

Parallels Server

Programmer's Guide

Copyright © 1999-2008 by Parallels Software International, Inc. All rights reserved.

Parallels, Coherence, Parallels Transporter, Parallels Compressor, Parallels Desktop, and Parallels Explorer are registered trademarks of Parallels Software International, Inc. The Parallels logo is a trademark of Parallels Software International, Inc.

This product is based on a technology that is the subject matter of a number of patent pending applications. Distribution of this work or derivative of this work in any form is prohibited unless prior written permission is obtained from the copyright holder.

Microsoft, Windows, Windows Server, Windows NT, Windows Vista, and MS-DOS are registered trademarks of Microsoft Corporation.

Linux is a registered trademark of Linus Torvalds.

Apple, Bonjour, Finder, Mac, Macintosh and Mac OS are trademarks of Apple Inc.

Solaris is a registered trademark of Sun Microsystems, Inc.

eComStation is a trademark of Serenity Systems International.

FreeBSD is a registered trademark of the FreeBSD Foundation.

Intel and Intel Core are trademarks or registered trademarks of Intel Corporation.

OS/2 Warp is a registered trademark of International Business Machines Corporation.

VMware is a registered trademark of VMware, Inc.

All other marks and names mentioned herein may be trademarks of their respective owners.

Contents

Getting Started	6
Overview	6
System Requirements	7
Mac OS X Clients.....	7
Windows Clients	7
Linux Clients	7
Common Network Requirements	8
Installation	8
Installing Parallels Server SDK on Mac OS X	8
Installing Parallels Server SDK on Windows.....	8
Installing Parallels Server SDK on Linux.....	9
Compiling Client Applications	9
Mac OS X.....	9
Windows.....	10
Linux	11

API Basics 12

Handles13
 Working with Results15
 Asynchronous Functions.....18
 Strings as Return Values.....23
 Error Codes.....24

Obtaining Server Handle and Logging In 28

Host Server Operations 33

Retrieving Host Server Configuration Information34
 Managing Parallels Server Preferences.....37
 Searching for Parallels Servers40
 Managing Parallels Server Users43
 Managing Files on Host Server.....48
 Parallels Server License.....51
 Obtaining a Problem Report54

Virtual Machine Operations 56

Obtaining a List of Virtual Machines57
 Obtaining Virtual Machine Configuration Data60
 Determining Virtual Machine State62
 Starting, Stopping, Restarting a Virtual Machine64
 Suspending and Pausing a Virtual Machine68
 Creating a New Virtual Machine70
 Searching for Virtual Machines73
 Adding an Existing Virtual Machine77
 Cloning a Virtual Machine79
 Deleting a Virtual Machine.....81
 Modifying Virtual Machine Configuration82
 PrlVm_BeginEdit and PrlVm_Commit Functions83
 Obtaining a PHT_VM_CONFIGURATION handle84
 Name, Description, Boot Options.....85
 RAM Size86
 Hard Disks87
 Network Adapters.....89
 Managing User Access Rights91
 Working with Virtual Machine Templates93
 Obtaining a List of Templates94
 Creating a Template From Scratch96
 Converting a Regular Virtual Machine to a Template.....98
 Converting a Template to a Regular Virtual Machine.....99
 Creating a New Virtual Machine From a Template.....100

Events	102
Receiving and Handling Events.....	103
Responding to Server Questions.....	106
Performance Statistics	113
Obtaining Performance Report	114
Performance Monitoring.....	117
Index	122

CHAPTER 1

Getting Started

In This Chapter

Overview	6
System Requirements	7
Installation	8
Compiling Client Applications	9

Overview

Parallels Server SDK is a development kit used to build Parallels Server client applications. The SDK provides cross-platform C and Python APIs. The Python API is a wrapper of the C API. The SDK comprises the following components:

- C header files.
- Dynamic libraries.
- A Python package used to develop client applications in Python.
- Parallels Command-Line Tool (`prlctl`) - a sample client application that was developed using the Parallels Server SDK. The C source code and the executable version is included. This is a fully functional console application that can be used to perform a full range of operations on a Parallels server and its virtual machines.
- A collection of sample virtual machine configuration files in XML format for use with the Parallels Command-Line Tool. The configuration information contained in each file can be customized to suit a particular need.
- Parallels Server Programmer's Guide (this document).
- Parallels Server C API Reference.
- Parallels Server Python API Reference.

System Requirements

Mac OS X Clients

Hardware Requirements

- Intel-powered Core™ Duo or Core Solo Mac® Mini, iMac®, MacBook™, MacBook Pro, MacBook Air, Mac Pro, or Xserve.
- Ethernet or WiFi network adapter.

Software Requirements

- Mac OS X Tiger 10.4.8 or later.
- Mac OS X Leopard 10.5.2 or later.

Windows Clients

Hardware Requirements

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

Software Requirements

- Windows 2000 or higher.

Linux Clients

Hardware Requirements

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

Software Requirements

- Red Hat® Enterprise Linux WS4 (x32, x64).
- Red Hat Enterprise Linux AS4 (x32, x64).
- Red Hat Enterprise Linux ES4 (x32, x64).
- Red Hat Enterprise Linux 5 (x32, x64)
- CentOS 4.x (x32, x64).
- CentOS 5.0 (x32, x64).
- CentOS 5.1 (x32, x64).
- Ubuntu Server 7.10 (x32, x64).
- SUSE® Linux Enterprise Server 10 SP1 (x32, x64).

Common Network Requirements

To connect to a Parallels server using the provided APIs, your client computer must be able to establish a network connection with the host computer running the server. The client computer can be connected to a local area network via a wired or a wireless interface. Clients communicate with a Parallels server via TCP/IP. The server is listening on port 6400. Please make sure that the port is not blocked by a firewall.

Installation

Installing Parallels Server SDK on Mac OS X

- 1 Locate and open the Parallels Server SDK DMG package.
- 2 Double-click Install Parallels Server SDK to launch the installation wizard.
- 3 In the Welcome window, click Continue.
- 4 In the Easy Install on Macintosh HD window, click Install to perform basic installation of the Parallels Server SDK. The default installation path for Parallels Server is `/Applications/Parallels/`. The Python package files are copied to the standard Python path.
- 5 In the Finish Up window, click Restart to restart your Mac and finish the installation.

Installing Parallels Server SDK on Windows

- 1 Locate the Parallels Server package and double-click the `Parallels Server SDK.exe` file to launch the installation wizard.
- 2 In the Welcome window, click Next.
- 3 In the License Agreement window, carefully read the **Software End User License Agreement for Parallels Server**. If you agree with the terms of the license agreement, select **I accept the terms** in the license agreement and click **Next**. If you want to print the text of the license agreement for your records, click **Print**. You have to accept the Software License Agreement to proceed with the installation.
- 4 In the Destination Folder window, specify the folder where you want to install the SDK and click **Next**. By default, Parallels Server SDK is installed to the following location: `C:\Program Files\Parallels\Parallels Server SDK`. The Python package files are copied to the standard Python path.
- 5 In the Ready to Install The Program window, click **Install** to begin the installation.
- 6 When the installation is complete, click **Finish** to exit the wizard.

Installing Parallels Server SDK on Linux

- 1 Locate the Parallels Server installation package and launch `parallels-server-sdk-3.0.XXXX.XXXXX` to run the Parallels Server SDK installer. You can also open this file in a terminal.
- 2 Confirm that you want to install Parallels Server SDK by clicking **Run** when prompted.
- 3 In the **License Agreement** window, carefully read the Software License Agreement. If you agree with the stated terms and conditions, choose **Accept**. If you don't agree, choose **Decline**. You have to accept the Software License Agreement to proceed with the installation.
- 4 In the **Installation Completed** window, click **Exit** to quit the installer.

Compiling Client Applications

Mac OS X

The following is a sample make file that can be used to compile Parallels Server client applications on Mac OS X:

```
# set the appropriate path to SDK headers
SDK_INSTALL_PATH=/Library/Frameworks/ParallelsServer.framework/Versions/1.0/

OBJS = SdkWrap.o main.o
CXX = g++
CXXFLAGS = -I$(SDK_INSTALL_PATH)/Include
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)/Helpers/SdkWrap/SdkWrap.cpp
    $(CXX) -c -o $@ $(CXXFLAGS)
$(SDK_INSTALL_PATH)/Helpers/SdkWrap/SdkWrap.cpp

clean:
    @rm -f $(TARGET) $(OBJS)

.PHONY : all clean
```

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 directory where you've installed the Parallels Server SDK. The default installation directory is `C:\Program Files\Parallels\Parallels Server SDK`.
- 3** In the C/C++ -> **Precompiled Headers** -> **Create/Use Precompiled Header** section, select "Not Using Precompiled Headers" option.
- 4** Add the following files from the `Helpers\SdkWrap` subdirectory (located in the Parallels Server SDK directory) 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.

