object is available,
    // job handle is no longer needed, so free it.
    PrlHandle_Free(hJob);

    // Iterate through the results (list of virtual servers returned).
    PRL_UINT32 nParamsCount = 0;
    ret = PrlResult_GetParamsCount(hJobResult, &nParamsCount);
    for (PRL_UINT32 i = 0; i < nParamsCount; ++i)
    {
        // Virtual server handle
        PRL_HANDLE hVm = PRL_INVALID_HANDLE;

        // Get a handle to result at index i.
        PrlResult_GetParamByIndex(hJobResult, i, &hVm);

        // Obtain the PHT_VM_CONFIGURATION object.
        PRL_HANDLE hVmCfg = PRL_INVALID_HANDLE;
        ret = PrlVm_GetConfig(hVm, &hVmCfg);

        // See if the handle contains information about a template.
        PRL_BOOL isTemplate = PRL_FALSE;
        PrlVmCfg_IsTemplate(hVmCfg, &isTemplate);

        // If this is not a template, proceed to the next
        // virtual server in the list.
        if (isTemplate == PRL_FALSE)
        {
            PrlHandle_Free(hVmCfg);
            PrlHandle_Free(hVm);
            continue;
        }

        // Get the name of the template for result i.
        char szVmNameReturned[1024];
        PRL_UINT32 nBufSize = sizeof(szVmNameReturned);
        ret = PrlVmCfg_GetName(hVmCfg, szVmNameReturned, &nBufSize);
        if (PRL_FAILED(ret))
        {
            printf("PrlVmCfg_GetName returned with error (%s)\n",
                prl_result_to_string(ret));
        }
        else
        {
            printf("Template name: '%s'.\n",
                    szVmNameReturned);
        }

        PrlHandle_Free(hVm);
        PrlHandle_Free(hVmCfg);
    }

    return PRL_ERR_SUCCESS;
}
```

Creating New Template

The steps in creating a new template and the steps in creating a new virtual server are exactly the same, with one exception: before registering a template with Virtuozzo, a call to `PrlVmCfg_SetTemplateSign` function must be made passing the `PRL_TRUE` in the `bVmIsTemplate` parameter. This will set a flag in the configuration structure indicating that you want to create a template, *not* a regular virtual server. The rest of the configuration parameters are set exactly as they are set for a regular virtual server.

The following example illustrates how to create a virtual server template. For simplicity reasons, we only set a template name in this example. The rest of the configuration parameters are omitted. As a result, a blank template will be created. It still can be used to create new virtual servers from it but you will not be able to run them until you configure them properly.

```
PRL_RESULT CreateTemplateFromScratch(PRL_HANDLE hServer)
{
    PRL_HANDLE hVm = PRL_INVALID_HANDLE;
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Create a new virtual server handle.
    ret = PrlSrv_CreateVm(hServer, &hVm);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Set the name for the new template.
    ret = PrlVmCfg_SetName(hVm, "A simple template");
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hVm);
        return -1;
    }

    // Set a flag indicating to create a template.
    PRL_BOOL isTemplate = PRL_TRUE;
    ret = PrlVmCfg_SetTemplateSign(hVm, isTemplate);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hVm);
        return -1;
    }

    // Create and register the new template.
    // The empty string in the configuration path
    // indicates to create a template in the default
    // virtual server directory.
    // The bNonInteractiveMode parameter indicates not to
    // use interactive mode.
    PRL_CHAR_PTR sVmConfigPath = "";
    PRL_BOOL bNonInteractiveMode = PRL_TRUE;
```

```
    hJob = PrlVm_Reg(hVm, sVmConfigPath, bNonInteractiveMode);
    // Wait for the job to complete.
    ret = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        PrlHandle_Free(hVm);
        return -1;
    }

    // Analyze the result of PrlVm_Reg.
    ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        PrlHandle_Free(hVm);
        return -1;
    }
    // Check the job return code.
    if (PRL_FAILED(nJobReturnCode))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        PrlHandle_Free(hVm);
        return -1;
    }
    PrlHandle_Free(hJob);
    PrlHandle_Free(hVm);
    return 0;
}
```

Creating Template from Virtual Server

Any registered virtual server can be converted to a template. This task is accomplished by modifying the virtual server configuration. Only a single parameter must be modified: a flag indicating whether the server is a regular virtual server or a template, the rest will be handled automatically and transparently to you on the server side. The name of the function that allows to modify this parameter is `PrlVm_SetTemplateSign`.

The following code example illustrates how to convert a regular virtual server to a template. Note that any of the virtual server (or a template) configuration changes must begin with the `PrlVm_BeginEdit` and end with the `PrlVm_BeginCommit` function call. You should already know that these two functions are used to prevent collisions with other clients trying to modify the configuration of the same virtual server or template at the same time.

```
PRL_RESULT ConvertVMtoTemplate(PRL_HANDLE hVm)
{
    PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Begin of the VM configuration changes operation.
    hJobBeginEdit = PrlVm_BeginEdit(hVm);
    ret = PrlJob_Wait(hJobBeginEdit, 10000);
```

```
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJobBeginEdit);
        return -1;
    }
    ret = PrlJob_GetRetCode(hJobBeginEdit, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJobBeginEdit);
        return -1;
    }
    // Check the job return code.
    if (PRL_FAILED(nJobReturnCode))
    {
        // Handle the error...
        PrlHandle_Free(hJobBeginEdit);
        return -1;
    }
    PrlHandle_Free(hJobBeginEdit);

    // Set a flag in the virtual server configuration
    // indicating that we want it to become a template.
    PRL_BOOL isTemplate = PRL_TRUE;
    ret = PrlVmCfg_SetTemplateSign(hVm, isTemplate);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Commit the changes.
    hJobCommit = PrlVm_Commit(hVm);
    // Check the results of the commit operation.
    ret = PrlJob_Wait(hJobCommit, 10000);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJobCommit);
        return -1;
    }
    ret = PrlJob_GetRetCode(hJobCommit, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJobCommit);
        return -1;
    }
    // Check the job return code.
    if (PRL_FAILED(nJobReturnCode))
    {
        // Handle the error...
        PrlHandle_Free(hJobCommit);
        return -1;
    }
    PrlHandle_Free(hJobCommit);

    return 0;
}
```

Converting Template to Virtual Server

Converting a template to a regular virtual server is no different than converting a virtual server to a template (see the previous section for the description and an example). Simply set the boolean parameter in the `PrlVmCfg_SetTemplateSign` function to `PRL_FALSE` and leave the rest of the sample code the same.

Creating Virtual Server from Template

The primary purpose of templates is to create new virtual servers from them. New virtual servers are created from templates using the cloning functionality. The `PrlVm_Clone` function that clones a virtual server can also be used to create virtual servers from templates. The function has a boolean parameter that allows to specify whether a virtual server or a template should be created. The following is almost the same example that we used in the **Cloning Virtual Servers** section (p. 50) but this time we are setting the `bCreateTemplate` parameter to `PRL_TRUE`, thus creating a template instead of a regular virtual server.

```
PRL_RESULT CreateVmFromTemplate(PRL_HANDLE hVm)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;
    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;

    // Declare and populate variables that
    // will be used as input parameters
    // in the function that clones a VM.

    // Virtual server name.
    // Get the name of the original VM (template) and use
    // it in the new virtual server name. You can
    // use any name that you like of course.
    char vm_name[1024];
    PRL_UINT32 nBufSize = sizeof(vm_name);
    ret = PrlVmCfg_GetName(hVm, vm_name, &nBufSize);
    char new_vm_name[1024] = "Created from template ";
    strcat(new_vm_name, vm_name);

    // Name of the target directory on the
    // host.
    // Empty string indicates that the default
    // directory should be used.
    PRL_CHAR_PTR new_vm_root_path = "";

    // Virtual server or template?
    // The cloning functionality allows to create
    // a new virtual server or a new template.
    // True specifies to create a template.
    // False indicates to create a virtual server.
    // We want to create a virtual server here, so we
    // set it to PRL_FALSE.
    PRL_BOOL bCreateTemplate = PRL_FALSE;

    // Begin the cloning operation.
    hJob = PrlVm_Clone(hVm, new_vm_name, new_vm_root_path, bCreateTemplate);
    // Wait for the job to complete.
    ret = PrlJob_Wait(hJob, 1000);
```

```
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        printf("Error: (%s)\n",
                prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        return -1;
    }

    // Analyze the result of PrlVm_Clone.
    ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        return -1;
    }
    // Check the job return code.
    if (PRL_FAILED(nJobReturnCode))
    {
        // Handle the error...
        printf("Error: (%s)\n",
                prl_result_to_string(nJobReturnCode));
        PrlHandle_Free(hJob);
        return -1;
    }
    PrlHandle_Free(hJob);
    return 0;
}
```

Events

Receiving and Handling Events

A Virtuozzo program can receive the data describing the event and take appropriate action if needed. Events are received asynchronously (it is not possible to receive event-related data on-demand). All possible event types are defined in the `PRL_EVENT_TYPE` enumeration. Most of them are triggered automatically when the corresponding action takes place. Some event types are generated in response to client requests and are used to pass the data to the client. For example, the `PET_DSP_EVT_HW_CONFIG_CHANGED` event triggers when the host configuration changes, the `PET_DSP_EVT_VM_STOPPED` event triggers when one of the virtual servers is stopped, etc. On the other hand, an event of type `PET_DSP_EVT_FOUND_LOST_VM_CONFIG` is generated in response to the `PrlSrv_StartSearchVms` function call and is used to pass the information about unregistered virtual servers to the client.

In order to receive an event notification, a client program needs an event handler. An event handler (also called *callback*) is a function that you have to implement yourself. We've already discussed event handlers and provided code samples in the **Asynchronous Functions** section (p. 11). If you haven't read it yet, please do so now. To subscribe to event notifications, you must register your event handler with Virtuozzo. This is accomplished using the `PrlSrv_RegEventHandler` function. Once this is done, the event handler (callback) function will be called automatically by the

background thread every time it receives an event notification. The code inside the event handler can then handle the event according to the application logic.

The following describes the general steps involved in handling an event in a callback function:

1. Determine if the notification received is an event (not a *job*, because event handlers are also called when an asynchronous job begins). This can be accomplished using the `PrlHandle_GetType` function (determines the type of the handle received) and then checking if the handle is of type `PHT_EVENT` (not `PHT_JOB`).
2. Determine the type of the event using the `PrlEvent_GetType` function. Check the event type against the `PRL_EVENT_TYPE` enumeration. If it is relevant, continue to the next step.
3. If needed, you can use the `PrlEvent_GetIssuerType` or `PrlEvent_GetIssuerId` function to find out what part of the system triggered the event. This could be a host, a virtual server, an I/O service, or a Web service. These are defined in the `PRL_EVENT_ISSUER_TYPE` enumeration.
4. If, in order to precess the event, you need a server handle, you can obtain it by using the `PrlEvent_GetServer` function.
5. A handle of type `PHT_EVENT` received by the callback function may include event related data. The data is included in the event object as a list of handles of type `PHT_EVENT_PARAMETER`. You can use the `PrlEvent_GetParamsCount` function to determine the number of parameters the event object contains. Some of the events simply inform the client of a change and don't include any data. For example, the virtual server state change events (started, stopped, suspended, etc.) indicate that a virtual server has been started, stopped, suspended, and so forth. These events don't produce any data, so no event parameters are included in the event object. The type of the data and the number of parameters depends on the type of the event received. If you know that an event contains data by definition, continue to the next step, if not, skip it.
6. This step applies only to the events that contain data. Iterate through the event parameters calling the `PrlEvent_GetParam` function in each iteration. This function obtains a handle of type `PHT_EVENT_PARAMETER` which contains the parameter data. Use the functions of the `PHT_EVENT_PARAMETER` handle to process the data as needed. In general, an event parameter contains the following:
   - Parameter name. To retrieve the name, use the `PrlEvtPrm_GetName` function. This is an internal name and is, most likely, not of any interest to a client application developer.
   - Parameter data type. Depending on the event type, a parameter can be of any type defined in the `PRL_PARAM_FIELD_DATA_TYPE` enumeration. To retrieve the parameter data type, use the `PrlEvtPrm_GetType` function.
   - Parameter value. Depending on the parameter data type, the value must be retrieved using an appropriate function from the `PHT_EVENT_PARAMETER` handle. For example, a boolean value must be retrieved using the `PrlEvtPrm_ToBoolean` function, the string value must be retrieved using the `PrlEvtPrm_ToString` function, if a parameter contains a handle, it must be obtained using the `PrlEvtPrm_ToHandle`, etc. The meaning of the value is

usually different for different event types. For the complete list of PHT_EVENT_PARAMETER functions, please see the **Virtuozzo C API Reference**.

7 When finished, release the received event handle. This step is necessary regardless of if you actually used the handle or not. Failure to release the handle will result in a memory leak.

The following is a simple event handler function that illustrates the implementation of the steps described above. We are not including an example of how to register an event handler here, please see the **Asynchronous Functions** section (p. 11) for that.