- 5** Add the following `include` directives to your program:

```
#include "Include\Parallels.h"
```

```
#include "Helpers\SdkWrap\SdkWrap.h"
```

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

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

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

Linux

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

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

OBJS = SdkWrap.o main.o
CXX = g++
CXXFLAGS = -I$(SDK_INSTALL_PATH)/include/parallels-server-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-server-
sdk/helpers/SdkWrap/SdkWrap.cpp
    $(CXX) -c -o $@ $(CXXFLAGS) $(SDK_INSTALL_PATH)/share/parallels-server-
sdk/helpers/SdkWrap/SdkWrap.cpp

clean:
    @rm -f $(TARGET) $(OBJS)

.PHONY : all clean
```

CHAPTER 2

API Basics

In This Chapter

Handles.....	13
Working with Results.....	15
Asynchronous Functions.....	18
Strings as Return Values	23
Error Codes	24

Handles

Parallels Server functionality can be accessed programmatically via an API included with the Parallels Server SDK. The Parallels Server API is a set of functions that operate on objects. Parallels Server objects are not accessed directly. Instead, references to these objects are used. These references are known as *handles*.

Handle Types

A `PRL_HANDLE` type is the only type of handle in the Parallels Server C API. It is a pointer to an integer, and is defined in `PrlTypes.h`. Within this guide, the terms *handle* and `PRL_HANDLE` will be used interchangeably.

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

A handle's 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 that operate on another (we may call it a "parent") handle. For example, a virtual machine handle is obtained by calling a function that operates on the server handle. A virtual device handle is obtained by calling a function that operates on the virtual machine handle, and so forth. The [Parallels Server C API Reference](#) guide contains a description of every available handle and explains how each particular handle can be obtained. The examples in this guide also demonstrate how to obtain handles of different types.

Freeing a Handle

Handles used within the Parallels Server API 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 client application is responsible for freeing any handles that are no longer required. 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

Handles used within the Parallels Server API 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 a Parallels server (the full example is provided later in this guide).

```
PRL_HANDLE hServer;
PRL_RESULT ret;

ret = PrlSrv_Create(&hServer);
if (PRL_FAILED(ret))
{
    fprintf(stderr, "PrlSrv_Create failed, error: %s",
            prl_result_to_string(ret));
    PrlHandle_Free(hServer);
    return -1;
}

// Get hServer handle type.
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);
```

Working with Results

Functions within the Parallels Server C API operate on objects. Some of these functions require handles as input parameters, others will return a handle as an output parameter, some functions will simply return an integer (`PRL_RESULT`) that can be used to determine the success or failure of the call, while others will return a `PRL_HANDLE` that will need to be operated on further. Some functions will generate events, and some functions will return multiple handles/results. This section will explain how to access the results of the different *types* of API functions.

Functions with a `PRL_RESULT` Return Value

A function that has a return value of type `PRL_RESULT` is a synchronous function. When a synchronous function completes, it returns control to the caller. The return value can then be examined to determine the success or failure of the call.

Consider the function `PrlSrv_Create`. The purpose of this function is to return a handle of type `PHT_SERVER`. This is returned as an output parameter - a pointer to a `PRL_HANDLE`, or `PRL_HANDLE_PTR`. This type of handle is required to access most of the functionality within the Parallels Server C API. The definition for `PrlSrv_Create` is:

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

An example of using `PrlSrv_Create` is:

```
PRL_HANDLE hServer;
PRL_RESULT res = PrlSrv_Create(&hServer);

if (PRL_FAILED(res))
{
    printf("PrlSrv_Create returned error: %s\n",
        prl_result_to_string(res));
    PrlHandle_Free(hServer);
    exit(ret);
}

// This point reached if PrlSrv_Create was successful.
// hServer is now a handle of type PHT_SERVER.
```

In the above example, a handle `hServer` is declared. `PrlSrv_Create` is passed a reference to this handle. The return value is examined to determine the success or failure of the `PrlSrv_Create` call. If the call succeeded, then `hServer` will be a reference to an object of type `PHT_SERVER` that can be used where ever a handle of type `PHT_SERVER` is required - for example, `PrlSrv_Login`.

Functions with `PRL_HANDLE` Return Value

A function that has a return type of `PRL_HANDLE` is an asynchronous function. When an asynchronous function is called, it will generate events (or jobs that are sent to the event handler/callback function). An asynchronous function can be used in a synchronous manner (and hence negate the need to write an event handler) by using `PrlJob_Wait`. For more information about asynchronous functions, consult the section called **Asynchronous Functions** (on page 18).

When calling an asynchronous function in a synchronous manner, the steps involved to obtain the end result are:

- 1 Call `PrlJob_Wait` to wait for the asynchronous function to complete.
- 2 Call `PrlJob_GetRetCode` to determine if the asynchronous function succeeded or failed.
- 3 Call `PrlJob_GetResult` to obtain a handle to the result object (a `PHT_RESULT` handle).
- 4 Call `PrlResult_GetParam` (or `PrlResult_GetParamsCount` and `PrlResult_GetParamByIndex` if the function returned more than one handle) to get the final result. The handle type returned depends on the function that was initially called.

The following example demonstrates using asynchronous function `PrlSrv_GetSrvConfig` synchronously, and the steps involved in obtaining a handle to an object of type `PHT_SERVER_CONFIG` (which is returned by `PrlSrv_GetSrvConfig`):

```
PRL_HANDLE hJob = PrlSrv_GetSrvConfig(hServer);

// Step 1 - emulate a synchronous call using PrlJob_Wait.
PRL_RESULT ret = PrlJob_Wait(hJob, 10000);

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

// Step 2 - determine if PrlSrv_GetSrvConfig succeeded or failed
// using PrlJob_GetRetCode.
PrlJob_GetRetCode(hJob, &nJobResult);

if (PRL_FAILED(nJobResult))
{
    printf("PrlSrv_GetSrvConfig returned error: %s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

// Step 3 - Obtain a handle to the result object (a handle
// of type PHT_RESULT) using PrlJob_GetResult.
PRL_HANDLE hResult;
ret = PrlJob_GetResult(hJob, &hResult);
PrlHandle_Free(hJob); // hJob no longer needed, free it.
if (PRL_FAILED(ret))
{
    printf("PrlJob_GetResult for job PrlSrv_GetSrvConfig returned error:
%s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hResult);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
}
```

```

    return -1;
}

// Step 4 - Get a handle to the final result (in this case,
// a handle of type PHT_SERVER_CONFIG) using PrlResult_GetParam.
// For PHT_RESULT handles that contain more than one result, first use
// PrlResult_GetParamsCount, then use PrlResult_GetParamByIndex.
PRL_HANDLE hServerConfig;
ret = PrlResult_GetParam(hResult, &hServerConfig);
PrlHandle_Free(hResult); // hResult no longer needed, free it.
if (PRL_FAILED(ret))
{
    printf("PrlResult_GetParam for job PrlSrv_GetSrvConfig returned error:
    %s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hServerConfig);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

// At this point, a handle of type PHT_SERVER_CONFIG is available, and
// functions that operate on PHT_SERVER_CONFIG handles can be used on
// the handle.

PRL_UINT32 nCpuCount = 0;
PrlSrvCfg_GetCpuCount(hServerConfig, &nCpuCount);
printf("Number of CPUs: %d\n", nCpuCount);

PRL_CPU_MODE CpuMode;
PrlSrvCfg_GetCpuMode(hServerConfig, &CpuMode);
printf("CPU Mode: %d bit\n", CpuMode == PCM_CPU_MODE_32 ? 32 : 64);

PRL_UINT32 nCpuSpeed = 0;
PrlSrvCfg_GetCpuSpeed(hServerConfig, &nCpuSpeed);
printf("CPU Speed: %.3f MHz\n", nCpuSpeed / 1000.0);

PRL_CHAR szHostOsString[1024];
PRL_UINT32 nHostOsStringSize = sizeof(szHostOsString);
PrlSrvCfg_GetHostOsStrPresentation(hServerConfig, szHostOsString,
&nHostOsStringSize);
printf("Host OS: %s\n", szHostOsString);

PrlHandle_Free(hServerConfig);

```

For an example of dealing with a handle of type PHT_RESULT that contains multiple results (more than a single handle), see section [Obtaining a List of Virtual Machines](#) (on page 57).

Asynchronous Functions

The Parallels Server API contains synchronous and asynchronous functions. Synchronous functions run in the same thread as the caller and have a return type of `PRL_RESULT`. When a synchronous function is called, it completes executing before returning control to the client application.

Some functions are time consuming, like starting a virtual machine on a remote host. It makes sense to execute such functions within their own thread (ie. asynchronously). Such functions are asynchronous in the Parallels Server API, and have a return type of `PRL_HANDLE`.

The general procedure for using asynchronous functions within the Parallels Server API is:

- 1 Register an event handler (callback function).
- 2 Analyse the results of events (jobs) within the callback function.
- 3 Handle the appropriate event in the callback function.
- 4 Un-register the event handler.

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 type can be found in `PrlApi.h`:

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

An example of a callback function:

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

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

The handle sent to the callback function can be of type `PHT_EVENT` or `PHT_JOB`. Each must be handled in a slightly different way. The first thing that should be done within an event handler is finding out what generated the event -- an event, or an asynchronous job. This can be achieved using `PrlHandle_GetType`.

To handle 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 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 machines.
- 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 machines on a specified Parallels Server. 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 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 machine 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;
            PrlEvent_GetParam(hHandle, 0, &hEventParameters);

            // Get handle to PHT_SYSTEM_STATISTICS.
            PRL_HANDLE hServerStatistics;
            PrlEvtPrm_ToHandle(hEventParameters, &hServerStatistics);
        }
    }
}
```

```

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

        PrlHandle_Free(hServerStatistics);
        PrlHandle_Free(hEventParameters);
    }
}
else if (nHandleType == PHT_JOB) // Job handle
{
    // Get 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 machine handle (PHT_VIRTUAL_MACHINE).
            PrlResult_GetParamByIndex(hJobResult, nParamIndex, &hVm);

            // Code goes here to obtain virtual machine 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 events/jobs generated from the Parallels server. It is the responsibility of the client application to identify and handle the relevant event(s).

```

// 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);
```

Using Asynchronous Functions Synchronously

It is possible to use an asynchronous function like a synchronous function by using `PrlJob_Wait` and `PrlJob_GetRetCode`. `PrlJob_Wait` takes two parameters - the handle of an asynchronous job that has been executed, and a timeout value in milliseconds. The timeout value is the maximum amount of time `PrlJob_Wait` will wait before timing out. If the asynchronous job completes before this timeout value, then `PrlJob_Wait` will exit and execution of the calling thread will continue.

The following code snippet illustrates how to synchronously use asynchronous function `PrlServer_Login`:

```
// Perform a server login (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

String values in the Parallels Server 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 the 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. At the same time, 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 API function:

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

The `PrlVmCfg_GetName` function above is a typical API function that returns a string value (in this case, the name of a virtual machine). The `hVmCfg` parameter is a handle to an object containing the virtual machine configuration information. The `sVmName` parameter is a `char` pointer. It is used as output that receives the virtual machine name. The variable must be initialized on the client 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 machine 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 machine name.
ret = PrlVmCfg_GetName(hVmCfg, sVmName, &nBufSize);

// Check for errors.
if (PRL_SUCCEEDED(ret))
{
    // Everything's OK, print the machine 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 machine name.
    ret = PrlVmCfg_GetName(hVmCfg, pBuf, &nSize);

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

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

```

Error Codes

All synchronous Parallels Server API functions return a `PRL_RESULT`, which is an integer value. The type definition for `PRL_RESULT` is in header file `prlerrors.h`. The return value is an error code that indicates the success or failure of the function call. The complete list of Parallels Server API error codes can be found in header file `prlerrorvalues.h`.

Error Codes for Asynchronous Functions

All Parallels Server API asynchronous functions return a handle. The error code (return value) for an asynchronous function can be extracted after it has completed executing by using the `PrlJob_GetRetCode` function.

Analyzing Return Values

The Parallels Server API provides the following helper macros to work with error codes:

<code>PRL_FAILED</code>	Returns true if the return value indicates failure, or false if the return value indicates success.
-------------------------	---

<code>PRL_SUCCEEDED</code>	Returns true if the return value indicates success, or false if the return value indicates failure.
<code>prl_result_to_string</code>	Returns a string representation of the error code.

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

```
// Attempt to create directory /tmp/TestDir on the server.
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 PrlSrv_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 for this is that sometimes errors contain dynamic parameters. The following example demonstrates obtaining 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

Obtaining Server Handle and Logging In

The following steps are required for any client application in order to perform operations on a Parallels server:

- 1 Load the Parallels Server SDK library using the `SdkWrap_Load` function.
- 2 Initialize the Parallels Server API using the `PrlApi_Init` function.
- 3 Create a server handle using the `PrlSrv_Create` function.
- 4 Call `PrlSrv_Login` (or `PrlSrv_LoginLocal`) with valid login credentials.

If these steps are executed without any errors, the handle created in step 3 will reference a server object (the handle type will be `PHT_SERVER`). A handle to a valid server object is required to access most of the functionality within the Parallels Server API. The `PrlSrv_Login` function (step 4) establishes a connection with a specified Parallels server and performs a login operation using specified credentials. The function operates on the server object created in step 3. Upon successful login, the object can be used to perform operations on the Parallels server.

To end the session with a Parallels server, the following steps must be performed before exiting an application:

- 1 Call `PrlSrv_Logoff` to log off the server.
- 2 Free the server handle using `PrlHandle_Free`.
- 3 Call `PrlApi_Deinit` to shut down the Parallels Server API engine.
- 4 Call `SdkWrap_Unload` to unload the Parallels Server SDK library.

C API Sample

The following is a complete sample program that demonstrates how to perform the steps described above. Before compiling and running the program on your machine, substitute the hard-coded IP address, user name, and password values with the ones appropriate to your Parallels Server setup. The `PrlSrv_Login` function is used for both local and remote login operations. To login to a Parallels server running on the local host, use `127.0.0.1` or `localhost` as the host parameter. The rest of the code can be left unchanged.

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

//
//

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

```

////////////////////////////////////
//
int main(int argc, char* argv[])
{
    // Variables for handles.
    PRL_HANDLE hJob = PRL_INVALID_HANDLE; // job handle
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE; // job result
    PRL_HANDLE hServer = PRL_INVALID_HANDLE; // server handle

    // Variables for return codes.
    PRL_RESULT ret = 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"
    #elif defined(_MAC_)
    #define SDK_LIB_NAME "libprl_sdk.dylib"
    #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.
    ret = PrlApi_Init(PARALLELS_API_VER);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlApi_Init returned with error: %s.\n",
            prl_result_to_string(ret));
        PrlApi_Deinit();
        SdkWrap_Unload();
        return ret;
    }

    // Create a server handle (PHT_SERVER).
    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;
    }

    // Perform a server login (asynchronous call).
    // Substitute the IP address, name, and password
    // with your own.
    hJob = PrlSrv_Login(
        hServer, // Server handle
        "10.30.22.159", // Server IP address
        "jdoe", // User name
        "secret", // Password
        0, // Previous session ID
        0, // Port number
        0, // Timeout
        PSL_NORMAL_SECURITY); // Security level.
    // The minimum allowable security

```

```

// level can be determined using
// PrlDispCfg_GetMinSecurityLevel.

// Wait for a maximum of 10 seconds for
// the job 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.
ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);

// First, check PrlJob_GetRetCode success/failure.
if (PRL_FAILED(ret))
{
    fprintf(stderr, "PrlJob_GetRetCode returned with error: %s\n",
        prl_result_to_string(ret));
    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" );
}

/*****
 *
 * To test sample functions provided in this guide,
 * call them from here. Most of the functions take a
 * server handle as a parameter. Pass the hServer
 * variable (the server handle that we obtain earlier
 * in the program).
 *
 *****/

// Logoff the Parallels server.
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;
}

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

// Check the PrlJob_GetRetCode success/failure.
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(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 Parallels API, and unload the SDK.
PrlApi_Deinit();
SdkWrap_Unload();

printf( "\n\nEnd of program.\n\n" );

// Wait for user to press a key...
char c;
scanf( &c, "%c" );

return 0;
}

```

Python API Sample

The following code snippet illustrates how to log in to a Parallels server using the Python API.

```

import prlsdkapi

consts = prlsdkapi.consts

server = prlsdkapi.Server()

```

```
server.login("10.30.20.73", "jdoe", "secret",  
            secur_level=consts.PSL_NORMAL_SECURITY).wait()
```

CHAPTER 4

Host Server Operations

This chapter describes the common tasks that can be performed on the host server. This includes the physical machine that hosts a Parallels server and the Parallels server itself.

In This Chapter

Retrieving Host Server Configuration Information	34
Managing Parallels Server Preferences.....	37
Searching for Parallels Servers	40
Managing Parallels Server Users	43
Managing Files on Host Server.....	48
Parallels Server License	51
Obtaining a Problem Report.....	54

Retrieving Host Server Configuration Information

The Parallels Server API provides a set of functions to retrieve detailed information about a physical machine hosting a Parallels server. This includes:

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

This information can be used when modifying Parallels server preferences, setting up devices inside virtual machines, or whenever you need to know what resources are available on the physical host.

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](#) (on page 28).

```
PRL_RESULT HostConfigSample(const 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;
}

// 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;
}

// 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.
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);
}
```

Managing Parallels Server Preferences

Parallels server preferences is a set of parameters that control the server default behaviour. The most of the important parameters are:

- Memory limits for the Parallels server itself.
- Memory limits and recommended values for virtual machines.
- Virtual network adapter(s) information.
- Default virtual machine directory (the directory where all new virtual machines are created by default).
- Minimum communication security level.

Operations on Parallels server preferences are performed using the `PHT_DISP_CONFIG` handle, which is obtained using the `PrlSrv_GetCommonPrefs` function. For the complete list of functions provided by the `PHT_DISP_CONFIG` object, see the [Parallels Server API Reference guide](#).