```
static PRL_RESULT simple_event_handler(PRL_HANDLE hEvent, PRL_VOID_PTR pUserData)
{
    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_HANDLE_TYPE nHandleType;

    // Get the type of the handle recevied.
    PrlHandle_GetType(hEvent, &nHandleType);

    // If this is a job, release the handle and exit.
    // It is up to you if you want to handle jobs and events in
    // the same callback function or if you want to do it in
    // separate functions. You can have as many event handlers
    // registered in your client program as needed.
    if (nHandleType == PHT_JOB)
    {
        PrlHandle_Free(hEvent);
        return 0;
    }

    // If it's not a job, then it is an event (PHT_EVENT).
    // Get the type of the event received.
    PRL_EVENT_TYPE eventType;
    ret = PrlEvent_GetType(hEvent, &eventType);

    // Check the type of the event received.
    switch (eventType) {
        case PET_DSP_EVT_VM_STARTED:
            // Handle the event here...
            printf("A virtual server was started. \n");
            break;
        case PET_DSP_EVT_VM_STOPPED:
            // Handle the event here...
            printf("A virtual server was stopped. \n");
            break;
        case PET_DSP_EVT_VM_CREATED:
            // Handle the event here...
            printf("A new virtual server has been created. \n");
            break;
        case PET_DSP_EVT_VM_SUSPENDED:
            // Handle the event here...
            printf("A virtual server has been suspended. \n");
            break;
        case PET_DSP_EVT_HW_CONFIG_CHANGED:
            // Handle the event here...
            printf("Virtuozzo configuration has been modified. \n");
            break;
        default:
            printf("Unhandled event: %d\n", eventType);
    }
}
```

Responding to Virtuozzo Service Questions

One of the event types in the `PRL_EVENT_TYPE` enumeration deserves special attention. This event type is `PET_DSP_EVT_VM_QUESTION`. While processing a task, a Virtuozzo service may come to a situation that requires client input. For example, let's say that a client requested to create a new virtual server but specified the hard drive size larger than the free disk space available on the host. Since virtual hard drives can dynamically allocate disk space, this is not necessarily a reason to abort the operation. In such a case, the Virtuozzo service will pause the operation and will send a question to the client requiring one of the two possible answers: "Yes, create the server anyway" or "Abort". The question is sent to the client as an event of type `PET_DSP_EVT_VM_QUESTION`. This section describes how to properly handle events of this type.

Handling of the event involves the following steps (we skip the general event handling steps described in the previous section):

1. Obtaining a string containing the question. This is accomplished by making the `PrlEvent_GetErrString` function call.
2. Obtaining the list of possible answers. Answers are included as *event parameters*, therefore they are retrieved using `PrlEvent_GetParamsCount` and `PrlEvent_GetParam` functions as described in the previous section.
3. Selecting an answer. Every available answer has its own unique code which is included in the corresponding event parameter.
4. Sending a response containing the answer back to the Virtuozzo service. This is performed in two steps: first, the `PrlEvent_CreateAnswerEvent` function is used to properly format the answer; second, the answer is sent to the Virtuozzo service using the `PrlSrv_SendAnswer` function.

The following is a complete example that demonstrates how to handle events of type `PET_DSP_EVT_VM_QUESTION` and how to answer Virtuozzo service questions. In the example, we create a blank virtual server and try to add a virtual hard drive to it with the size larger than the free disk space available on the physical drive. This will trigger an event on the server side and a question will be sent to the client asking if we really want to create a drive like that. The virtual server creation operation will not continue unless we send an answer to the service. We then send an answer and the operation continues normally.

```
#include "Parallels.h"
#include "Wrappers/SdkWrap/SdkWrap.h"
#include <stdio.h>
#include <stdlib.h>

#ifdef _WIN_
#include <windows.h>
#else
#include <unistd.h>
#endif

#define MY_JOB_TIMEOUT 10000 // Default timeout.
#define MY_HDD_SIZE 70*1024 // The size of the new hard drive.
#define MY_STR_BUF_SIZE 1024 // The default string buffer size.
```

```
////////////////////////////////////////////////////////////////////////////////

// A helper function that will attempt to create a hard drive larger
// than the free space available, thus triggering an event on the
// server.
static PRL_RESULT create_big_hdd(PRL_HANDLE hVm);

// The callback function (event handler).
static PRL_RESULT callback(PRL_HANDLE, PRL_VOID_PTR);

////////////////////////////////////////////////////////////////////////////////

int main(int argc, char* argv[])
{
    // Pick the correct dynamic library file depending on the platform.
    #ifdef _WIN_
        #define SDK_LIB_NAME "prl_sdk.dll"
    #elif defined(_LIN_)
        #define SDK_LIB_NAME "libprl_sdk.so"
    #elif defined(_MAC_)
        #define SDK_LIB_NAME "libprl_sdk.dylib"
    #endif

    // Load the dynamic library.
    if (PRL_FAILED(SdkWrap_Load(SDK_LIB_NAME)) &&
        PRL_FAILED(SdkWrap_Load("./" SDK_LIB_NAME)))
    {
        // Error handling goes here...
        return -1;
    }

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT err = PRL_ERR_UNINITIALIZED;
    PRL_RESULT rc = PRL_ERR_UNINITIALIZED;
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;
    PRL_HANDLE hServer = PRL_INVALID_HANDLE;

    // Initialize API library.
    err = PrlApi_InitEx(PARALLELS_API_VER, PAM_SERVER, 0, 0);
    if (PRL_FAILED(err))
    {
        // Error handling goes here...
        return -1;
    }

    // Create server object.
    PrlSrv_Create(&hServer);

    // Log in.
    hJob = PrlSrv_Login(
        hServer, // Server handle
        "10.30.22.82", // Server IP address
        "jdoe", // User
        "secret", // Password
        0, // Previous session ID
        0, // Port number
        0, // Timeout
        PSL_NORMAL_SECURITY); // Security
```

```
    ret = PrlJob_Wait(hJob, MY_JOB_TIMEOUT);
    PrlHandle_Free(hJob);

    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlJob_Wait for PrlSrv_Login returned with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Analyze the result of PrlSrv_Login.
    PRL_RESULT nJobResult;
    ret = PrlJob_GetRetCode( hJob, &nJobResult );
    if (PRL_FAILED( nJobResult))
    {
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        printf( "Login job returned with error: %s\n",
            prl_result_to_string(nJobResult));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Create a new virtual server.
    PRL_HANDLE hVm = PRL_INVALID_HANDLE;
    PrlSrv_CreateVm(hServer, &hVm);
    PrlVmCfg_SetName(hVm, "My simple VM");

    // Register the virtual server with Virtuozzo service
    hJob = PrlVm_Reg(hVm, "", PRL_FALSE);
    PrlJob_Wait(hJob, MY_JOB_TIMEOUT);
    PrlHandle_Free(hJob);

    // Register the event handler with the service.
    // The second parameter is a pointer to our callback function.
    PrlSrv_RegEventHandler(hServer, &callback, NULL);

    // Try creating a virtual hard drive larger than the
    // free space available (increase MY_HDD_SIZE value if needed).
    // This should produce an event that will
    // contain a question from the service.
    // We create the drive using a simple helper function.
    // The function is listed at the end of the example.
    create_big_hdd(hVm);

    //
    // At this point, the background thread should call the
    // callback function.
    //

    // We can now clean up and exit the program.
    // Unregister the event handler and log off.
    PrlSrv_UnregEventHandler(hServer, &callback, NULL);
    hJob = PrlSrv_Logoff(hServer);
```

```
    PrlJob_Wait(hJob, MY_JOB_TIMEOUT);
    PrlHandle_Free( hJob );
    PrlHandle_Free( hServer );
    PrlApi_Deinit();
    SdkWrap_Unload();
    return 0;
}

////////////////////////////////////////////////////////////////////////////////
// The callback function implementation.
// The event handling is demonstrated here.
//
static PRL_RESULT callback(PRL_HANDLE hEvent, PRL_VOID_PTR pUserData)
{
    PRL_HANDLE_TYPE nHandleType;
    PrlHandle_GetType(hEvent, &nHandleType);

    // A callback function will be called more than once.
    // It will be called for every job that we initiate and it
    // will be called for the event that we intentionally trigger.
    // In this example, we are interested in events only.
    if (nHandleType != PHT_EVENT)
    {
        return PrlHandle_Free(hEvent);
    }

    // Get the type of the event received.
    PRL_EVENT_TYPE type;
    PrlEvent_GetType(hEvent, &type);

    // See if the received event is a "question".
    if (type == PET_DSP_EVT_VM_QUESTION)
    {
        PRL_UINT32 nParamsCount = 0;
        PRL_RESULT err = PRL_ERR_UNINITIALIZED;

        // Extract the text of the question and display it on the screen.
        PRL_BOOL bIsBriefMessage = true;
        char errMsg [MY_STR_BUF_SIZE];
        PRL_UINT32 nBufSize = MY_STR_BUF_SIZE;
        PrlEvent_GetErrString(hEvent, bIsBriefMessage, errMsg, &nBufSize);
        printf("Question: %s\n\n", errMsg);

        // Extract possible answers. They are stored in the
        // hEvent object as event parameters.
        // First, determine the number of parameters.
        err = PrlEvent_GetParamsCount(hEvent, &nParamsCount);
        if (PRL_FAILED(err))
        {
            fprintf(stderr, "[3]%.8X: %s\n", err,
                prl_result_to_string(err));
            PrlHandle_Free(hEvent);
            return err;
        }

        // Declare an array to hold the answer choices.
        PRL_UINT32_PTR choices =(PRL_UINT32_PTR)
            malloc(nParamsCount * sizeof(PRL_UINT32));

        // Now, itereate through the parameter list obtaining a
        // handle of type PHT_EVENT_PARAMETER containing an individual
```

```
        // parameter data.
        for(PRL_UINT32 nParamIndex = 0; nParamIndex < nParamsCount; ++nParamIndex)
        {
            PRL_HANDLE hParam = PRL_INVALID_HANDLE;
            PRL_RESULT err = PRL_ERR_UNINITIALIZED;

            // The PrlEvent_GetParam function obtains a handle of type
            // PHT_EVENT_PARAMETER containing an answer choice.
            err = PrlEvent_GetParam(hEvent, nParamIndex, &hParam);
            if (PRL_FAILED(err))
            {
                fprintf(stderr, "[4]%.8X: %s\n", err,
                    prl_result_to_string(err));
                PrlHandle_Free(hParam);
                PrlHandle_Free(hEvent);
                return err;
            }

            // Get the answer description that can be shown to the user.
            // First, obtain the event parameter value.
            err = PrlEvtPrm_ToUint32(hParam, &choices[nParamIndex]);
            if (PRL_FAILED(err))
            {
                fprintf(stderr, "[9]%.8X: %s\n", err,
                    prl_result_to_string(err));
                PrlHandle_Free(hParam);
                PrlHandle_Free(hEvent);
                return err;
            }
            // Now, get the answer description using the
            // event parameter value as input in the following call.
            char sDesc [MY_STR_BUF_SIZE];
            err = PrlApi_GetResultDescription(choices[nParamIndex], true,
                                              sDesc, &nBufSize);
            if (PRL_FAILED(err))
            {
                fprintf(stderr, "[8]%.8X: %s\n", err,
                    prl_result_to_string(err));
                PrlHandle_Free(hParam);
                PrlHandle_Free(hEvent);
                return err;
            }

            // Display the answer choice on the screen.
            printf("Answer choice: %s\n", sDesc);
            PrlHandle_Free(hParam);
        }

        // Select an answer choice (we are simply using the "No"
        // answer here) and create a valid answer object (hAnswer).
        PRL_HANDLE hAnswer = PRL_INVALID_HANDLE;
        err = PrlEvent_CreateAnswerEvent(hEvent, &hAnswer, choices[1]);
        if (PRL_FAILED(err))
        {
            fprintf(stderr, "[A]%.8X: %s\n", err, prl_result_to_string(err));
            PrlHandle_Free(hEvent);
            return err;
        }

        // Obtain a server handle. We need it to send an answer.
        PRL_HANDLE hServer = PRL_INVALID_HANDLE;
```

```
        PrlEvent_GetServer(hEvent, &hServer);

        // Send the handle containing the answer data to the Virtuozzo service.
        PrlSrv_SendAnswer(hServer, hAnswer);

        free(choices);
        PrlHandle_Free(hServer);
        PrlHandle_Free(hAnswer);
    }
    else // other event type
    {
        PrlHandle_Free(hEvent);
    }

    return PRL_ERR_SUCCESS;
}

////////////////////////////////////////////////////////////////////////////

// A helper function that will attempt to crate a hard drive larger
// than the free space available, thus triggering an event.
PRL_RESULT create_big_hdd(PRL_HANDLE hVm)
{
    PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_RESULT nJobRetCode = PRL_ERR_UNINITIALIZED;
    PRL_RESULT err = PRL_ERR_UNINITIALIZED;

    // Timestamp the beginning of the configuration changes operation.
    hJobBeginEdit = PrlVm_BeginEdit(hVm);
    err = PrlJob_Wait(hJobBeginEdit, MY_JOB_TIMEOUT);
    PrlJob_GetRetCode(hJobBeginEdit, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        fprintf(stderr, "[B]%.8X: %s\n", nJobRetCode,
            prl_result_to_string(nJobRetCode));
        PrlHandle_Free(hJobBeginEdit);
        return nJobRetCode;
    }

    // Create a new device handle.
    // This will be our new virtual hard disk.
    PRL_HANDLE hHDD = PRL_INVALID_HANDLE;
    err = PrlVmCfg_CreateVmDev(
            hVm, // Target virtual server.
            PDE_HARD_DISK, // Device type.
            &hHDD ); // Device handle.