The following sample function demonstrates how to obtain a handle of type `PHT_DISP_CONFIG` and how to use its functions to retrieve and modify some of the Parallels server preferences. 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](#) (on page 28).

```
PRL_RESULT ServerPreferencesSample(const PRL_HANDLE &hServer)
{
    // Variables for handles.
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;
    PRL_HANDLE hDispConfig = PRL_INVALID_HANDLE;

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

    // An asynchronous call that obtains a handle
    // of type PHT_DISP_CONFIG.
    hJob = PrlSrv_GetCommonPrefs(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_GetCommonPrefs.
    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_DISP_CONFIG handle.
ret = PrlResult_GetParam(hJobResult, &hDispConfig);
PrlHandle_Free(hJobResult);
if (PRL_FAILED(ret))
{
    // Handle the error...
    return -1;
}

// Get the default virtual machine directory.
char sDefaultDir[1024];
PRL_UINT32 nBufSize = sizeof(sDefaultDir);
ret = PrlDispCfg_GetDefaultVmDir(hDispConfig,
                                sDefaultDir, &nBufSize);

if (PRL_FAILED(ret))
{
    fprintf(stderr, "Error: %s\n",
            prl_result_to_string(ret));
    PrlHandle_Free(hDispConfig);
    return -1;
}

// Get the recommended virtual machine memory size.
// This is one of the preferences set on the server level.
PRL_UINT32 pnMemSize = 0;
ret = PrlDispCfg_GetRecommendMaxVmMem(hDispConfig, &pnMemSize);
if (PRL_FAILED(ret))
{
    fprintf(stderr, "Error: %s\n",
            prl_result_to_string(ret));
    PrlHandle_Free(hDispConfig);
    return -1;
}

// Modify some of the server preferences.
// Begin edit.
hJob = PrlSrv_CommonPrefsBeginEdit(hServer);
ret = PrlJob_Wait(hJob, 1000);
if (PRL_FAILED(ret))
{
    fprintf(stderr, "Error: %s\n",
            prl_result_to_string(ret));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hDispConfig);
    return -1;
}

// Get the "begin edit" operation success code.
ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);

```

```
if (PRL_FAILED(ret))
{
    fprintf(stderr, "Error: %s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hDispConfig);
    return -1;
}
if (PRL_FAILED(nJobReturnCode))
{
    fprintf(stderr, "Error: %s\n",
        prl_result_to_string(nJobReturnCode));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hDispConfig);
    return -1;
}

PrlHandle_Free(hJob);

// Modify the recommended virtual machine memory size.
pnMemSize = 512;
ret = PrlDispCfg_SetRecommendMaxVmMem(hDispConfig, pnMemSize);
if (PRL_FAILED(ret))
{
    fprintf(stderr, "Error: %s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hDispConfig);
    return -1;
}

// Commit the changes to the server.
hJob = PrlSrv_CommonPrefsCommit(hServer, hDispConfig);
ret = PrlJob_Wait(hJob, 1000);
if (PRL_FAILED(ret))
{
    fprintf(stderr, "Error: %s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hDispConfig);
    return -1;
}

// Get the "begin edit" operation success code.
ret = PrlJob_GetRetCode(hJob, &nJobReturnCode);
if (PRL_FAILED(ret))
{
    fprintf(stderr, "Error: %s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hDispConfig);
    return -1;
}
if (PRL_FAILED(nJobReturnCode))
{
    fprintf(stderr, "Error: %s\n",
        prl_result_to_string(nJobReturnCode));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hDispConfig);
    return -1;
}

PrlHandle_Free(hDispConfig);
}
```

Searching for Parallels Servers

If you have multiple Parallels servers running on your network and don't know their exact locations and/or connection parameters, you can search for them using the `PrlSrv_LookupParallelsServers` function. The function returns the information as a list of handles of type `PHT_SERVER_INFO`, each containing the information about an individual server. The information includes server host name, port number, version of the OS that a host is running, Parallels server version number, and the global server ID (UUID). This information can then be used to establish a connection with the server of interest (you will have to know the correct user name and password in addition to the returned parameters).

The `PrlSrv_LookupParallelsServers` function can be executed asynchronously using the callback functionality, or it can be used synchronously. To use the function asynchronously, you must implement a callback function. The callback function pointer must then be passed to the `PrlSrv_LookupParallelsServers` as a parameter. During the search operation, the callback function will be called for every server found, and a handle of type `PHT_SERVER_INFO` containing the server information will be passed to it. Since searching a network can take a significant time, this is the recommended approach.

To use the `PrlSrv_LookupParallelsServers` function synchronously, pass a null pointer instead of the callback function pointer, and use `PrlJob_Wait` to wait for the job to complete. The returned job object will contain a list of `PHT_SERVER_INFO` objects.

The `PrlSrv_LookupParallelsServers` function can be executed without being logged in to a Parallels server.

The following samples demonstrate how to search local network for Parallels servers.

```
PRL_RESULT SearchServersSynch()
{
    // Variables for handles.
    PRL_HANDLE hJob = PRL_INVALID_HANDLE; // job handle
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE; // job result
    PRL_HANDLE hHostConfig = PRL_INVALID_HANDLE; // PHT_SERVER_CONFIG

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

    // Search for Parallels servers.
    hJob = PrlSrv_LookupParallelsServers(
        1000, // timeout
        NULL, // callback function (not used)
        NULL // user object pointer (not used)
    );

    // Wait for the job to complete.
    ret = PrlJob_Wait(hJob, 10000);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        PrlHandle_Free(hJob);
        fprintf(stderr, "Error: %s. \n",
            prl_result_to_string(ret));
        return -1;
    }
}
```

```

// Analyze the result of PrlSrv_LookupParallelsServers.
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 number of objects returned.
PRL_UINT32 nCount = 0;
PrlResult_GetParamsCount(hJobResult, &nCount);

// Iterate and obtain handle to each object.
for (PRL_UINT32 i = 0; i < nCount ; ++i)
{
    PRL_HANDLE hParam;
    PrlResult_GetParamByIndex(hJobResult, i, &hParam);

    PRL_CHAR sBuf[1024];
    PRL_UINT32 nBufSize = sizeof(sBuf);

    // Get the server host name.
    ret = PrlSrvInfo_GetHostName(hParam, sBuf, &nBufSize);

    if (PRL_SUCCEEDED(ret))
    {
        printf("Found server: %s\n", sBuf);
    }
    else
    {
        fprintf(stderr, "Error: %s \n",
            prl_result_to_string(ret));
    }

    PrlHandle_Free(hParam);
}
}

```

In the following example, the `PrlSrv_LookupParallelsServers` is called asynchronously. In order to that, we first have to implement a callback function (let's call it `ourCallback`):

```

static PRL_RESULT ourCallback(PRL_HANDLE hEvent, PRL_VOID_PTR pUserData)
{
    printf("%s: ", pUserData);

    // Get the server hostname.
    PRL_UINT32 nBufSize = 1024;
    PRL_CHAR sBuf[nBufSize];

```

```
PrlSrvInfo_GetHostName(hEvent, sBuf, &nBufSize);

// Get other server properties and process them here...

// The handle must be freed.
PrlHandle_Free(hEvent);
return rc;
}
```

The `PrlSrv_LookupParallelsServers` function can now be called as follows:

```
hJob = PrlSrv_LookupParallelsServers(
    1000,
    &ourCallback,
    (PRL_VOID_PTR) ("callback"));
```

Managing Parallels Server Users

Parallels Server doesn't have its own user database. It performs user authentication against the host operating system user database. However, it has a user registry, where the user information that relates to Parallels Server operations is kept. The information includes user UUID (Universally Unique ID), user name, the name and path of the virtual machine directory for the user, and two flags indicating if a user is allowed to modify server preferences and use management console application. A new user record is created in the registry for every OS user who logs in to a Parallels server for the very first time.

There are two API handles that are used to obtain information about Parallels Server users and to modify some of their profile parameters. These handles are `PHT_USER_INFO` and `PHT_USER_PROFILE`. Both handles are containers that contain information about a user. The difference between the two is `PHT_USER_PROFILE` contains information about currently logged in user while `PHT_USER_INFO` contains information about a user that can be specified. There are also some differences in the type of the information that each container contains.

Getting the information about the currently logged in user

The information about the currently logged in user can be retrieved using the functions of the `PHT_USER_PROFILE` handle. The following sample demonstrates how to obtain the handle and how to use its functions to retrieve the user information. The sample also shows how to set up a default virtual machine directory for the user. Parallels Server automatically assigns a default virtual machine directory (the directory where new virtual machines are created) for every user. If needed, a user can specify a different default directory for his/her virtual machines. At the time of this writing, this is the only property of the Parallels Server user profile that can be modified. The important thing to remember is that when modifying a user profile, the operation must begin with the `PrlSrv_UserProfileBeginEdit` function call and end with the `PrlSrv_UserProfileCommit` call. These two functions are used to prevent collisions with other clients trying to modify the same user profile at the same time.

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

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Get user info from the server.
    hJob = PrlSrv_GetUserProfile(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_GetUserProfile.
    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 user profile handle (PHT_USER_PROFILE) from
// the result.
ret = PrlResult_GetParam(hJobResult, &hUserProfile);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hJobResult);
    return -1;
}

// Free job result handle.
PrlHandle_Free(hJobResult);

// See if the user is allowed to modify
// the Parallels server preferences.
PRL_BOOL bCanChange = PRL_FALSE;
ret = PrlUsrCfg_CanChangeSrvSets(hUserProfile, &bCanChange);
printf("Can modify server preferences: %d\n", bCanChange);

// See if the user is allowed to use management
// console application.
ret = PrlUsrCfg_CanUseMngConsole(hUserProfile, &bCanChange);
printf("Can use management console: %d\n", bCanChange);

// Get the default virtual machine folder
// for the user.
PRL_CHAR sBufFolder[1024];
PRL_UINT32 nBufSize = sizeof(sBufFolder);
ret = PrlUsrCfg_GetDefaultVmFolder(hUserProfile, sBufFolder, &nBufSize);

// If sBufFolder contains an empty string then this user
// does not have a default virtual machine folder and is
// currently using the default virtual machine folder set
// for this Parallels server. If this is the case, retrieve
// that folder.
if (sBufFolder == "")
{
    ret = PrlUsrCfg_GetVmDirUuid(hUserProfile, sBufFolder, &nBufSize);
}

printf("VM folder: %s\n", sBufFolder);

// Modify the name and location of the virtual
// machine folder.
// This operation must begin with the
// PrlSrv_UserProfileBeginEdit that marks the
// beginning of the operation. This is done to

```

```

// prevent collisions with other sessions trying to
// modify the same profile at the same time.
hJob = PrlSrv_UserProfileBeginEdit(hServer);
ret = PrlJob_Wait(hJob, 1000);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hJob);
    return -1;
}
// Analyze the result of PrlSrv_UserProfileBeginEdit.
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;
}
// Set the new virtual machine folder.
// The folder must already exist on the server.
ret = PrlUsrCfg_SetDefaultVmFolder(hUserProfile,
"/Users/Shared/Parallels/JDoe");
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hUserProfile);
    return -1;
}
// Finally, commit the changes to the server.
hJob = PrlSrv_UserProfileCommit(hServer, hUserProfile);
ret = PrlJob_Wait(hJob, 1000);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hJob);
    return -1;
}
// Analyze the result of PrlSrv_UserProfileCommit.
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;
}

PrlHandle_Free(hUserProfile);
}

```

Getting the information about a particular user

The information about a particular Parallels server user can be obtained using the functions of the PHT_USER_INFO handle. The handle can be obtained using one of the following functions: PrlSrv_GetUserInfo or PrlSrv_GetUserInfoList. The first function takes the user UUID as an input parameter and returns a single handle of type PHT_USER_INFO containing the user information. The second function returns information about all users that exist in the Parallels server user registry. The information is returned as a list of handles of type PHT_USER_INFO.

The following sample uses the PrlSrv_GetUserInfoList function to obtain information about all users in the Parallels server user registry. It then iterates through the returned list of PHT_USER_INFO handles and retrieves information about individual users.

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

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Get user info from the server.
    hJob = PrlSrv_GetUserInfoList(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_GetUserInfoList.
    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 parameter count (the number of PHT_USER_INFO
    // handles in the result set).
    PRL_UINT32 nParamCount = 0;
    ret = PrlResult_GetParamsCount(hJobResult, &nParamCount);

    // Iterate through the list obtaining
    // a handle of type PHT_USER_INFO for
```

```
// each user.
for (PRL_UINT32 i = 0; i < nParamCount; ++i)
{
    ret = PrlResult_GetParamByIndex(hJobResult, i, &hUserInfo);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
        return -1;
    }

    // Get user UUID.
    PRL_CHAR sBufID[1024];
    PRL_UINT32 nBufSize = sizeof(sBufID);
    ret = PrlUsrInfo_GetUuid(hUserInfo, sBufID, &nBufSize);
    printf("UUID: %s\n", sBufID);

    // Get user name.
    PRL_CHAR sBufName[1024];
    nBufSize = sizeof(sBufName);
    PrlUsrInfo_GetName(hUserInfo, sBufName, &nBufSize);
    printf("Name: %s\n", sBufName);

    // Get default virtual machine folder
    // for the user.
    PRL_CHAR sBufFolder[1024];
    nBufSize = sizeof(sBufFolder);
    PrlUsrInfo_GetDefaultVmFolder(hUserInfo, sBufFolder, &nBufSize);
    printf("VM folder: %s\n", sBufFolder);

    // See if the user is allowed to modify
    // the Parallels server preferences.
    PRL_BOOL bCanChange = PRL_FALSE;
    PrlUsrInfo_CanChangeSrvSets(hUserInfo, &bCanChange);
    printf("Can modify server preferences: %d\n\n", bCanChange);

    PrlHandle_Free(hUserInfo);
}
PrlHandle_Free(hJobResult);
}
```

Managing Files on Host Server

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

- 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 a directory listing from the host server

The directory listing is obtained using the `PrlSrv_FsGetDirEntries` function. The function returns a handle of type `PHT_REMOTE_FILESYSTEM_INFO` containing 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.

```
PRL_RESULT DirectoryListSample(PRL_HANDLE hServer, PRL_CONST_STR path)
{
    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 server.
    // 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 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.
    DirectoryListSample(hServer, sBuf);
}

```

```
}  
PrlHandle_Free(hParentDirectory);  
PrlHandle_Free(hJob);  
}
```

Parallels Server License

Parallels Server license information can be retrieved using the `PrlSrv_GetLicenseInfo` function. The function returns a handle of type `PHT_LICENSE` containing the license details. The handle provides a set of functions to retrieve the details. To install or update a license on a Parallels server, use the `PrlSrv_UpdateLicense` function.

The following sample function demonstrates how to obtain license information and how to install a license on a Parallels server.

```
PRL_RESULT UpdateLicenseSample(PRL_HANDLE hServer)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE;
    PRL_HANDLE hLicense = PRL_INVALID_HANDLE;
    PRL_HANDLE hUpdateLicense = PRL_INVALID_HANDLE;

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Get the license info from the server.
    hJob = PrlSrv_GetLicenseInfo(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_GetUserInfoList.
    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 parameter from job result.
    ret = PrlResult_GetParam(hJobResult, &hLicense);
    PrlHandle_Free(hJobResult);
    if (PRL_FAILED(ret))
    {
        // Handle the error...
```

```

    return -1;
}

// Get company name.
PRL_CHAR sCompany[1024];
PRL_UINT32 nCompanyBufSize = sizeof(sCompany);
ret = PrlLic_GetCompanyName(hLicense, sCompany, &nCompanyBufSize);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hLicense);
    return -1;
}
printf("Company: %s\n", sCompany);

// Get user name.
PRL_CHAR sUser[1024];
PRL_UINT32 nUserBufSize = sizeof(sUser);
ret = PrlLic_GetUserName(hLicense, sUser, &nUserBufSize);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hLicense);
    return -1;
}
printf("User: %s\n", sUser);

// Get license key.
PRL_CHAR sKey[1024];
PRL_UINT32 nKeyBufSize = sizeof(sKey);
ret = PrlLic_GetLicenseKey(hLicense, sKey, &nKeyBufSize);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hLicense);
    return -1;
}
printf("Key: %s\n", sKey);

// See license type.
PRL_BOOL isTrial = PRL_TRUE;
ret = PrlLic_IsTrial(hLicense, &isTrial);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hLicense);
    return -1;
}
printf("Trial: %d\n", isTrial);

PrlHandle_Free(hLicense);

// Update the license info.
// Here, we use the same license information that we
// retrieved earlier. Normally, you would use the
// information that you received from
// your Parallels Server distributor.
hUpdateLicense = PrlSrv_UpdateLicense(hServer, sKey,
                                     sUser, sCompany);

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

```

```
PrlHandle_Free(hUpdateLicense);  
}
```

Obtaining a Problem Report

If you experience problems with virtual machine operations, you can obtain a problem report from the server, which can then be sent to the Parallels Server technical support for evaluation. A report is generated by the server and contains technical data about your Parallels Server 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 a 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 server.
    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.
PRL_CHAR_PTR sReportData =(PRL_CHAR_PTR)malloc(nBufSize*sizeof(PRL_CHAR));
ret = PrlResult_GetParamAsString(hJobResult, sReportData, &nBufSize);
if (PRL_FAILED(ret))
{
    // Handle the error...
    PrlHandle_Free(hJobResult);
    return -1;
}

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

CHAPTER 5

Virtual Machine Operations

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

In This Chapter

Obtaining a List of Virtual Machines.....	57
Obtaining Virtual Machine Configuration Data.....	60
Determining Virtual Machine State	62
Starting, Stopping, Restarting a Virtual Machine	64
Suspending and Pausing a Virtual Machine.....	68
Creating a New Virtual Machine	70
Searching for Virtual Machines	73
Adding an Existing Virtual Machine	77
Cloning a Virtual Machine.....	79
Deleting a Virtual Machine.....	81
Modifying Virtual Machine Configuration	82
Managing User Access Rights	91
Working with Virtual Machine Templates.....	93

Obtaining a List of Virtual Machines

A physical node running Parallels Server can contain zero or more virtual machines (VMs). Each virtual machine has a number of configuration parameters, including operating system type, number of CPUs, amount of physical memory, virtual disk space etc.

`PrlSrv_GetVmList` is used to obtain a list of virtual machines available on a server. The return value will be a handle to a job object (a reference to an object of type `PHT_JOB`). `PrlJob_GetResults` is then used to obtain a handle to a results object (a reference to an object of type `PHT_RESULT`). A result object is a container that contains zero or more handles. The type of the handles within the results object depends on the function that was called to obtain the job handle. Each individual handle (result) within a result object is referred to as a *parameter*. Functions that operate on `PHT_RESULT` objects are prefixed with `PrlResult_`.

To obtain a list of available virtual machines on a Parallels server (with error checking omitted):

- 1 Call `PrlSrv_GetVmList`. This will return a job handle.
- 2 Call `PrlJob_GetResults`. This will return a results handle.
- 3 Free the job handle that was obtained using `PrlSrv_GetVmList`, as it is no longer needed.
- 4 Get the number of handles (ie. the number of virtual machines) returned using `PrlResult_GetParamsCount`.
- 5 For each result (parameter) within the result object, do:
 - 6 Extract a virtual machine handle using `PrlResult_GetParamByIndex`.
 - 7 Obtain the name of the virtual machine using `PrlVmCfg_GetName`.
 - 8 Free the virtual machine handle using `PrlHandle_Free`.
 - 9 Free the results handle using `PrlHandle_Free`.

C API Sample

The following sample function implements the steps above demonstrating how to:

- obtain a list of virtual machines from the server.
- obtain a handle of type `PHT_VIRTUAL_MACHINE` for each machine in the list.
- use the virtual machine handle functions to retrieve information about the machine.