    // Set disk type to "expanding".
    err = PrlVmDevHd_SetDiskType(hHDD, PHD_EXPANDING_HARD_DISK);

    // Set max disk size, in megabytes.
    err = PrlVmDevHd_SetDiskSize(hHDD, MY_HDD_SIZE);

    // This option determines whether the image file will be split
    // into chunks or created as a single file.
    err = PrlVmDevHd_SetSplitted(hHDD, PRL_FALSE);

    // Choose and set the name for the new image file.
    err = PrlVmDev_SetFriendlyName(hHDD, "harddisk4.hdd");
```

```
    err = PrlVmDev_SetSysName(hHDD, "harddisk4.hdd");

    // Set the emulation type.
    err = PrlVmDev_SetEmulatedType(hHDD, PDT_USE_IMAGE_FILE);

    // Enable the new disk on successful creation.
    err = PrlVmDev_SetEnabled(hHDD, PRL_TRUE);

    // Create the new image file.
    hJob = PrlVmDev_CreateImage(hHDD,
            PRL_TRUE, // Do not overwrite if file exists.
            PRL_FALSE ); // Use non-interactive mode.

    err = PrlJob_Wait(hJob, MY_JOB_TIMEOUT);
    if (PRL_FAILED(err))
    {
        fprintf(stderr, "[C]%.8X: %s\n", err,
            prl_result_to_string(err));
        PrlHandle_Free(hJob);
        return err;
    }

    // Commit the changes.
    hJobCommit = PrlVm_Commit(hVm);
    err = PrlJob_Wait(hJobCommit, MY_JOB_TIMEOUT);
    PrlJob_GetRetCode(hJobCommit, &nJobRetCode);
    if (PRL_FAILED(nJobRetCode))
    {
        fprintf(stderr, "[D]%.8X: %s\n", nJobRetCode,
            prl_result_to_string( nJobRetCode));
        PrlHandle_Free(hJobCommit);
        return nJobRetCode;
    }
    return PRL_ERR_SUCCESS;
}
```

Performance Statistics

Statistics about the CPU(s), memory, disk drives, processes, user session, system uptime, network packets, etc. for a host or a virtual server are available using the Virtuozzo C API. There are two main methods for obtaining statistics:

1. Using `PrlSrv_GetStatistics` (for host statistics) or `PrlVm_GetStatistics` (for virtual server statistics) to obtain a report containing the latest performance data. In addition, the virtual server disk I/O statistics can be obtained using the `PrlVm_GetPerfStats` function.
2. Using `PrlSrv_SubscribeToHostStatistics` (for host statistics) or `PrlVm_SubscribeToGuestStatistics` (for virtual server statistics) to receive statistics on a periodic basis.

The following sections describe each method in detail.

Performance Monitoring

To monitor the host or a virtual server performance on a periodic basis, an event handler (callback function) is required. Within the event handler, first check the type of event. Events of type `PET_DSP_EVT_HOST_STATISTICS_UPDATED` indicate an event containing statistics data. To access the statistics handle (a handle of type `PHT_SYSTEM_STATISTICS`), first extract the event parameter using `PrlEvent_GetParam`, then convert the result (which will be a handle to an object of type `PHT_EVENT_PARAMETER`) to a handle using `PrlEvtPrm_ToHandle`. The functions that operate on `PHT_SYSTEM_STATISTICS` references can then be used to obtain statistics data.

For the event handler to be called, it is necessary to register it with `PrlSrv_RegEventHandler`. Before the event handler will receive statistics events, the application must subscribe to statistics events using `PrlSrv_SubscribeToHostStatistics`. When statistics data is no longer required, unsubscribe from statistics events using `PrlSrv_UnsubscribeFromHostStatistics`. When events are no longer required, unregister the event event handler using `PrlSrv_UnregEventHandler`.

The following is a complete example that demonstrates how to obtain statistics data asynchronously using `PrlSrv_SubscribeToHostStatistics`. Note that the same code could be used to receive statistics data for a virtual server, instead of the host computer, by using `PrlVm_SubscribeToGuestStatistics` instead of `PrlSrv_SubscribeToHostStatistics`, and passing it a handle to a virtual server that is running. This would also require using `PrlVm_UnsubscribeFromGuestStatistics` to stop receiving statistics data for the virtual server.

```
#include "Parallels.h"
#include "Wrappers/SdkWrap/SdkWrap.h"
#include <stdio.h>

#ifdef _WIN_
#include <windows.h>
#else
#include <unistd.h>
#endif

const char *szServer = "123.123.123.123";
const char *szUsername = "Your Username";
const char *szPassword = "Your Password";

// ------------------------------------------------------------------------
// Event handler.
// ------------------------------------------------------------------------
// 1. Check for events of type PET_DSP_EVT_HOST_STATISTICS_UPDATES.
// 2. Display a header if first call to this event handler.
// 3. Get the event param (PHT_EVENT_PARAMETER) from the PHT_EVENT handle.
// 4. Convert event param to a handle (will be type PHT_SYSTEM_STATISTICS).
// 5. Use PHT_SYSTEM_STATISTICS handle to obtain CPU usage, memory usage,
//    and disk usage data.
// ------------------------------------------------------------------------
static PRL_RESULT OurCallback(PRL_HANDLE handle, void *pData)
{
```

```
    PRL_HANDLE_TYPE nHandleType;
    PRL_RESULT ret = PrlHandle_GetType(handle, &nHandleType);
    // Check for PrlHandle_GetType error here.

    if (nHandleType == PHT_EVENT)
    {
        PRL_EVENT_TYPE EventType;
        PrlEvent_GetType(handle, &EventType);

        // Check if the event type is a statistics update.
        if (EventType == PET_DSP_EVT_HOST_STATISTICS_UPDATED)
        {
            // Output a header if first call to this function.
            static PRL_BOOL bHeaderHasBeenPrinted = PRL_FALSE;
            if (!bHeaderHasBeenPrinted)
            {
                bHeaderHasBeenPrinted = PRL_TRUE;
                printf("CPU (%%) Used RAM (MB) Free RAM (MB) Used Disk Space (MB)"
                       " Free Disk Space (MB)\n");
                printf("-------------------------------------------------------------"
                       "--------------------\n");
            }

            PRL_HANDLE hEventParameters = PRL_INVALID_HANDLE;
            PRL_HANDLE hServerStatistics = PRL_INVALID_HANDLE;
            // Get the event parameter (PHT_EVENT_PARAMETER) from the event handle.
            PrlEvent_GetParam(handle, 0, &hEventParameters);
            // Convert the event parameter to a handle (PHT_SYSTEM_STATISTICS).
            PrlEvtPrm_ToHandle(hEventParameters, &hServerStatistics);

            // Get CPU statistics (usage in %).
            PRL_HANDLE hCpuStatistics = PRL_INVALID_HANDLE;
            ret = PrlStat_GetCpuStat(hServerStatistics, 0, &hCpuStatistics);
            PRL_UINT32 nCpuUsage = 0;
            ret = PrlStatCpu_GetCpuUsage(hCpuStatistics, &nCpuUsage);

            // Get RAM statistics.
            PRL_UINT64 nUsedRam, nFreeRam;
            PrlStat_GetFreeRamSize(hServerStatistics, &nFreeRam);
            PrlStat_GetUsageRamSize(hServerStatistics, &nUsedRam);
            nUsedRam /= (1024 * 1024);
            nFreeRam /= (1024 * 1024);

            // Get disk space statistics.
            PRL_UINT64 nFreeDiskSpace, nUsedDiskSpace;
            PRL_HANDLE hDiskStatistics = PRL_INVALID_HANDLE;
            PrlStat_GetDiskStat(hServerStatistics, 0, &hDiskStatistics);
            PrlStatDisk_GetFreeDiskSpace(hDiskStatistics, &nFreeDiskSpace);
            PrlStatDisk_GetUsageDiskSpace(hDiskStatistics, &nUsedDiskSpace);
            nUsedDiskSpace /= (1024 * 1024);
            nFreeDiskSpace /= (1024 * 1024);

            printf("%7d %10lld %13lld %20lld %20lld\n",
                nCpuUsage, nUsedRam, nFreeRam, nUsedDiskSpace, nFreeDiskSpace);

            PrlHandle_Free(hDiskStatistics);
            PrlHandle_Free(hCpuStatistics);
            PrlHandle_Free(hServerStatistics);
            PrlHandle_Free(hEventParameters);
        }
    }
```

```
    PrlHandle_Free(handle);

    return PRL_ERR_SUCCESS;
}


// ------------------------------------------------------------------------
// Program entry point.
// ------------------------------------------------------------------------
// 1. Call SdkWrap_Load(SDK_LIB_NAME).
// 2. Call PrlApi_InitEx().
// 3. Create a PRL_SERVER handle using PrlSrv_Create.
// 4. Log in using PrlSrv_Login.
// 5. Register our event handler (OurCallback function).
// 6. Subscribe to host statistics events.
// 7. Keep receiving events until user presses <enter> key.
// 8. Unsubscribe from host statistics events.
// 9. Un-register our event handler.
// 10. Logoff using PrlSrv_Logoff.
// 11. Call PrlApi_Uninit.
// 12. Call SdkWrap_Unload.
// ------------------------------------------------------------------------
int main(int argc, char* argv[])
{
    PRL_HANDLE hServer = PRL_INVALID_HANDLE;
    PRL_RESULT ret;

    // Use the correct dynamic library depending on the platform.
    #ifdef _WIN_
    #define SDK_LIB_NAME "prl_sdk.dll"
    #elif defined(_LIN_)
    #define SDK_LIB_NAME "libprl_sdk.so"
    #elif defined(_MAC_)
    #define SDK_LIB_NAME "libprl_sdk.dylib"
    #endif

    // Try to load the SDK library, terminate on failure to do so.
    if (PRL_FAILED(SdkWrap_Load(SDK_LIB_NAME)) &&
        PRL_FAILED(SdkWrap_Load("./" SDK_LIB_NAME)))
    {
        fprintf(stderr, "Failed to load " SDK_LIB_NAME "\n");
        return -1;
    }

    // Initialize the Virtuozzo API.
    ret = PrlApi_InitEx(PARALLELS_API_VER, PAM_SERVER, 0, 0);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlApi_InitEx returned with error: %s.\n",
            prl_result_to_string(ret));
        PrlApi_Deinit();
        SdkWrap_Unload();
        return ret;
    }

    // Create a PHP_SERVER handle.
    ret = PrlSrv_Create(&hServer);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlSvr_Create failed, error: %s",
```

```
            prl_result_to_string(ret));
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Log in (PrlSrv is asynchronous).
    PRL_HANDLE hJob = PrlSrv_Login(
            hServer,            // PRL_HANDLE of type PHT_SERVER.
            szServer,           // Host name or IP address.
            szUsername,         // Username.
            szPassword,         // Password.
            0,                  // Deprecated - UUID of previous session.
            0,                  // Optional - port number (0 for default).
            0,                  // Optional - timeout value (0 for default).
            PSL_LOW_SECURITY);  // Security level (can be PSL_LOW_SECURITY,
                                   PSL_NORMAL_SECURITY, or PSL_HIGH_SECURITY).

    // Wait for a maximum of 10 seconds for
    // asynchronous function PrlSrv_Login to complete.
    ret = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlJob_Wait for PrlSrv_Login returned with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Analyze the result of PrlSrv_Login.
    PRL_RESULT nJobResult;
    ret = PrlJob_GetRetCode(hJob, &nJobResult);
    if (PRL_FAILED(nJobResult))
    {
        printf("Login job returned with error: %s\n",
            prl_result_to_string(nJobResult));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }
    else
        printf("Login was successful.\n");

    // ------------------------------------------------------------------------
    // 1. Register our event handler (OurCallback function).
    // 2. Subscribe to host statistics events.
    // 3. Keep receiving events until user presses <enter> key.
    // 4. Unsubscribe from host statistics events.
    // 5. Un-register out event handler.
    // ------------------------------------------------------------------------

    PrlSrv_RegEventHandler(hServer, OurCallback, NULL);
    PrlSrv_SubscribeToHostStatistics(hServer);
    char c;
    scanf(&c, 1);
```

```
    PrlSrv_UnsubscribeFromHostStatistics(hServer);
    PrlSrv_UnregEventHandler(hServer, OurCallback, NULL);

    // ------------------------------------------------------------------------

    // Log off.
    hJob = PrlSrv_Logoff(hServer);
    ret = PrlJob_Wait(hJob, 1000);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlJob_Wait for PrlSrv_Logoff returned error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    ret = PrlJob_GetRetCode(hJob, &nJobResult);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlJob_GetRetCode failed for PrlSrv_Logoff with error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Report success or failure of PrlSrv_Logoff.
    if (PRL_FAILED(nJobResult))
    {
        fprintf(stderr, "PrlSrv_Logoff failed with error: %s\n",
            prl_result_to_string(nJobResult));
        PrlHandle_Free(hJob);
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }
    else
        printf("Logoff was successful.\n");

    // Free handles that are no longer required.
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);