The rest of the examples in this chapter assume that the virtual machine handle has been obtained as shown in the example provided here.

```
PRL_RESULT DisplayVmList(const 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 a list of the available virtual machines.
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(hJobResult);
    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 machines returned).
PRL_UINT32 nParamsCount = 0;
ret = PrlResult_GetParamsCount(hJobResult, &nParamsCount);
for (PRL_UINT32 i = 0; i < nParamsCount; ++i)
{
    PRL_HANDLE hVm; // virtual machine handle

    // Get a handle to 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 machine information.
    // As an example, we will get the virtual machine 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",

```


Obtaining Virtual Machine Configuration Data

If a virtual machine handle (a handle of type `PHT_VIRTUAL_MACHINE`) references a valid virtual machine object, configuration data about the virtual machine can be extracted by using the `PHT_VM_CONFIGURATION` functions. `PHT_VM_CONFIGURATION` functions are prefixed with `PrlVmCfg_`. Virtual machine functions that can be used to get data about a virtual machine are prefixed with `PrlVmCfg_Get`. To use the functions, a handle of type `PHT_VM_CONFIGURATION` must be obtained from the virtual machine object by using the `PrlVm_GetConfig` function. The following code snippet obtains the virtual machine name, operating system type, operating system version, RAM size, default HDD size, and CPU count for a virtual machine referenced by `hVm`:

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

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

// Call once with NULL (PRL_INVALID_HANDLE) to get size of buffer
// to allocate for VM name.
PrlVmCfg_GetName(hVmCfg, PRL_INVALID_HANDLE, &nVmNameSize);
printf("Vm name size: %d\n", nVmNameSize);

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

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

// Free the memory allocated for the VM name.
free(szVmName);

// Get the OS type.
PRL_UINT32 nOsType = 0;
PrlVmCfg_GetOsType(hVmCfg, &nOsType);
printf("OS Type: %.8x\n", 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);
```

Determining Virtual Machine State

To determine the current state of a virtual machine, first obtain a handle to the virtual machine as described in the [Obtaining a List of Virtual Machines](#) section (see page 57). 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 machine state as an enumerator from the `VIRTUAL_MACHINE_STATE` enumeration. The enumeration defines every possible state and transition applicable to a virtual machine.

The following example demonstrates how obtain state information for the specified virtual machine.

```
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 machine 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 machine 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;
    }

    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_RESTOREING:
            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("Unhandled state\n");
    }

    PrlHandle_Free(hVmCfg);
    PrlHandle_Free(hVmInfo);

    return 0;
}
```

Starting, Stopping, Restarting a Virtual Machine

To start, stop, or restart a virtual machine, a handle to the virtual machine must first be obtained. `PrlVm_Start`, `PrlVm_Stop`, and `PrlVm_Reset` can then be used to start, stop, or restart a virtual machine, respectively. The virtual machine must be registered on the Parallels server in order to perform these operations.

To obtain a handle to a virtual machine by the specified machine name, the following sample function can be used:

```
// -----
// Return a handle to VM with name szVmName.
// Returns:
// PRL_HANDLE to object of type PHT_VIRTUAL_MACHINE.
// PRL_ERR_INVALID_HANDLE is returned if the VM was not found.
// -----
PRL_HANDLE GetVmByName(char *szVmName, PRL_HANDLE &hServer)
{
    PRL_HANDLE hResult;
    PRL_RESULT nJobResult;

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

    // Wait for a maximum of 10 seconds for PrlSrv_GetVmList.
    PRL_RESULT 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);
        PrlHandle_Free(hServer);
        exit(ret);
    }

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

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

```

PrlHandle_Free(hJob);

// Iterate through the results (list of virtual machines returned).
PRL_UINT32 nParamsCount = 0;
ret = PrlResult_GetParamsCount(hResult, &nParamsCount);
for (PRL_UINT32 i = 0; i < nParamsCount; ++i)
{
    PRL_HANDLE hVm;

    // Get a handle to result i.
    PrlResult_GetParamByIndex(hResult, i, &hVm);

    // Get the name of the virtual machine for result i.
    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));
    }

    // printf("VM extracted with name '%s'\n\n", szVmNameReturned);

    // If the name of the virtual machine at this index is equal to
szVmName,
    // then return the handle this virtual machine handle.
    if (strcmp(szVmName, szVmNameReturned) == 0)
    {
        PrlHandle_Free(hResult);
        return hVm;
    }

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

// The virtual machine being searched for was not found, so
// free the results and return an invalid handle.
PrlHandle_Free(hResult);

return PRL_INVALID_HANDLE;
}

```

An example of using the above GetVmByName function:

```

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

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

```

If GetVmByName successfully returns a handle to a virtual machine, the virtual machine can be started, stopped, or restarted.

An example of starting a virtual machine:

```

PRL_RESULT nVmStartResult;

```

```

PRL_HANDLE hVmStartJob = PrlVm_Start(hVm);
PrlJob_Wait(hVmStartJob, 10000);
PrlJob_GetRetCode(hVmStartJob, &nVmStartResult);
if (PRL_FAILED(nVmStartResult))
{
    fprintf(stderr, "PrlVm_Start failed with error: %s\n",
        prl_result_to_string(nVmStartResult));
    PrlHandle_Free(hVm);
    PrlHandle_Free(hVmStartJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}
else
{
    printf("Successfully started VM \"%s\".\n", szVmName);
}

```

An example of stopping a virtual machine:

```

PRL_RESULT nVmStopResult;
PRL_HANDLE hVmStopJob = PrlVm_Stop(hVm, PRL_FALSE);
PrlJob_Wait(hVmStopJob, 10000);
PrlJob_GetRetCode(hVmStopJob, &nVmStopResult);
if (PRL_FAILED(nVmStopResult))
{
    fprintf(stderr, "PrlVm_Stop failed with error: %s\n",
        prl_result_to_string(nVmStopResult));
    PrlHandle_Free(hVm);
    PrlHandle_Free(hVmStopJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}
else
{
    printf("Successfully stopped virtual machine \"%s\".\n", szVmName);
}

```

Note: Stopping a virtual machine is not the same as performing an operating system shutdown operation. When a virtual machine is stopped, it is a *cold stop* (ie. 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 machine 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 machine will be properly shut down.

An example of restarting (or resetting) a virtual machine:

```

PRL_RESULT nVmResetResult;
PRL_HANDLE hVmResetJob = PrlVm_Reset(hVm);
PrlJob_Wait(hVmResetJob, 10000);
PrlJob_GetRetCode(hVmResetJob, &nVmResetResult);
if (PRL_FAILED(nVmResetResult))
{
    fprintf(stderr, "PrlVm_Rest failed with error: %s\n",
        prl_result_to_string(nVmResetResult));
    PrlHandle_Free(hVm);
    PrlHandle_Free(hVmResetJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

```

Note: Resetting (or restarting) a virtual machine is not the same as performing an operating system restart operation, it is the same as pressing the reset button on a computer. Any unsaved data will be lost.

Suspending and Pausing a Virtual Machine

Suspending a Virtual Machine

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

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

The following example will suspend a virtual machine, that is running, called *Windows XP - 01*. This example uses the `GetVmByName` function from the *Starting, Stopping Restarting a Virtual Machine* section.

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

// Get a handle to virtual machine with name szVmName.
PRL_HANDLE hVm = GetVmByName((char*)szVmName, hServer);
if (hVm == PRL_INVALID_HANDLE)
{
    fprintf(stderr, "Virtual machine \"%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 virtual machine that is in a suspended state can be stopped completely (put into a stopped state) using `PrlVm_DropSuspendedState`.

Pausing a Virtual Machine

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

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

to:

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

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

Resuming / Continuing a Virtual Machine

A virtual machine that has been placed into a suspended or paused state can be restarted using `PrlVm_Start`. Alternatively, `PrlVm_Resume` can be used to resume execution of a virtual machine that is in a suspended state.

Dropping Suspended State

A virtual machine that is in a suspended state can be shut down using `PrlVm_DropSuspendedState`. If this is used, any unsaved data will be lost.

Creating a New Virtual Machine

The first step in creating a new virtual machine is to create a blank virtual machine and register it with a Parallels server. A blank virtual machine is the equivalent of a hardware box with no operating system installed on the hard drive. Once a blank virtual machine is created and added to the Parallels server registry, 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 machine without knowing all of the details about configuring a virtual machine for a particular operating system type.