    // De-initialize the Virtuozzo API, and unload the SDK.
    PrlApi_Deinit();
    SdkWrap_Unload();

    return 0;
}
```

Obtaining Performance Statistics

The first step required to obtain performance statistics is to obtain a handle of type `PHT_SYSTEM_STATISTICS`:

1. Call `PrlSrv_GetStatistics` or `PrlVm_GetStatistics`. This will return a job (`PHT_JOB`) reference.
2. Get the job result (a reference to an object of type `PHT_RESULT`) using `PrlJob_GetResult`.
3. Get the handle to the `PHT_SYSTEM_STATISTICS` object using `PrlResult_GetParam` (there will be only one parameter returned).

Functions that can be used to extract statistics data from a `PHT_SYSTEM_STATISTICS` handle can be found in the C API Reference under the following sections:

C API Reference Section	Description
PHT_SYSTEM_STATISTICS	Functions to drill deeper into specific system statistics. As an example, to use functions that return CPU statistics, a handle of type `PHT_SYSTEM_STATISTICS_CPU` will be required. This handle can be obtained using `PrlStat_GetCpuStat`. Functions that return memory statistics are also grouped here.
PHT_SYSTEM_STATISTICS_CPU	Functions that provide CPU statistics data.
PHT_SYSTEM_STATISTICS_DISK	Functions that provide hard disk statistics data.
PHT_SYSTEM_STATISTICS_DISK_PARTITION	Functions that provide statistics data for a disk partition.
PHT_SYSTEM_STATISTICS_IFACE	Functions that provide statistics data for a network interface.
PHT_SYSTEM_STATISTICS_PROCESS	Functions that provide statistics data about processes that are running.
PHT_SYSTEM_STATISTICS_USER_SESSION	Functions that provide statistics data about a user session.

The following code example will display CPU usage, used RAM, free RAM, used disk space, and free disk space using the first method (`PrlSrv_GetStatistics`):

```
// Obtain host statistics (PHT_SYSTEM_STATISTICS handle), and wait for a
// maximum of 10 seconds for the asynchronous call PrlSrv_GetStatistics to complete.
// Note: PrlVm_GetStatistics(hVm) could be used instead of
// PrlSrv_GetStatistics(hServer) if statistics are required for a
// virtual server that is running.
PRL_HANDLE hServerStatisticsJob = PrlSrv_GetStatistics(hServer);
PRL_RESULT nServerStatistics = PrlJob_Wait(hServerStatisticsJob, 10000);
if (PRL_FAILED(nServerStatistics))
{
    printf("PrlSrv_GetStatistics returned error: %s\n",
        prl_result_to_string(nServerStatistics));
    PrlHandle_Free(hServerStatisticsJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
```

```
    return -1;
}

// Check that the job (call to PrlSrv_GetStatistics) was successful.
PrlJob_GetRetCode(hServerStatisticsJob, &nServerStatistics);
if (PRL_FAILED(nServerStatistics))
{
    printf("PrlSrv_GetStatistics returned error: %s\n",
        prl_result_to_string(nServerStatistics));
    PrlHandle_Free(hServerStatisticsJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

// Extract the result (PHT_RESULT handle) for the job.
PRL_HANDLE hResult = PRL_INVALID_HANDLE;
nServerStatistics = PrlJob_GetResult(hServerStatisticsJob, &hResult);
if (PRL_FAILED(nServerStatistics))
{
    printf("PrlJob_GetResult returned error: %s\n",
        prl_result_to_string(nServerStatistics));
    PrlHandle_Free(hServerStatisticsJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

// Get the result (PHT_SYSTEM_STATISTICS handle).
PRL_HANDLE hServerStatistics = PRL_INVALID_HANDLE;
PrlResult_GetParam(hResult, &hServerStatistics);

PRL_HANDLE hCpuStatistics = PRL_INVALID_HANDLE;
ret = PrlStat_GetCpuStat(hServerStatistics, 0, &hCpuStatistics);
if (PRL_FAILED(ret))
{
    printf("PrlStat_GetCpuStat returned error: %s\n",
        prl_result_to_string(ret));
    // Clean up and exit here.
}

// Get CPU usage data (% used).
PRL_UINT32 nCpuUsage = 0;
PrlStatCpu_GetCpuUsage(hCpuStatistics, &nCpuUsage);
printf("CPU usage: %d%%\n", nCpuUsage);

// Get memory statistics.
PRL_UINT64 nUsedRam, nFreeRam;
PrlStat_GetFreeRamSize(hServerStatistics, &nFreeRam);
PrlStat_GetUsageRamSize(hServerStatistics, &nUsedRam);
printf("Used RAM: %I64d MB\nFree RAM: %I64d MB\n",
    nUsedRam/1024/1024, nFreeRam/1024/1024);

// Get disk statistics.
PRL_UINT64 nFreeDiskSpace, nUsedDiskSpace;
PRL_HANDLE hDiskStatistics = PRL_INVALID_HANDLE;
PrlStat_GetDiskStat(hServerStatistics, 0, &hDiskStatistics);
PrlStatDisk_GetFreeDiskSpace(hDiskStatistics, &nFreeDiskSpace);
PrlStatDisk_GetUsageDiskSpace(hDiskStatistics, &nUsedDiskSpace);
```

```
printf("Used Disk Space: %I64d MB\nFree Disk Space: %I64d MB\n",
    nUsedDiskSpace/1024/1024, nFreeDiskSpace/1024/1024);

PrlHandle_Free(hDiskStatistics);
PrlHandle_Free(hCpuStatistics);
PrlHandle_Free(hResult);
PrlHandle_Free(hServerStatistics);
PrlHandle_Free(hServerStatisticsJob);
```

Obtaining Disk I/O Statistics

The virtual server disk I/O statistics are obtained using the `PrlVm_GetPerfStats` function. The function is a part of the `PHT_VIRTUAL_MACHINE` handle.

The `PrlVm_GetPerfStats` function obtains a handle of type `PHT_EVENT`. The objects referenced by `PHT_EVENT` handle can contain one or more `PHT_EVENT_PARAMETER` objects, each of which is used as a container for a particular type of performance statistics. Statistics are identified by a corresponding performance counter which name is also contained in the `PHT_EVENT_PARAMETER` container. The following disk I/O performance counters are available:

Counter Name	Description
`PRL_IDE_READ_REQUESTS_PTRN`	Total count of read requests to IDE controller.
`PRL_IDE_READ_TOTAL_PTRN`	Total count of bytes read through IDE controller.
`PRL_IDE_WRITE_REQUESTS_PTRN`	Total count of write requests to IDE controller.
`PRL_IDE_WRITE_TOTAL_PTRN`	Total count of bytes written through IDE controller.
`PRL_SCSI_READ_REQUESTS_PTRN`	Total count of read requests to SCSI controller.
`PRL_SCSI_READ_TOTAL_PTRN`	Total count of bytes read through SCSI controller.
`PRL_SCSI_WRITE_REQUESTS_PTRN`	Total count of write requests to SCSI controller.
`PRL_SCSI_WRITE_TOTAL_PTRN`	Total count of bytes written through SCSI controller.
`PRL_SATA_READ_REQUESTS_PTRN`	Total count of read requests to SATA controller.
`PRL_SATA_READ_TOTAL_PTRN`	Total count of bytes read through SATA controller.
`PRL_SATA_WRITE_REQUESTS_PTRN`	Total count of write requests to SATA controller.
`PRL_SATA_WRITE_TOTAL_PTRN`	Total count of bytes written through SATA controller.

Example

The following example shows how to obtain the virtual server disk I/O statistics.

```
PRL_HANDLE hVm, hJob, hResult, hEvent, hPerfCounter;
// Obtain performance statistics.
// In this example, we are getting the total count of read
// requests to IDE controller by passing the corresponding
// performance counter name. The performance counter names are
// defined in the PrlPerfCounters.h header file.
hJob = PrlVm_GetPerfStats(hVm, PRL_IDE_READ_REQUESTS_PTRN);
// Wait for the job to complete.
PrlJob_Wait(hJob, 15000);
// Obtain the PHT_RESULT object from the job.
PrlJob_GetResult(hJob, &hResult);
```

```
// Obtain the PHT_EVENT object from the result.
PrlResult_GetParam(hResult, &hEvent);
// Get the PHT_EVENT_PARAMETER object containing
// the actual performance counter value.
PrlEvent_GetParam(hEvent, 0, &hPerfCounter);
// Get the performance counter value.
PRL_UINT32 nValue;
PrlEvtPrm_ToInt32(hPerfCounter, &nValue);
// Process the nValue here...
```

CHAPTER 4

Virtuozzo Python API Concepts

Virtuozzo Python API is a wrapper of the C API described earlier in this guide. While it is based on the same essential principles as the C API, there are some notable differences. They are summarized below.

- Handles are not directly visible in the Python API. Instead, Python classes are used. You don't obtain a handle in Python, you obtain an instance of a class.
- Instead of calling a C function passing a handle to it, you use a Python class and call a method of that class.
- Memory management is automatic. This means that you don't have to free a handle (destroy an object) when it is no longer needed.
- No callbacks! Callback functionality does not exist in the Virtuozzo Python API. This means a few things. First, it is impossible to receive asynchronous method results via callbacks, which essentially means that these methods are not truly asynchronous in the Python API. Second, you cannot receive system event notifications in Python. Finally, you cannot automatically receive periodic performance reports (you can still obtain the reports via synchronous calls).
- Error handling is implemented using exceptions.
- Strings are handled as objects (not as char arrays compared to the C API), which makes it much easier to work with strings as input and output parameters.

In This Chapter

Packages and Modules

The following table lists packages and modules comprising the Virtuozzo Python API.

`prlsdkapi`	This is the main package containing the majority of the classes.
`prlsdkapi.prlsdk`	*This is an internal module. Do not use it in your applications.*

`prlsdkapi.prlsdk.consts`	This module contains constants that are used throughout the API. Most of the constants are combined into groups which are used as pseudo enumerations. Constants that belong to the same group have the same prefix in their names. For example, constants with a `PDE_` prefix identify device types: `PDE_GENERIC_DEVICE`, `PDE_HARD_DISK`, `PDE_GENERIC_NETWORK_ADAPTER`, etc. In this guide, and in the **Virtuozzo Python API Reference** guide, we identify individual groups of constants using these prefixes. For example, we might say, "for the complete list of device types, see the `PDE_xxx` constants".
`prlsdkapi.prlsdk.errors`	This module contains error code constants. There's a very large number of error codes in the API, but the majority of them are used internally. The error code constant are also grouped using prefixes in their names.

The Virtuozzo Python package is installed automatically during the SDK installation and is placed into the default directory for Python site-packages.

Classes

Compared to the Virtuozzo C API, a Python class is an equivalent of a C handle. In most cases, an instance of a class must be obtained using a method of another class. Instances of particular classes are obtained in a certain order. A typical program must first obtain an instance of the `prlsdkapi.Server` class identifying a Virtuozzo host. If the intention is to work with a virtual server, an instance of the `prlsdkapi.Vm` class identifying the virtual server must then be obtained using the corresponding methods of the `prlsdkapi.Server` class.To view or modify the virtual server configuration setting, an instance of the `prlsdkapi.VmConfig` class must be obtained using a method of the `prlsdkapi.Vm` class, and so forth.

The examples in this guide provide information on how to obtain the most important and commonly used objects (server, virtual server, devices, etc.). In general, an instance of a class is obtained using a method of a class to which the first class logically belongs. For example, a virtual server belongs to a server, so the `Server` class must be used to obtain the virtual server object. A virtual device belongs to a virtual server, so the virtual server object must be used to obtain a device object, and so on. In some cases an object must be created manually, but these cases are rare. The most notable one is the `prlsdkapi.Server` class, which is created using the `server = prlsdkapi.Server()` statement in the very beginning of a typical program.

Methods

There are two basic types of method invocations in the Virtuozzo Python API: *synchronous* and *asynchronous*. A synchronous method completes executing before returning to the caller. An asynchronous method starts a job in the background and returns to the caller immediately without waiting for the operation to finish. The following subsections describe both method types in detail.

Synchronous Methods

A typical synchronous method returns the result directly to the caller as soon as it completes executing. In the following example the `vm_config.get_name` method obtains the name of a virtual server and returns it to the caller:

```
vm_name = vm_config.get_name()
```

Synchronous methods in the Virtuozzo Python API are usually used to extract data from local objects that were populated earlier in the program. The data can be extracted as objects or native Python data types. Examples include obtaining virtual server properties, such as name and OS version, virtual hard disk type and size, network interface emulation type or MAC address, etc. In contrast, objects that are populated with data from the Virtuozzo side are obtained using asynchronous methods, which are described in the following section.

Synchronous methods throw the `prlsdkapi.PrlSDKError` exception. For more information on exceptions, see the **Error Handling** section (p. 98).

Asynchronous Methods

All asynchronous methods in the Virtuozzo Python API return an instance of the `prlsdkapi.Job` class. A `Job` object is a reference to the background job that the asynchronous method has started. A job is executed in the background and may take some time to finish. In other languages, asynchronous jobs are usually handled using *callbacks* (event handlers). Unfortunately, callbacks are not available in the Virtuozzo Python API. You have two ways of handling asynchronous jobs in your application. The first one consists of implementing a loop and checking the status of the asynchronous job in every iteration. The second approach involves the main thread suspending itself until the job is finished (essentially emulating a synchronous operation). The following describes each approach in detail.

Checking the job status

The `prlsdkapi.Job` class provides the `get_status` method that allows to determine whether the job is finished or not. The method returns one of the following constants:

`prlsdkapi.prlsdk.consts.PJS_RUNNING` -- indicates that the job is still running.

`prlsdkapi.prlsdk.consts.PJS_FINISHED` -- indicates that the job is finished.

`prlsdkapi.prlsdk.consts.PJS_UNKNOWN` -- the job status cannot be determined for unknown reason.

By evaluating the code returned by the `prlsdkapi.Job.get_status` method, you can determine whether you can process the results of the job or still have to wait for the job to finish. The following code sample illustrates this approach.

```
# Start the virtual server.
```

```
job = vm.start()

# Loop until the job is finished.
while True:
    status = job.get_status()
    if job.get_status() == prlsdkapi.prlsdk.consts.PJS_FINISHED:
        break
```

The scope of the loop in the example above doesn't have to be local of course. You can check the job status in the main program loop (if you have one) or in any other loop, which can be a part of your application design. You can have as many jobs running at the same time as you like and you can check the status of each one of them in the order of your choice.

Suspending the main thread

The `prlsdkapi.Job` class provides the `wait` method that can be used to suspend the execution of the main thread until the job is finished. The method can be invoked as soon as the `Job` object is returned by the original asynchronous method or at any time later. The following code snippet illustrates how it is accomplished.

```
# Start the virtual server. This is an asynchronous call.
job = vm.start()

# Suspend the main thread and wait for the job to complete.
result = job.wait()

# The job is now finished and our program continues...
vm_config = vm.get_config()
print vm_config.get_name() + " was started."
```

You can also execute both the original asynchronous method and the `Job.wait` method on the same line without obtaining a reference to the `Job` object, as shown in the following example. Please note that if you do that, you will not be able to use any of the other methods of the `Job` class that can be useful in certain situations. The reason is, this type of method invocation returns the `prlsdkapi.Result` object containing the results of the operation, not the `Job` object (the `Result` class is described in the **Obtaining the job result** subsection below). It it still a perfectly valid usage that simplifies the program and reduces the number of lines in it.

```
# Start a virtual server, wait for the job to complete.
vm.start().wait()
```

Obtaining the job results

Asynchronous methods that are used to perform actions of some sort (e.g. start or stop a virtual server) don't usually return any data to the caller. Other asynchronous methods are used to obtain data from the Virtuozzo side. The data is usually returned as an object or a list of objects. A good example would be a `prlsdkapi.Vm` object (virtual server), a list of which is returned by the `prlsdkapi.Server.get_vm_list` asynchronous method. The data is not returned to the caller directly. It is contained in the `prlsdkapi.Result` object, which must be obtained from the `Job` object using the `Job.get_result` method. The `prlsdkapi.Result` class is a container that can contain one or more objects or strings depending on the operation that populated it. To determine the number of objects that it contains, the `Result.get_params_count` method must be used. To obtain an individual object, use the `get_param_by_index` method passing an index as a parameter (from 0 to count - 1).When an asynchronous operation returns a single object,

the `get_param` method can be used. Strings are obtained in the similar manner using the corresponding methods of the `Result` class (`get_param_by_index_as_string`, `get_param_as_string`).

The following code snippet shows how to execute an asynchronous operation and then obtain the data from the `Job` object. In this example, we are obtaining the list of virtual servers registered with the Virtuozzo.

```
# Obtain the virtual server list.
# get_vm_list is an asynchronous method that returns
# a prlsdkapi.Result object containing the list of virtual servers.
job = server.get_vm_list()
job.wait()
result = job.get_result()

# Iterate through the Result object parameters.
# Each parameter is an instance of the prlsdkapi.Vm class.
for i in range(result.get_params_count()):
    vm = result.get_param_by_index(i)

    # Obtain the prlsdkapi.VmConfig object containing the virtual server
    # configuration information.
    vm_config = vm.get_config()

    # Get the name of the virtual server.
    vm_name = vm_config.get_name()
```

Other useful Job methods

The Job class provides other useful methods:

`get_ret_code` -- obtains the asynchronous operation return code. On asynchronous operation completion, the job object will contain a return code indicating whether the operation succeeded or failed. Different jobs may return different error codes. The most common codes are `prlsdkapi.prlsdk.consts.PRL_ERR_INVALID_ARG` (invalid input parameters were specified during the asynchronous method invocation) and `prlsdkapi.prlsdk.consts.PRL_ERR_SUCCESS` (method successfully completed).

`cancel` -- attempts to cancel a job that is still in progress. Please note that not all jobs can be canceled.

Asynchronous methods throw the `prlsdkapi.PrlSDKError` exception. For more information on exceptions, see the **Error Handling** section.

Error Handling

Error handling in the Virtuozzo Python API is done through the use of exceptions that are caught in `try` blocks and handled in `except` blocks. All methods in the API throw the `prlsdkapi.PrlSDKError` exception. The `PrlSDKError` object itself contains the error message. To obtain the error code, examine the `prlsdkapi.PrlSDKError.error_code` argument. The error code can be evaluated against standard Virtuozzo API errors, which can be

found in the `prlsdkapi.prlsdk.errors` module. The most common error codes are `PRL_ERR_SUCCESS`, `PRL_ERR_INVALID_ARG`, `PRL_ERR_OUT_OF_MEMORY`. For the complete list of errors, see the `prlsdkapi.prlsdk.errors` module documentation or the **Virtuozzo Python API Reference** guide.

The following code sample illustrates the exception handling:

```
try:
    # The call returns a prlsdkapi.Result object on success.
    result = server.login(host, user, password, '', 0, 0, security_level).wait()
except prlsdkapi.PrlSDKError, e:
    print "Login error: %s" % e
    print "Error code: " + str(e.error_code)
    raise Halt
```

CHAPTER 5

Virtuozzo Python API by Example

This chapter provides code samples and descriptions of how to perform the most common tasks using the Virtuozzo Python API. Each sample is provided as a complete function that can be used in your own program. Each sample function accepts a parameter—usually an instance of the `prlsdkaspi.Server` class identifying a Virtuozzo host or an instance of the `prlsdkapi.Vm` class identifying a virtual server. The **Creating a Basic Application** section (p. 100) shows how to create and initialize the `prlsdkapi.Server` object and contains a skeleton program that can be used to run individual examples provided later in this chapter. To run the examples, simply paste a sample function into the program and then call it from `main()` passing the correct object and/or other required values.

In This Chapter

Creating a Basic Application

The following steps are required in any programs using the Virtuozzo Python API:

1. Import the `prlsdkapi` package. This is the main Virtuozzo Python API package containing the majority of the classes and additional modules.
2. Initialize the API using the `prlsdkapi.init_server_sdk()` function. To verify that the API was initialized successfully, use the `prlsdkapi.is_sdk_initialized()` function.
3. Create an instance of the `prlsdkapi.Server` class. The `Server` class provides methods for logging in and for obtaining other object references (a virtual machine object in particular).
4. Perform the login operation using the `Server.login()` or `Server.login_local()` method. Use the proper method depending on whether this is a local or a remote application.

To exit gracefully, the program should perform the following steps:

1. Log off using the `Server.logoff()` method. The method does not accept any parameters and simply ends the client session.
2. Deinitialize the API using the `prlsdkapi.deinit_sdk()` function.

Example

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# (c) 1999-2016 Parallels IP Holdings GmbH and its affiliates. All rights reserved.
#
#
# Example of prlsdkapi usage.
#

# Import the main Virtuozzo Python API package.
import prlsdkapi

# Import some of the standard Python modules.
# We will not use all of them in this sample, but
# we will use them in other samples later.
import sys, time, getopt, operator, re, random

# Define constants for easy referencing of the Virtuozzo Python API modules.
consts = prlsdkapi.prlsdk.consts

# An exception class to use to terminate the program.
class Halt(Exception):
    pass

"""
Virtuozzo Service login.

    @param server: A new instance of the prlsdkapi.Server class.
    @param host: The host machine IP address. For local login specify "localhost".
    @param user: User name (must be a valid host OS user).
    @param password: User password.
    @param security_level: Connection secuirty level. Must be one of the
                           prlsdk.consts.PSL_xxx constants.
"""
def login_server(server, host, user, password, security_level):

    # Local or remote login?
    if host=="localhost":
        try:
            # The call returns a prlsdkapi.Result object on success.
            result = server.login_local('', 0, security_level).wait()
        except prlsdkapi.PrlSDKError, e:
            print "Login error: %s" % e
            raise Halt
    else:
        try:
            # The call returns a prlsdkapi.Result object on success.
            result = server.login(host, user, password, '', 0, 0,
security_level).wait()
        except prlsdkapi.PrlSDKError, e:
            print "Login error: %s" % e
            print "Error code: " + str(e.error_code)
            raise Halt
    # Obtain a LoginResponse object contained in the Result object.
    # LoginResponse contains the results of the login operation.
    login_response = result.get_param()

    # Get the Virtuozzo version number.
    product_version = login_response.get_product_version()
```

```
    # Get the host operating system version.
    host_os_version = login_response.get_host_os_version()

    # Get the host UUID.
    host_uuid = login_response.get_server_uuid()

    print""
    print "Login successful"
    print""
    print "PCS version: " + product_version
    print "Host OS verions:           " + host_os_version
    print "Host UUID:                 " + host_uuid
    print ""

################################################################################

def main():

    # Initialize the library.
    prlsdkapi.init_server_sdk()

    # Create a Server object and log in to PCS.
    server = prlsdkapi.Server()
    login_server(server, "192.168.10.3", "root", "secret", consts.PSL_NORMAL_SECURITY);

    # Log off and deinitialize the library.
    server.logoff()
    prlsdkapi.deinit_sdk()

if __name__ == "__main__":
    try:
        sys.exit(main())
    except Halt:
        pass
```

Connecting and and Logging In to Virtuozzo Host

The sample program in the previous section provided basic instructions on how to connect and log in to a Virtuozzo host. In this section, we will discuss these operations in greater detail.

Virtuozzo accepts both local and remote connections. If running a program locally, you have an option to login as the current user or as a specific user. If a program is running on a remote server, you always have to specify the user login information.

The `prlsdkapi.Server` class provides two login methods: `login_local` and `login`:

- The `login_local` method is used to establish a local connection as a current user.
- The `login()` method is used to establish a local or a remote connection as a specified user.

The following tables describe the parameters of the two login methods.

prlsdkapi.Server.login_local

Parameter	Type	Description
sPrevSessionUuid	string	[optional] Previous session ID. This parameter can be used in recovering from a lost connection. The ID will be used to restore references to the asynchronous jobs that were started in the previous session and are still running on the server. If you are not restoring a connection, omit this parameter or pass an empty string. The default value is empty string.
port	integer	[optional] Port number at which Virtuozzo is listening for incoming requests. To use the default port number, pass 0. If the default port number was changed by the administrator of your system, specify the correct value. The default value is 0.
security_level	integer	[optional] Communication security level to use for the session. The value must be specified using one of the constants with the PSL_ prefix. The following options are currently available (for possible changes, consult the Virtuozzo Python API Reference guide): PSL_HIGH_SECURITY - using SSL. PSL_LOW_SECURITY - no encryption. PSL_NORMAL_SECURITY - mixed. Virtuozzo configuration have a setting specifying the minimum security level (the setting can be modified). You must specify here an equal or a higher level to establish a connection. To find out the minimum level, use the get_min_security_level method of the prlsdkapi.DispConfig class. The default value is PSL_HIGH_SECURITY.

prlsdkapi.Server.login

Parameter	Type	Description
host	string	The IP address or name of the host.
user	string	User name.
passwd	string	User password.
sPrevSessionUuid	string	[optional] Previous session ID. This parameter can be used in recovering from a lost connection. The ID will be used to restore references to asynchronous jobs that were started in the previous session and are still running on the server. If you are not restoring a connection, omit this parameter or pass an empty string value. The default value is empty string.
port_cmd	integer	[optional] Port number on which Virtuozzo is listening for incoming requests. To use the default port number, pass 0. If the default port number was changed by the administrator, specify the correct value. The default value is 0.

`timeout`	`integer`	[optional] Timeout value in milliseconds. The operation will be automatically interrupted if a connection is not established within this timeframe. Specify 0 (zero) for infinite timeout. The default value is `0`.
`security_level`	`integer`	[optional] Communication security level to use for the session. The value must be specified using one of the constants with the `PSL_` prefix. The following options are currently available (for possible changes, consult the Virtuozzo Python API Reference guide): `PSL_HIGH_SECURITY` - using SSL. `PSL_LOW_SECURITY` - no encryption. `PSL_NORMAL_SECURITY` - mixed. Virtuozzo configuration has a setting specifying the minimum security level (the setting can be modified). You must specify here an equal or a higher level to establish a connection. To find out the minimum level, use the `get_min_security_level` method of the `prlsdkapi.DispConfig` class. The default value is `PSL_HIGH_SECURITY`.

Both methods return an instance of the `prlsdkapi.LoginResponse` class containing some basic information about the host, the new session ID, and the information about the previous session (if applicable).

Obtaining Host Configuration Information

The Virtuozzo Python API provides a set of methods to retrieve detailed information about a host server. This includes:

- CPU(s) -- number of, mode, model, speed.
- Memory (RAM) size.
- Operating system -- type, version, etc.
- Devices -- disk drives, network interfaces, ports, sound.

The information is obtained using the `get_srv_config` method of the `prlsdkapi.Server` class. This is an asynchronous method, so the information is returned via the `Job` and `Result` objects (see the **Asynchronous Methods** section (p. 96) for more information). The name of the class containing the host configuration information is `prlsdkapi.ServerConfig`.