The steps involved in creating a typical virtual machine are:

- 1 Obtain a handle to a new virtual machine object using the `PrlSrv_CreateVm` function.
- 2 Obtain a handle of type `PHT_VM_CONFIGURATION` by calling `PrlVm_GetConfig` function. The handle is used for manipulating virtual machine configuration settings.
- 3 Set the default configuration parameters based on the version of the OS that you will later install on the machine. 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 OS version 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 machine to their physical counterparts on the host server. The devices includes floppy disk drive, CD drive, serial and parallel ports, sound, 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 virtual machine 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 machine, 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 machine with a Parallels server using the `PrlVm_Reg` function. This step will create the necessary virtual machine files on the host server and will add the machine to the Parallels server virtual machine registry. The directory containing the virtual machine files will have the same name as the name you've chosen for your virtual machine. The directory will be created in the default virtual machine root directory for this server. If you would like to create the virtual machine 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 machine. The sample assumes that the client program has already obtained a server object handle (`hServer`) and performed the server 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 machine 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 handle to the server configuration object from it.
ret = PrlJob_GetResult(hJob, &hResult);
PRL_HANDLE hSrvCfg;
PrlResult_GetParam(hResult, &hSrvCfg);

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

// Now that we have the host server configuration data,
// we can set the default configuration for the new virtual machine.
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 machine 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 machine configuration.
// These calls are optional. You may remove them to use the default values.
//

// Set RAM size for the machine 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;
ret = PrlVmCfg_GetHardDisk(hVmCfg, 0, &hHDD);
ret = PrlVmDevHd_SetDiskSize(hHDD, 20000);

// Create and register the machine on the server.
// 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 Machines

A physical machine hosting Parallels Server may have virtual machines on its hard drive that are not currently registered with the server. This can happen when a virtual machine is removed from the server registry but its files are kept on the drive, or when a virtual machine files are manually copied to the drive from another computer. Parallels Server API provides the `PrlSrv_StartSearchVms` function that can be used to find such virtual machines 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 machines. It then returns a list of `PHT_FOUND_VM_INFO` handles, each containing information about an individual virtual machine that it finds. You can then decide whether you want to keep the machine as-is, register it with the Parallels server, or remove it from the hard drive.

Since the search operation may take a long time (depends on the size of the specified directory tree), the `PrlSrv_StartSearchVms` function should be executed using the callback functionality (see page 18). The callback function will be called for every virtual machine found and a single instance of the `PHT_FOUND_VM_INFO` handle will be passed to it. As we discussed earlier in this guide (see page 18), 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`, continue to the next step.
- 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 machine information.
- 4 Use functions of the `PHT_FOUND_VM_INFO` object to obtain the name and path where the virtual machine files were found, the virtual machine name, OS version, and some other virtual machine 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;
    PRL_HANDLE hParam; // this will receive the event parameter 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;
    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 machine 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 machine 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 machine is found. If there are no unused virtual machines in the specified directory tree, then the function will never be called (it will still be called at least once as a result of the started job and will receive the job object).

Receiving the search results synchronously

It is also possible to use this function synchronously using the `PrlJob_Wait` function (see page 18). 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 machine.
PRL_UINT32 nIndex, nCount;
PrlResult_GetParamsCount(hJobResult, &nCount);
for(nIndex = 0; nIndex < nCount ; nIndex++)
{
    PrlResult_GetParamByIndex(hJobResult, nIndex, &hFoundVmInfo);

    // Get the virtual machine 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 machine 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 Machine

When a virtual machine is created on a Parallels Server, a directory is created on the host (the computer running Parallels Server). Files that constitute the virtual machine are written to that directory. The path, directory name, and files created will vary depending on operating system and configuration options/parameters used during installation. For example, an installation of a Windows XP virtual machine may create a directory named `WinXP02` (if `WinXP02` is what the virtual machine was named). All the files required for this Windows XP virtual machine to function will reside in the `WinXP02` directory created.

Every virtual machine directory will contain a file called `config.pvs`, which is an XML file containing configuration data. When registering an existing virtual machine, `config.pvs` is what Parallels Server will look for. Multiple copies of a virtual machine directory can be made and registered with Parallels Server, but each virtual machine will require a unique name. It is possible to copy a virtual machine (directory) to another computer, register the virtual machine, and start the virtual machine - as long as the computer the virtual machine is copied to is running the same operating system as the computer the virtual machine was copied from.

To register a virtual machine with Parallels Server, use `PrlSrv_RegisterVm`. The following example demonstrates registering a virtual machine that resides in directory `/Users/Shared/Parallels/WinXP02`:

```
PRL_CHAR_PTR szVmToRegister = "/Users/Shared/Parallels/WinXP02";
hJob = PrlSrv_RegisterVm(
    hServer,          // Handle to a Parallels Server (PrlSrv_Login
called).
    szVmToRegister,  // Path to the virtual machine directory to
register.
    PRL_TRUE);      // Non-interactive mode.
ret = PrlJob_Wait(hJob, 10000);

// Check that PrlJob_Wait succeeded.
if (PRL_FAILED(ret))
{
    printf("PrlJob_Wait failed for PrlSrv_RegisterVm with error: %s\n",
        prl_result_to_string(ret));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

// Get and check the return code for PrlSrv_RegisterVm.
PrlJob_GetRetCode(hJob, &nJobResult);

if (PRL_FAILED(nJobResult))
{
    printf("PrlSrv_RegisterVm returned error: %s\n",
        prl_result_to_string(nJobResult));
    PrlHandle_Free(hJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}
else
{
```

```
printf("Virtual machine '%s' was successfully registered.",  
      szVmToRegister);  
}
```

Cloning a Virtual Machine

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

- 1 A valid handle of type `PHT_VIRTUAL_MACHINE` containing information about the source virtual machine.
- 2 A unique name for the new virtual machine (the name is NOT generated automatically).
- 3 The name of the directory where the virtual machine 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 machine as a valid virtual machine or as a template. `PRL_TRUE` indicates to create a template. `PRL_FALSE` indicates to create a virtual machine. See the [Working with Virtual Machine Templates](#) (on page 93) section for more virtual machine info and examples.

The source virtual machine must be registered with the Parallels server before it can be cloned.

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

```
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 machine name.
    // Get the name of the original VM and use
    // it in the new virtual machine 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 server.
    // Empty string indicates that the default
    // directory should be used.
    PRL_CHAR_PTR new_vm_root_path = "";

    // Virtual machine or template?
    // The cloning functionality allows to create
    // a new virtual machine or a new template.
```

```
// True indicates to create a template.
// False indicates to create a virtual machine.
// We are creating a virtual machine.
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 a Virtual Machine

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

- 1 Un-register the virtual machine using `PrlVm_Unreg`. This will remove the virtual machine from the list of available virtual machines on the Parallels server. Once a virtual machine has been unregistered it is not possible to use it. The directory containing the virtual machine files will remain on the hard drive of the Parallels server, and the virtual machine can later be re-registered and used.
- 2 Delete the virtual machine using `PrlVm_Delete`. The virtual machine will be unregistered from the Parallels server, and the directory, or specified files belonging to the virtual machine, will be deleted.

The following example demonstrates un-registering a virtual machine from a Parallels server. Note that this example makes use of a function called `GetVmByName` that can be found in the Obtaining a List of Virtual Machines section.

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

// Get a handle to virtual machine 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 machine.
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 machine and deleting `config.pvs` within the virtual machine directory:

```
// Delete a virtual machine 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 machine and the virtual machine directory (all files belonging to the virtual machine), omit the line:

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

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

Modifying Virtual Machine Configuration

The Parallels Server API provides a complete set of functions to modify the configuration parameters for an existing virtual machine. You can find the list of the available functions in the [Parallels Server 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 to modify the virtual machine 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 a Parallels server (see [Logging on to a Parallels Server](#) (on page 28)).
- You've already obtained a handle to the virtual machine that you would like to modify (see [Obtaining a List of Virtual Machines](#) (on page 57)).

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

PrlVm_BeginEdit and PrlVm_Commit Functions

All virtual machine 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 machine.

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

// 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 machine.
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 a PHT_VM_CONFIGURATION handle

Before you can use any of the virtual machine configuration management functions, you have to obtain a handle of type `PHT_VM_CONFIGURATION`. The handle is obtained from the virtual machine 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 machine configuration settings. As usual, don't forget to free the handle when it is no longer needed.

Name, Description, Boot Options

The virtual machine 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_HANDLE hJobCommit;
PRL_RESULT nJobRetCode;

// 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.
//
PRL_UINT32 nDevCount;    // Device count.
PRL_HANDLE hDevice;     // A handle identifying the device.
PRL_DEVICE_TYPE devType; // Device type.

// 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 machine is performed using the `PrlVmCfg_SetRamSize` function. The first parameter is the virtual machine 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 machine 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_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 machine. 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 machine.

```
PRL_HANDLE hJobBeginEdit;
PRL_HANDLE hJobCommit;
PRL_RESULT nJobRetCode;

// Timestamp the beginning of the configuration changes operation.
// The hVm specifies the virtual machine 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;
```

```

ret = PrlVmCfg_CreateVmDev(
    hVmCfg, // The target virtual machine.
    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 machine 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 machine. 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 machine 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;
ret = PrlVmCfg_CreateVmDev(
    hVmCfg, // Target virtual machine.
    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 machine 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 machine and will appear in the list of the available devices.

Network Adapters

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

- **Host-only networking.** A virtual machine can communicate with the host and other virtual machines, but not with external networks.
- **Shared networking.** Uses the NAT feature. A virtual machine 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 machine appears as a standalone computer on the network.

Host-only and Shared Networking

The following example illustrates how to add adapters using the host-only and shared networking (both types are created similarly).

```
PRL_HANDLE hJobBeginEdit;
PRL_HANDLE hJobCommit;
PRL_RESULT nJobRetCode;

// Timestamp the beginning of the configuration changes operation.
// The hVm specifies the virtual machine 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 hNet;
ret = PrlVmCfg_CreateVmDev(
    hVmCfg, // Target virtual machine.
    PHT_VIRTUAL_DEV_NET_ADAPTER, // Device type.
    &hNet); // Device handle.