Example

```
"""
    This function demonstrates how to obtain the
    host server configuration information.
    @param server: An instance of prlsdkapi.Server
    identifying the Virtuozzo host.
"""
def get_host_configuration_info(server):
```

```
print ""
print "Host Configuration Information"
print "====================================="

# Obtain an instance of prlsdkapi.ServerConfig containing the
# host configuration information.
try:
    result = server.get_srv_config().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e

srv_config = result.get_param()

# Get CPU count and model.
print "CPU count: " + str(srv_config.get_cpu_count())
print "CPU model: " + srv_config.get_cpu_model()
print "VT-d support: " + str(int(srv_config.is_vtd_supported()))

# Get RAM size.
print "RAM size: " + str(srv_config.get_host_ram_size())

# Get network adapter info.
# The type of the netd object is prlsdkapi.SrvCfgNet.
print ""
print "Network adapters"
print ""
print "No.  Type             Status   System Index"
print "-----------------------------------------"

for i in range(srv_config.get_net_adapters_count()):
    hw_net_adapter = srv_config.get_net_adapter(i)
    adapter_type = hw_net_adapter.get_net_adapter_type()

    if adapter_type == consts.PHI_REAL_NET_ADAPTER:
        adapter_type = "Physical adapter"
    elif adapter_type == consts.PHI_VIRTUAL_NET_ADAPTER:
        adapter_type = "Virtual adapter"
    elif adapter_type == consts.PHY_WIFI_REAL_NET_ADAPTER:
        adapter_type = "Wi-Fi adapter"

    if hw_net_adapter.is_enabled():
        status = "enabled"
    else:
        status = "disabled"

    print " " + str(i+1) + ". " + adapter_type + "   " + \
              status + "   " + str(hw_net_adapter.get_sys_index())
```

Virtual Server Operations

This section and its subsections describe the most common tasks that can be performed on virtual servers using the Virtuozzo Python API.

In order to perform operations on a virtual server, an instance of the `prlsdkapi.Vm` class identifying the virtual server must be obtained. Once you have the instance, you can use its methods to perform some of the virtual server management operations (start, stop, pause, create

snapshot, clone, and many others) and to obtain other objects that allow to perform additional virtual server management functions, such as modifying the virtual server configuration.

Virtuozzo Virtual Machines vs. Virtuozzo Containers

With Virtuozzo, you have a choice of creating and running two types of virtual servers: Virtuozzo Virtual Machines and Virtuozzo Containers. While the two server types use different virtualization technologies (Hypervisor and Operating System Virtualization respectively), the Virtuozzo Python API can be used to manage both server types transparently. This means that the same set of classes and methods (with some exceptions) is used to perform operations on both Virtual Machines and Containers.

The following list provides an overview of the specifics and differences when using the API to manage Virtual Machines or Containers:

- When obtaining the list of the available virtual servers with the `Server.get_vm_list_ex` method, the result set will contain the virtual servers of both types. When iterating through the list and obtaining an individual server configuration data, you can determine the server type using the `VmConfig.get_vm_type` method. The method returns the server type: `consts.PVT_VM` (Virtual Machine) or `consts.PVT_CT` (Container). Once you know the server type, you can perform the desired server management tasks accordingly.
- The majority of classes and methods operate on both virtual server types. A method and its input/output parameters don't change. Some classes and methods are Virtual Machine specific or Container specific. In the Python API Reference documentation such classes and methods have a *Virtuozzo Virtual Machines only* or *Virtuozzo Containers only* note at the beginning of the description. If a class or a method description doesn't specify the server type, it means that it can operate on both Virtual Machines and Containers.
- When working with a particular server type, the input parameters of some methods may have different meaning and may require different values. There are only a few methods like that. The code samples in this documentation provide server type specifics when such specifics exist.

Listing Available Virtual Servers

A list of all available virtual servers is obtained using the `Server.get_vm_list_ex` method. The method obtains a `prlsdkapi.Result` object containing a list of `prlsdkapi.Vm` objects, each of which can be used to obtain a complete information about an individual virtual server. Once we obtain the `Result` object, we will have to extract individual `Vm` objects from it using `Result.get_params_count` and `Result.get_param` methods. The first method returns the `Vm` object count. The second method returns a `Vm` object specified by its index inside the container.

The following example shows how to obtain the virtual server list. The sample functions accepts a `prldsdkapi.Server` object. Before passing it to the function, the object must be properly created, the API library must be initialized, and a connection with a Virtuozzo host must be established. Please see **Creating a Basic Application** (p. 100) for more information and code samples.

Example

```
"""
    Obtain a list of the existing virtual servers and print it
    on the screen.
    @param server: An instance of prlsdkapi.Server
    identifying the Virtuozzo host.
"""
def get_vm_list(server):

    # Obtain the virtual server list.
    # get_vm_list_ex is an asynchronous method that returns
    # a prlsdkapi.Result object containing the list of virtual servers.
    job = server.get_vm_list_ex(consts.PVTF_VM | consts.PVTF_CT)
    result = job.wait()


    # Iterate through the Result object parameters.
    # Each parameter is an instance of the prlsdkapi.Vm class.
    for i in range(result.get_params_count()):
        vm = result.get_param_by_index(i)

        # Obtain the prlsdkapi.VmConfig object containing
        # the virtual server
        # configuration information.
        vm_config = vm.get_config()

        # Get the name of the virtual server.
        vm_name = vm_config.get_name()

        # Get the virtual server type (PVT_VM or PVT_CT)
        vm_type = vm_config.get_vm_type()

        # Translate the obtained server type
        if vm_type == consts.PVT_VM:
            vm_type_desc = "Virtuozzo Virtual Machine"
        elif vm_type == consts.PVT_CT:
            vm_type_desc = "Virtuozzo Container"

        # Obtain the VmInfo object containing the
        # virtual server state info.
        # The object is obtained from the Result object returned by
        # the vm.get_state() method.
        try:
            state_result = vm.get_state().wait()
        except prlsdkapi.PrlSDKError, e:
            print "Error: %s" % e
            return

        # Now obtain the VmInfo object.
        vm_info = state_result.get_param()

        # Get the virtual server state code.
        state_code = vm_info.get_state()
        state_desc = "unknown status"

        # Translate the state code into a readable description.
        # For the complete list of states, see the
        # VMS_xxx constants in the Python API Reference guide.
        if state_code == consts.VMS_RUNNING:
            state_desc = "running"
```

```
        elif state_code == consts.VMS_STOPPED:
            state_desc = "stopped"
        elif state_code == consts.VMS_PAUSED:
            state_desc = "paused"
        elif state_code == consts.VMS_SUSPENDED:
            state_desc = "suspended"

        # Print the virtual server name, type, and status on the screen.
        vm_name = vm_name + "                          "
        print vm_name[:25] + "\t" + vm_type_desc + "\t" + state_desc
```

Searching for Virtual Servers

The example provided in this section does not really show anything new but it can be useful when testing the sample code provided in later sections. The sample function below accepts a virtual server name as a parameter (the name can be partial) and searches for it in the virtual server list retrieved from the host. If it finds it, it returns the `prlsdkapi.Vm` object identifying the virtual server to the caller. The function uses the same approach that was used in **Listing Available Virtual Servers** (p. 30). It obtains the list of virtual server from Virtuozzo, then iterates through it comparing a virtual server name to the specified name.

Example

```
# Obtain a Vm object for the virtual server specified by its name.
# @param vm_to_find: Name of the virtual server to find.
#                    Can be a partial name (starts with the specified string).
def search_vm(server, vm_to_find):

    try:
        result = server.get_vm_list().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    for i in range(result.get_params_count()):
        vm = result.get_param_by_index(i)
        vm_name = vm.get_name()
        if vm_name.startswith(vm_to_find):
          return vm

    print 'Virtual server "' + vm_to_find + '" not found.'
```

The following code demonstrates how the function can be called from the `main()` function.

```
# Search for a virtual server specified by name.
search_name = "Windows"
print ""
print "Searching for '" + search_name + "%'"

vm = search_vm(server, search_name)

if isinstance(vm, prlsdkapi.Vm):
    print "Found virtual server " + vm.get_name()
```

Starting, Stopping, Pausing, Suspending, Resuming

To start, stop, pause, reset, suspend, or resume a virtual server, a `prlsdkaspi.Vm` object must first be obtained. The `prlsdkapi.Vm` class provides individual methods for each of the power operations.

Please note that powering off a virtual server is not the same as performing an operating system shutdown. When a virtual server is stopped, it is a cold stop (i.e. it is the same as turning off the power to a computer). Any unsaved data will be lost. However, if the OS in the virtual server supports ACPI (Advanced Configuration and Power Interface) then it can be used to shut down the virtual server properly. ACPI is currently supported only with "stop" and "pause" operations. Corresponding methods have an additional parameter that can be used to instruct them to use ACPI.

The following code snippets demonstrate how to perform each of the power operations.

Examples

```
# Stop the virtual server.
# The boolean parameter (True) specifies to use ACPI.
try:
    vm.stop(True).wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e

# Start the virtual server.
try:
    vm.start().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e

# Pause the virtual server.
# The boolean parameter specifies to use ACPI.
try:
    vm.pause(True).wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e

# Resume the virtual server.
try:
    vm.resume().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e

# Restart the virtual server.
try:
    vm.restart().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e

# Reset the virtual server. This operation is an equivalent of
# Stop and Start performed in succession.
# The stop operation will NOT use ACPI, so the entire reset
# operation will resemble the "Reset" button pressed on a physical box.
try:
    vm.reset().wait()
```

```
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e
```

Creating New Virtual Servers

Creating a Virtuozzo Virtual Machine

The first step in creating a new Virtuozzo Virtual Machine is to create a blank virtual server and register it with Virtuozzo. A blank virtual server is an equivalent of a hardware box with no operating system installed on the hard drive. Once a blank virtual server is created and registered, it can be powered on and an operating system can be installed on it.

In this section, we will discuss how to create a typical Virtual Machine for a particular OS type using a sample configuration. By using this approach, you can easily create a virtual server without knowing all of the little details about configuring a virtual server for a particular operating system type.

The steps involved in creating a typical virtual server are:

1. Obtain a `prlsdkapi.SrvConfig` object containing the host server configuration information. This information is needed to configure the new virtual server, so it will run properly on the given host.
2. Obtain a new `prlsdkapi.Vm` object that will identify the new virtual server. This must be performed using the `prlsdkapi.Server.create_vm` method.
3. Set the desired virtual server type to create: Virtual Machine or Container. Use the `prlsdkapi.Vm.set_vm_type` method. In this instance, we will create a Virtual Machine.
4. Set the virtual server configuration parameters. For a Virtuozzo Virtual Machine, set the default configuration based on the version of the OS that you will later install in the server. This step is performed using the `prlsdkapi.Vm.set_default_config` method. The OS version is specified using predefined constants that have the `PVS_GUEST_VER_` prefix in their names.
5. Choose a name for the virtual server and set it using the `VmConfing.set_name` method.
6. Modify the default configuration parameters if needed. For example, you may want to modify the hard disk image type and size, the amount of memory available to the server, and the networking options. When modifying a device, an object identifying it must first be obtained and then its methods and properties can be used to make the modifications. The code sample provided in this section shows how to modify some of the default configuration values.
7. Create and register the new server using the `Vm.reg()` method. This step will create the necessary virtual server files on the host and register the server with Virtuozzo. The virtual server directory will have the same name as the name you've chosen for your virtual server and will be created in the default location for this Virtuozzo host. You may specify a different virtual server directory name and path if you wish.

Sample

```
"""
    Create a new virtual server.
"""
```

```
def create_vm(server):

    # Obtain the prlsdkapi.ServerConfig object.
    # The object contains the host server configuration
    # information.
    try:
        result = server.get_srv_config().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e

    srv_config = result.get_param()

    # Create a new prlsdkapi.Vm object.
    vm = server.create_vm()

    # Set the virtual server type (Virtual Machine)
    vm.set_vm_type(consts.PVT_VM)

    # Use the default configuration for the Virtual Machine.
    # Parameters of the  set_default_config method:
    #   param_1: The host server configuration object.
    #   param_2: Target OS type and version.
    #   param_3: Specifies to create the virtual server devices using
    #            default values (the settings can be modified
                 later if needed).
    vm.set_default_config(srv_config, \
                            consts.PVS_GUEST_VER_WIN_WINDOWS8, True)

    # Set the virtual server name and description.
    vm.set_name("Windows8")
    vm.set_description("Virtuozzo Python API sample")

    # Modify the default RAM size and HDD size.
    # These two steps are optional. If you omit them, the
    # default values will be used.
    vm.set_ram_size(256)

    # Set HDD size to 10 gig.
    # The get_device method obtains a prlsdkapi.VmHardDisk object.
    # The index 0 is used because the default configuration has a
    # single hard disk.
    dev_hdd = vm.get_hard_disk(0)
    dev_hdd.set_disk_size(10000)

    # Register the virtual server with the Virtuozzo host.
    # The first parameter specifies to create the server in the
    # default directory on the host computer.
    # The second parameter specifies that non-interactive mode
    # should be used.
    print "Creating a virtual server..."
    try:
        vm.reg("", True).wait()
    except prlsdkapi.PrlSDKError, e:
            print "Error: %s" % e
            return

    print "Virtual server was created successfully."
```

Creating a Virtuozzo Container

The steps involved in creating a typical Virtuozzo Container are:

1. Obtain a new `prlsdkapi.Vm` object that will identify the new virtual server. This must be performed using the `prlsdkapi.Server.create_vm` method.
2. Set the desired virtual server type to create: Virtual Machine or Container. Use the `prlsdkapi.Vm.set_vm_type` method. In this instance, we will create a Container.
3. Choose a name for the virtual server and set it using the `VmConfing.set_name` method.
4. Set the OS template from which the server will be created. Use the `Vm.set_os_template` method. To obtain the list of the available templates, use the `Server.get_ct_template_list` method.
5. Modify the default configuration parameters if needed. For example, you may want to modify the amount of memory available to the server, the networking options, etc. When modifying a device, an object identifying it must first be obtained and then its methods and properties can be used to make the modifications. The code sample provided in this section shows how to modify some of the default configuration values.
6. Create and register the new server using the `Vm.reg()` method. This step will create the necessary virtual server files on the host and register the server with Virtuozzo.

Sample

```
"""
    Create a new Virtuozzo Container.
"""
def create_vm(server):

    # Create a new prlsdkapi.Vm object.
    ct = server.create_vm()

    # Set the virtual server type (Container)
    ct.set_vm_type(consts.PVT_CT)

    # Set the Container name and description.
    ct.set_name("CentOS")
    ct.set_description("Virtuozzo Python API sample")

    # Set the OS template
    ct.set_os_template('centos-6-x86_64')

    # Modify the default RAM size.
    ct.set_ram_size(2048)

    # Register the virtual server with the Virtuozzo host.
    # The first parameter specifies to create the server in the
    # default directory on the host server.
    # The second parameter specifies that non-interactive mode
    # should be used.
    print "Creating a virtual server..."
    try:
        vm.reg("", True).wait()
    except prlsdkapi.PrlSDKError, e:
            print "Error: %s" % e
            return

    print "Virtuozzo Container was created successfully."
```

Obtaining Virtual Server Configuration Information

The virtual server configuration information includes the server name, guest operating system type and version, RAM size, disk drive and network adapter information, and other settings. To obtain this information, a `prlsdkapi.VmConfig` object must be obtained from the `prlsdkapi.Vm` object (the object that identifies the virtual server). The object methods can then be used to extract the data. The examples in this section show how to obtain the most commonly used configuration data.

All sample functions below accept a single parameter, an instance of `prlsdkapi.Vm` class. To obtain the object, you can use the helper function `search_vm()` that we've created in the **Searching for Virtual Servers** section (p. 33).

Obtaining the RAM size

```
def get_vm_ram_size(vm):

    # Obtain the VmConfig object containing the virtual server
    # configuration information.
    vm_config = vm.get_config()

    # Get the virtual server RAM size.
    ram_size = vm_config.get_ram_size()
    print "RAM size: " + str(ram_size)
```

Obtaining the OS type and version

```
def get_vm_os_info(vm):

    print ""

    # Virtual server name.
    print "Virtual server name: " + vm.get_name()

    # Obtain the VmConfig object containing the virtual server
    # configuration information.
    vm_config = vm.get_config()

    # Obtain the guest OS type and version.
    # OS types are defined as PVS_GUEST_TYPE_xxx constants.
    # For the complete list, see the documentation for
    # the prlsdkapi.prlsdk.consts module or
    # the Virtuozzo Python API Reference guide.
    os_type = vm_config.get_os_type()
    if os_type == consts.PVS_GUEST_TYPE_WINDOWS:
        osType = "Windows"
    elif os_type == consts.PVS_GUEST_TYPE_LINUX:
        osType = "Linux"
    else:
        osType = "Other type (" + str(os_type) + ")"

    # OS versions are defined as PVS_GUEST_VER_xxx constants.
    os_version = vm_config.get_os_version()
    if os_version == consts.PVS_GUEST_VER_WIN_WINDOWS8:
        osVersion = "Windows 8"
    elif os_version == consts.PVS_GUEST_VER_LIN_CENTOS:
        osVersion = "CentOS"
```

```
    else:
        osVersion = "Other version (" + str(os_version) + ")"

    print "Guest OS: " + osType + " " + osVersionRAM size
```

Obtaining optical disk drive information

This functionality applies to Virtuozzo Virtual Machines only.

```
def get_optical_drive_info(vm):

    # Obtain the VmConfig object containing the virtual server
    # configuration information.
    vm_config = vm.get_config()

    print ""
    print "Optical Drives:"
    print "-------------------"

    # Iterate through the existing optical drive devices.
    count = vm_config.get_optical_disks_count()
    for i in range(count):
        print ""
        print "Drive " + str(i)

        # Obtain an instance of VmDevice containing the optical drive info.
        device = vm_config.get_optical_disk(i)

        # Get the device emulation type.
        # In case of optical disks, this value specifies whether the virtual device
        # is using a real disk drive or an image file.
        emulated_type = device.get_emulated_type()

        if emulated_type == consts.PDT_USE_REAL_DEVICE:
            print "Uses physical device"
        elif emulated_type == consts.PDT_USE_IMAGE_FILE:
            print "Uses image file " + '"' + device.get_image_path() + '"'
        else:
            print "Unknown emulation type"

        if device.is_enabled():
            print "Enabled"
        else:
            print "Disabled"

        if device.is_connected():
            print "Connected"
        else:
            print "Disconnected"
```

Obtaining hard disk information

```
def get_hdd_info(vm):

    # Obtain the VmConfig object containing the virtual server
    # configuration information.
    vm_config = vm.get_config()

    print ""
    print "Virtual Hard Disks:"
    print "-------------------"
```

```
    count = vm_config.get_hard_disks_count()
    for i in range(count):
        print ""
        print "Disk " + str(i)

        hdd = vm_config.get_hard_disk(i)
        emulated_type = hdd.get_emulated_type()

        if emulated_type == consts.PDT_USE_REAL_DEVICE:
            print "Uses Boot Camp: Disk " + hdd.get_friendly_name()
        elif emulated_type == consts.PDT_USE_IMAGE_FILE:
            print "Uses image file " + '"' + hdd.get_image_path() + '"'

        if hdd.get_disk_type() == consts.PHD_EXPANDING_HARD_DISK:
            print "Expanding disk"
        elif hdd.get_disk_type() == consts.PHD_PLAIN_HARD_DISK:
            print "Plain disk"

        print "Disk size:" + str(hdd.get_disk_size()) + " Mbyte"
        print "Size on physical disk: " + str(hdd.get_size_on_disk()) + " Mbyte"
```

Obtaining network adapter information

```
def get_net_adapter_info(vm):

    # Obtain the VmConfig object containing the virtual server
    # configuration information.
    vm_config = vm.get_config()

    # Obtain the network interface info.
    # The vm.net_adapters sequence contains objects of type VmNetDev.
    print ""
    print "Network Adapters:"

    count = vm_config.get_net_adapters_count()
    for i in range(count):
        print ""
        print "Adapter " + str(i)

        net_adapter = vm_config.get_net_adapter(i)

        emulated_type = net_adapter.get_emulated_type()

        if emulated_type == consts.PNA_HOST_ONLY:
            print "Uses host-only networking"
        elif emulated_type == consts.PNA_SHARED:
            print "Uses shared networking"
        elif emulated_type == consts.PNA_BRIDGED_ETHERNET:
            print "Uses bridged ethernet (bound to " +
net_adapter.get_bound_adapter_name() + ")"

    print "MAC address " + str(net_adapter.get_mac_address())
```

Modifying Virtual Server Configuration

Vm.begin_edit and Vm.commit Methods

All virtual server configuration changes must begin with the `prlsdkapi.Vm.begin_edit` and end with the `prlsdkapi.Vm.commit` call. These two methods are used to detect collisions with

other programs trying to modify the configuration settings of the same virtual server at the same time.

The `Vm.begin_edit` method timestamps the beginning of the editing operation. It does not lock the server, so other programs can still make changes to the same virtual server. The method will also automatically update your local virtual server object with the current virtual server configuration information. This is done in order to ensure that the local object contains the changes that might have happened since you obtained the virtual server object. When you are done making the changes, you must invoke the `Vm.commit` method. The first thing that the method will do is verify that the virtual server configuration has not been modified by other programs since you began making your changes. If a collision is detected, your changes will be rejected and `Vm.commit` will throw an exception. In such a case, you will have to reapply the changes. In order to do that, you will have to get the latest configuration using the `Vm.refresh_config` method and re-evaluate it. Please note that `Vm.refresh_config` method will update the configuration data in your local virtual server object and will overwrite all existing data, including the changes that you've made so far. Furthermore, the `Vm.begin_edit` method will also overwrite all existing data (see above). If you don't want to loose your data, save it locally before invoking any of the two methods.

Modifying Name, Description, RAM Size

The virtual server name, description, and RAM size modifications are straightforward — they are performed by using a corresponding method of a `Vm` object. Note that most of the methods used in this example belong to the `VmConfig` class, but since the `Vm` class is inherited from `VmConfig`, the methods are called directly from it.

Example

```
"""
    Modify the vrtual server name, description, and RAM size.
"""
def vm_edit(vm):

    # Begin the virtual server editing operation.
    try:
        vm.begin_edit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    vm.set_name(vm.get_name() + "_modified")
    vm.set_ram_size(2048)
    vm.set_description("SDK Test Server")

    # Commit the changes.
    try:
       vm.commit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return
```

Setting Boot Device Priority in Virtual Machines

This functionality applies to Virtuozzo Virtual Machines only

To modify the boot boot device priority, obtain an instance of the `prlsdkapi.BootDevice` class representing each device. This step is performed using the `VmConfig.get_boot_dev_count` method to determine the total number of boot devices, then iterating through the list and obtaining a `BootDevice` object using the `VmConfig.get_boot_dev` method. To place a device at the specified position in the boot device priority list, use the `BootDevice.set_sequence_index` method passing a value of 0 to the first device, 1 to the second device, and so forth. If you have more than one instance of a particular device type in the boot list (i.e. more than one CD/DVD drive), you will have to set a sequence index for each instance individually. An instance is identified by an index that can be obtained using the `VmBootDev.get_index` method. A device in the boot priority list can be enabled or disabled using the `BootDevice.set_in_use` method. Disabling the device does not remove it from the boot device list. To remove a device from the list, use the `BootDevice.remove` method.

Example

```
"""
    Modify the virtual server boot device priority.
"""
def vm_edit(vm):

    # Begin the virtual server editing operation.
    try:
        vm.begin_edit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Modify boot device priority using the following order:
    # CD > HDD > Network > FDD.
    # Remove all other devices from the boot priority list (if any).
    count = vm.get_boot_dev_count()
    for i in range(count):

        # Obtain an instance of the prlsdkapi.BootDevice class
        # containing the boot device information.
        boot_dev = vm.get_boot_dev(i)

        # Enable the device.
        boot_dev.set_in_use(True)

        # Set the device sequence index.
        dev_type = boot_dev.get_type()

        if dev_type == consts.PDE_OPTICAL_DISK:
            boot_dev.set_sequence_index(0)
        elif dev_type == consts.PDE_HARD_DISK:
            boot_dev.set_sequence_index(1)
        elif dev_type == consts.PDE_GENERIC_NETWORK_ADAPTER:
           boot_dev.set_sequence_index(2)
        elif dev_type == consts.PDE_FLOPPY_DISK:
            boot_dev.set_sequence_index(3)
        else:
            boot_dev.remove()

    # Commit the changes.
    try:
       vm.commit().wait()
```

```
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return
```

Adding Hard Disks to Virtual Machines

The following sample function demonstrates how to add a virtual hard disk to a Virtuozzo Virtual Machine.

The steps are:

1. Mark the beginning of an editing operation.
2. Create a device object representing the new disk.
3. Set the device properties, including emulation type (image file or real device), disk type (expanding or fixed), disk size, and disk name.
4. Create an image file.
5. Commit the changes.

Example

```
"""
    Add a new virtual hard disk to a Virtual Machine.
"""
def add_hdd(vm):

    # Begin the virtual server configuration editing.
    try:
        vm.begin_edit().wait
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Create an instance of the prlsdkapi.VmHardDisk class.
    hdd_dev = vm.create_vm_dev(consts.PDE_HARD_DISK)

    # Populate the object.
    # Set emulation type (image file or real device).
    hdd_dev.set_emulated_type(consts.PDT_USE_IMAGE_FILE)

    # Set disk type (expanding or fixed)
    hdd_dev.set_disk_type(consts.PHD_EXPANDING_HARD_DISK)

    # Set disk size.
    hdd_dev.set_disk_size(1024)

    # Set a name for the new image file.
    # Both the friendly_name and the sys_name properties must be
    # populated and must contain the same value.
    # The new image file will be created in
    # the virtual server directory.
    # To create the file in a different directory,
    # the name must contain the full directory path and
    # the hard disk name.
    hdd_name = vm_config.get_name() + "_hdd_sample.hdd"
    hdd_dev.set_friendly_name(hdd_name)
    hdd_dev.set_sys_name(hdd_name)
```

```
    # Enable the disk.
    hdd_dev.set_enabled(True)

    # Create the image file.
    # First parameter - Overwrite the image file if it exists.
    # Second paramerer - Use non-interactive mode.
    try:
        hdd_dev.create_image(True, True).wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Commit the changes.
    try:
       vm.commit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    print("New hard disk was created successfully.")
```

Adding Hard Disks to a Virtuozzo Containers

The following sample function demonstrates how to add a virtual hard disk to a Virtuozzo Container.

The steps are:

1. Mark the beginning of an editing operation.
2. Create a device object representing the new disk.
3. Set the device properties, including emulation type (image file or real device), disk type (expanding or fixed), disk size, and disk name.
4. Create an image file.
5. Commit the changes.

Example

```
"""
    Add a new virtual hard disk to a Virtuozzo Container.
    The function accept a handle of type prlsdkapi.Vm identifying
    a Virtuozzo Container.
"""
def add_hdd(vm):

    # Begin the virtual server configuration editing.
    try:
        vm.begin_edit().wait
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Create an instance of the prlsdkapi.VmHardDisk class.
    hdd_dev = ct.create_vm_dev(consts.PDE_HARD_DISK)

    # Populate the object.
```

```
    # Set emulation type (image file or real device).
    hdd_dev.set_emulated_type(consts.PDT_USE_IMAGE_FILE)

    # Set disk type (expanding or fixed)
    hdd_dev.set_disk_type(consts.PHD_EXPANDING_HARD_DISK)

    # Set disk size.
    hdd_dev.set_disk_size(1024)

    # Enable the disk.
    hdd_dev.set_enabled(True)

    # Create the image file.
    # First parameter - overwrite the image file if it exists.
    # Second paramerer - use non-interactive mode.
    try:
        hdd_dev.create_image(True, True).wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Commit the changes.
    try:
       vm.commit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    print("New hard disk was created successfully.")
```

Adding Network Adapters to Virtual Machines

To add a new virtual network adapter to a Virtual Machine, the following steps must be taken:

1. Mark the beginning of the editing operation.
2. Create a device object representing the new adapter.
3. Set the emulation type (host-only, shared, or bridged).
4. If creating a bridged adapter, select the host adapter to bind the new adapter to.
5. Commit the changes.

Example

```
"""
    Add a network adapter to a Virtual Machine.

    Input parameters:
      vm: An instance of prlsdkapi.Vm class identifying
          a Virtual Machine.
      networking_type: Host-only/shared/bridged. Use one of the
                       consts.PNA_xxx constants.
      bound_default: Used with bridged networking only.
                     Specify True to bound a new adapter to the
                     default physical adapter. If False is passed,
                     the adapter will be bound to a specific physical
                     adapter (in this example, the adapter is
                     chosen randomly).
"""
```

```
def add_net_adapter(server, vm, networking_type, bound_default = True):

    # Begin an editing operation.
    try:
        vm.begin_edit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Create an instance of the prlsdkapi.VmNet class.
    net_adapter = vm.create_vm_dev(consts.PDE_GENERIC_NETWORK_ADAPTER)

    # Set emulation type.
    net_adapter.set_emulated_type(networking_type)

    # For bridged netowkring mode, we'll have to bind the
    # new adapter to a network adapter on the host server.
    if networking_type == consts.PNA_BRIDGED_ETHERNET:

        # To use the default adapter, simply set the
        # adapter index to -1.
        if bound_default == True:
            net_adapter.set_bound_adapter_index(-1)
        else:
            # To use a specific adapter, first obtain the
            # list of the adapters from the host server.

            # Obtain an instance of prlsdkapi.ServerConfig containing
            # the host configuration information.
            try:
                result = server.get_srv_config().wait()
            except prlsdkapi.PrlSDKError, e:
                print "Error: %s" % e

            srv_config = result.get_param()

            # Iterate through the list of the host network adapters.
            # In this example, we are simply selecting the first
            # adapter in the list and binding the virtual adapter to it.
            # The adapter is identified by its name.
            for i in range(srv_config.get_net_adapters_count()):
                hw_net_adapter = srv_config.get_net_adapter(i)
                hw_net_adapter_name = hw_net_adapter.get_name()
                net_adapter.set_bound_adapter_name(hw_net_adapter_name)
                exit

    # Connect and enable the new virtual adapter.
    net_adapter.set_connected(True)
    net_adapter.set_enabled(True)

    # Commit the changes.
    try:
       vm.commit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    print("Virtual network adapter created successfully")
```

Adding Network Adapters to Virtuozzo Containers

To add a new virtual network adapter to a Virtuozzo Container, the following steps must be taken:

1. Obtain a reference to a network adapter from the host server to which the new adapter in a Container will be connected.
2. Create a device object representing the new adapter.
3. Set the virtual network ID.
4. Bind the new adapter to the adapter obtained in step 1 above.
5. Commit the changes.

Example

```
"""
    Add a network adapter to the virtual server.

    Input parameters:
      server: An instance of prlsdkapi.Server class.
      vm: An instance of prlsdkapi.Vm class identifying
          a virtual server.
"""
def add_net_adapter(server, vm):

    # Get a reference to the network adapter on the
    # host server.
    srv_config = srv.get_srv_config().wait().get_param()
    net_adapter = srv_config.get_net_adapter(0)

    # Begin editing operation.
    try:
        vm.begin_edit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Create a new adapter object.
    net = vm.create_vm_dev(consts.PDE_GENERIC_NETWORK_ADAPTER)

    # Set the virtual network ID. You can use any value here.
    net.set_virtual_network_id('Bridged')

    # Enable and bind the new adapter to the
    # adapter on the host server.
    net.set_enabled(True)
    net.set_bound_adapter_index(net_adapter.get_sys_index())
    net.set_bound_adapter_name(net_adapter.get_name())

    # Commit the changes.
    try:
       vm.commit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return
```

Registering Virtual Servers with Virtuozzo

A host server may have virtual servers that are not registered with Virtuozzo. This can happen if a virtual server was previously removed from the Virtuozzo registry or if the virtual server files were manually copied from a different location. This topic describes how to register such servers with Virtuozzo.

Note: When adding an existing virtual server, the MAC addresses of its virtual network adapters are kept unchanged. If the server is a copy of another virtual server, then you should set new MAC addresses for its network adapters after you register it. The example below demonstrates how this can be accomplished.

Example:

The following sample function demonstrates how to register an existing virtual server. The function takes a `Server` object identifying the Virtuozzo and a string specifying the name and path of the virtual server directory (on Mac OS X it is the name of a bundle). It registers the virtual server and then modifies the MAC address of every virtual network adapter installed in it.

```
"""
    Add an existing virtual server.
    @param path: Name and path of the virtual server directory or bundle.
"""
def register_vm(server, path):

    try:
        result = server.register_vm(path, False).wait()
    except prlsdkapi.PrlSDKError, e:
            print "Error: %s" % e
            return

    vm = result.get_param()
    print vm.get_name() + " has been registered."

    # Generate a new MAC addresses for all virtual network adapters.
    # This should be done when a virtual server was copied from another host.
    try:
        vm.begin_edit()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Iterate through the network adapter list and
    # generate a new MAC address.
    # The get_net_adapter(i) method returns an instance of the VmNet class.
    for i in range(vm.get_net_adapters_count()):
        net_adapter = vm.get_net_adapter(i)
        net_adapter.generate_mac_addr()

    # Commit the changes.
    try:
       vm.commit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return
```

Unregistering or Deleting Virtual Servers

If a virtual server is no longer needed, it can be removed. There are two options for removing a virtual server:

- Unregister the virtual server without deleting its files. You can re-register the virtual server later if needed.
- Delete the virtual server from the host completely. The virtual server files will be permanently deleted and cannot be recovered if this option is used.

Example

The following sample function shows how to implement both options described above. The function takes a `Vm` object identifying the virtual server and a boolean value indicating whether the virtual server files should be deleted from the host server.

```
"""
    Remove an existing virtual server.
    @param vm: An instance of prlsdkapi.Vm class identifying
               a virtual server.
    @param delete: A boolean value indicating whether the
                   virtual server files should be permanently deleted
                   from the host.
"""
def remove_vm(vm, delete):

    if delete == False:
        # Unregister the virtual server but don't delete its files.
        try:
            vm.unreg()
        except prlsdkapi.PrlSDKError, e:
            print "Error: %s" % e
            return
    else:
        # Unregister the server and delete its files from the hard drive.
        try:
            vm.delete()
        except prlsdkapi.PrlSDKError, e:
            print "Error: %s" % e
            return
```

Cloning Virtual Servers

A virtual server can be created by cloning an existing virtual server. The server will be created as an exact copy of the source virtual server and will be automatically registered with the Virtuozzo host. The cloning operation is performed using the `prlsdkapi.Vm.clone` method. The following parameters must be specified when cloning a virtual server:

1. A unique name for the new virtual server (the new name is NOT generated automatically).
2. The name of the directory where the new virtual server files should be created or an empty string to create the files in the default directory.

3 A boolean value specifying whether to create a regular virtual server or a virtual server template. Virtual server templates are used to create other virtual servers from them. You cannot run a template.

Sample

```
"""
    Clone a virtual server.
    @param vm: An instance of the prlsdkapi.Vm class identifying
               the source virtual server.
"""
def clone_vm(vm):

    new_name = "Clone of " + vm.get_name()
    print "Clonning is in progress..."

    # Second parameter - create a new server in the
    # default directory.
    # Third parameter - create a virtual server (not a template).
    try:
        vm.clone(new_name, "", False).wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    print "Cloning was successful. New virtual server name: " + new_name
```

Executing Programs in Virtual Servers

Executing a Program in a Virtual Machine

You can execute a program inside a Virtual Machine from your program using the Python API.

To execute a program, you need to perform the following steps:

1 Log in to a Virtual Machine. You have three choices here (the details are described in the code sample below):

- If you want to execute a console program as a specific user, you can do so by supplying the user ID and password.
- If you want to execute a console program as a superuser (root, LocalSystem), you can use a login ID that is predefined in the API for this purpose.
- If you want to run a GUI application, you need to use a predefined login ID that's used to bind to an existing GUI session in a Virtual Machine. For this login type to work, the Virtual Machine must be running and a user must be logged in to it. Please note that after you launch a GUI application, you cannot control it. For instance, if the application requires user interaction, you cannot directly control it from your Python program.

2 Create an object to hold the program input parameters and populate it.

3 Execute the program.

4 Evaluate the results. Please note that you cannot obtain the results of the program execution directly. You can only determine whether the program executed successfully or not.

Sample

The following sample program demonstrates how to execute a batch file (C:\\123.bat) in Windows running in a Virtual Machine. To test the program, create a batch file with a simple instruction (e.g. creating a directory on the C: drive) and see if after executing the program the directory is actually created.

```
"""
    Executes a batch file in a Windows Virtual Machine.
    @param vm: An instance of the prlsdkapi.Vm class identifying
               the Virtual Machine.
"""
def exec_batch(vm):

    # Uncomment the desired "g_login = " and "g_password = " lines below.

    # Bind to an existing session.
    # Use this login to run a console program as a
    # superuser (LocalSystem in Windows, root in Linux).
    # Use this exact login ID and leave the password blank.
    g_login = "531582ac-3dce-446f-8c26-dd7e3384dcf4"
    g_password = ""

    # Log in as a specific user (a new session will be created).
    #g_login = "user_name"
    #g_password = "password"

    # Bind to an existing GUI session.
    # The Virtual Machine user must be logged in.
    # Use this exact login ID and leave the password blank.
    #g_login = "4a5533a7-31c6-4d7a-a400-1f330dc57a9d"
    #g_password = ""

    # Create a StringList object to hold the program input parameters and populate it.
    hArgsList = prlsdkapi.StringList()
    hArgsList.add_item("cmd.exe")
    hArgsList.add_item("/C")
    hArgsList.add_item("C:\\123.bat")

    # Create an empty StringList object.
    # The object is passed to the VmGuest.run_program method and is used to
    # specify the list of environment variables and their values to add to the program
    # execution environment. In this sample, we are not adding any variables.
    # If you wish to add a variable, add it in the var_name=var_value format.
    hEnvsList = prlsdkapi.StringList()
    hEnvsList.add_item("")

    # Establish a user session.
    # The Vm.login_in_guest() method returns a VmGuest object, which is
    # obtained from the Job using the Result.get_param() method.
    vm_guest = vm.login_in_guest(g_login, g_password).wait().get_param()

    # Run the program.
    try:
        vm_guest.run_program("cmd.exe", hArgsList, hEnvsList).wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e

    # Log out.
    vm_guest.logout().wait()
```

Executing a Program in a Virtuozzo Container

You can execute a program inside a Virtuozzo Container from your program using the Python API.

To execute a program, you need to perform the following steps:

1. Connect to a Container.
2. Create an object to hold the program input parameters and populate it.
3. Execute the program.
4. Evaluate the results.

Sample

```
"""
    Executes the '/bin/cat /proc/cpuinfo' command in a Container.

    @param vm: An instance of the prlsdkapi.Vm class identifying
               a Container.
"""
def exec_program(vm):

    # Connect to the Container.
    io = prlsdkapi.VmIO()
    io.connect_to_vm(vm, consts.PDCT_HIGH_QUALITY_WITHOUT_COMPRESSION).wait()

    # Open a guest session.
    result = vm.login_in_guest('root', '').wait()
    guest_session = result.get_param()

    # Format arguments and environment.
    sdkargs = prlsdkapi.StringList()

    cmd = '/bin/cat'
    args = '/proc/cpuinfo'

    for arg in args:
        sdkargs.add_item(arg)
        sdkargs.add_item("")
        sdkenvs = prlsdkapi.StringList()
        sdkenvs.add_item("")

    # Execute the program
    # Open a pipe to read the program execution output.
    ifd, ofd = os.pipe()
    flags = consts.PRPM_RUN_PROGRAM_ENTER|consts.PFD_STDOUT
    job = guest_session.run_program(cmd, sdkargs, sdkenvs, flags, ofd)

    # Read the program output
    output = ''
    while True:
        rfds, _, _ = select.select([ifd], [], [], 5)
        if not rfds:
            break
            buf = os.read(ifd, 256)
            output += buf

    print('output:\n%s--end of output' % output)
```

```
    # Cleanup and log out.
    os.close(ifd)
    job.wait()
    guest_session.logout()

    io.disconnect_from_vm(vm)
```

Index