// For host-only networking, set the device emulation type
// to PDT_USE_REAL_DEVICE. This is simply the enumerator
// specifying this networking type.
// Uncomment the following function call to create the
// "host-only" adapter.
// PrlVmDev_SetEmulatedType(hNet, PDT_USE_REAL_DEVICE);

// For shared networking, set the device emulation type
// to PDT_USE_IMAGE_FILE, which is also an enumerator
// used as a shared networking identifier.
// Un-comment the following function call to create the
// "shared" adapter.
// PrlVmDev_SetEmulatedType(hNet, PDT_USE_IMAGE_FILE);

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

Managing User Access Rights

User authorization (determining user access rights) in Parallels Server is performed using OS-level file access permissions. Essentially, a virtual machine is a set of files that a user can read, write, and execute. When determining access rights of a user for a particular virtual machine, Parallels Server looks at the rights the user has on the virtual machine files and uses this information to allow or deny privileges. The **Parallels Server Administration Guide** has a section that describes the Parallels Server tasks in relation to the file access rights. Using this information, you can determine the tasks that a user is allowed to perform based on the file access rights the user has. The same goal can also be accomplished programmatically through Parallels Server API.

The Parallels Server 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 machine 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 machine 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 server has full access rights to all virtual machines. A non-administrative user has full rights to the machines created by him/her and no rights to any other virtual machines by default (these machines will not even be included in the result set when the user requests a list of virtual machines from the server). The server administrator can grant virtual machine 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 machine 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 (the users currently present in the Parallels server user registry (see page 43)). To determine the current access level for other users on a particular virtual machine, use the `PrlAcl_GetAccessForOthers` function. For the complete set of user access management functions, see the `PHT_ACCESS_RIGHTS` object description in the **Parallels Server API Reference** guide.

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

```
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 machine access rights for other users
// to PAO_VM_SHARED_ON_VIEW_AND_RUN, which means that the
// users will be able to see the machine 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
// machine.
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;
}

// Finalize the changes (commit them to the server).
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 machine.
    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 machine.
    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;
}

```

Working with Virtual Machine Templates

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

The Parallels Server API allows to perform the following template-related operations:

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

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

Obtaining a List of Templates

Both regular virtual machines and templates are obtained from the server using the same function: `PrlSrv_GetVmList` (see page 57). A template is identified by calling the `PrlVmCfg_IsTemplate` function which returns a boolean value indicating whether the specified virtual machine handle contains information about a regular virtual or a handle. The value of `PRL_TRUE` indicates that the machine is a template. The value of `PRL_FALSE` indicates that the machine is a regular virtual machine. The following sample is identical to the sample provided in the **Obtaining a List of Virtual Machines** section (see page 57) 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 machines.
    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(hJobResult);
        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 machines returned).
PRL_UINT32 nParamsCount = 0;
ret = PrlResult_GetParamsCount(hJobResult, &nParamsCount);
for (PRL_UINT32 i = 0; i < nParamsCount; ++i)
{
    PRL_HANDLE hVm; // virtual machine 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 machine 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 a Template From Scratch

The steps in creating a new template and the steps in creating a new virtual machine are exactly the same, with one exception: before registering a template on the server, 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 machine. The rest of the configuration parameters are set exactly as they are set for a regular virtual machine. See the **Creating a New Virtual Machine** section (on page 70) for the detailed information about creating a virtual machine.

The following example illustrates how to create a virtual machine 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 machines from it but you will not be able to run them until you configure them properly. Once again, the **Creating a New Virtual Machine** section (on page 70) provides all the necessary information and code samples needed to properly configure a virtual machine or a template.

```
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 machine 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 machine directory.
    // The bNonInteractiveMode parameter indicates not to
    // use interactive mode (the server will not send questions
    // to the client and will make all decisions on its own).
    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;
}
```

Converting a Regular Virtual Machine to a Template

Any virtual machine that is registered on a Parallels server can be converted to a template. This task is accomplished by modifying the virtual machine configuration. Only a single parameter must be modified: a flag indicating whether the machine is a regular virtual machine 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 machine to a template. Note that any of the virtual machine (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 machine 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 machine 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 a Template to a Regular Virtual Machine

Converting a template to a regular virtual machine is no different than converting a virtual machine 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 a New Virtual Machine From a Template.

The primary purpose of templates is to be used as patterns to create new virtual machines. New virtual machines are created from templates using the cloning functionality. We've already discussed how to clone a virtual machine in the [Cloning a Virtual Machine](#) section (see page 79). The truth is, creating a virtual machine from a template is at all different than creating a clone of a virtual machine. The `PrlVm_Clone` function that clones a virtual machine can also be used to create virtual machines from templates. The function has a boolean parameter that allows to specify whether a virtual machine or a template should be created. The following is almost the same example that we used in the [Cloning a Virtual Machine](#) section (see page 79) but this time we are setting the `bCreateTemplate` parameter to `PRL_TRUE`, thus creating a template instead of a regular virtual machine.

```
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 machine name.
    // Get the name of the original VM (template) and use
    // it in the new virtual machine 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 server.
    // Empty string indicates that the default
    // directory should be used.
    PRL_CHAR_PTR new_vm_root_path = "";

    // Virtual machine or template?
    // The cloning functionality allows to create
    // a new virtual machine or a new template.
    // True specifies to create a template.
    // False indicates to create a virtual machine.
    // We want to create a virtual machine 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;
}
```

CHAPTER 6

Events

In This Chapter

Receiving and Handling Events	103
Responding to Server Questions	106

Receiving and Handling Events

A Parallels server and all running virtual machines are constantly monitored for any changes in their state and status. When something important changes on the server side, an event of the corresponding type is triggered. A client 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 server configuration changes, the `PET_DSP_EVT_VM_STOPPED` event triggers when one of the virtual machines 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 machines to the client (see [Searching for Virtual Machines](#) (on page 73) for more info).

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 (see page 18). If you haven't read it yet, please do so now. To subscribe to event notifications, you must register your event handler with the server. 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 from the server. 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 server, a virtual machine, an I/O service, or a Web service. These are defined in the `PRL_EVENT_ISSUER_TYPE` enumeration.
- 4 If, in order to process 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 machine state change events (started, stopped, suspended, etc.) indicate that a virtual machine 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 [Parallels Server 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 (see page 18) 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 received.
    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 machine was started. \n");
            break;
        case PET_DSP_EVT_VM_STOPPED:
            // Handle the event here...
            printf("A virtual machine was stopped. \n");
            break;
        case PET_DSP_EVT_VM_CREATED:
            // Handle the event here...
            printf("A new virtual machine has been created. \n");
            break;
        case PET_DSP_EVT_VM_SUSPENDED:
            // Handle the event here...
            printf("A virtual machine has been suspended. \n");
            break;
        case PET_DSP_EVT_HW_CONFIG_CHANGED:
            // Handle the event here...
            printf("Parallels server configuration has been modified. \n");
            break;
        default:
            printf("Unhandled event: %d\n", eventType);
    }
}
```

Responding to Server 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 Parallels server may come to a situation that requires client input. For example, let's say that a client requested to create a new virtual machine but specified the hard drive size larger than the free disk space available on the hosts server. Since virtual hard drives in Parallels Server can dynamically allocate disk space, this is not necessarily a reason to abort the operation. In such a case, the server will pause the operation and will send a question to the client requiring one of the two possible answers: "Yes, create the machine 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 server. 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 server 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 server questions. In the example, we create a blank virtual machine 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 machine creation operation will not continue unless we send an answer to the server. 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_Init(PARALLELS_API_VER);
    if (PRL_FAILED(err))
    {
        // Error handling goes here...
        return -1;
    }

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

    // Login to Parallels server.
    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 machine.
PRL_HANDLE hVm;
PrlSrv_CreateVm(hServer, &hVm);
PrlVmCfg_SetName(hVm, "My simple VM");

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

// Register the event handler with the server.
// 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 server.
// 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_GetErrMsg(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, iterate 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_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;
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;
PrlEvent_GetServer(hEvent, &hServer);

// Send the handle containing the answer data to the server.
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;
    err = PrlVmCfg_CreateVmDev(
        hVm, // Target virtual machine.
        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;  
}
```

CHAPTER 7

Performance Statistics

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

- 1 Using `PrlSrv_GetStatistics` (for server statistics) or `PrlVm_GetStatistics` (for virtual machine statistics) to obtain a report containing the latest performance data.
- 2 Using `PrlSrv_SubscribeToHostStatistics` (for server statistics) or `PrlVm_SubscribeToGuestStatistics` (for virtual machine statistics) to receive statistics on a periodic basis.

The following sections describe each method in detail.

In This Chapter

Obtaining Performance Report	114
Performance Monitoring	117

Obtaining Performance Report

The first step required to access statistics report is to obtain a handle of type `PHT_SYSTEM_STATISTICS`. To do this, the following steps are necessary:

- 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 only be 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
<code>PHT_SYSTEM_STATISTICS</code>	Functions to drill deeper into specific system statistics. As an example, to use functions that return CPU statistics, a handle of type <code>PHT_SYSTEM_STATISTICS_CPU</code> will be required. This handle type can be created using function <code>PrlStat_GetCpuStat</code> . Functions that return memory statistics are also grouped here.
<code>PHT_SYSTEM_STATISTICS_CPU</code>	Functions that provide CPU statistics data.
<code>PHT_SYSTEM_STATISTICS_DISK</code>	Functions that provide hard disk statistics data.
<code>PHT_SYSTEM_STATISTICS_DISK_PARTITION</code>	Functions that provide statistics data for a disk partition.
<code>PHT_SYSTEM_STATISTICS_IFACE</code>	Functions that provide statistics data for a network interface.
<code>PHT_SYSTEM_STATISTICS_PROCESS</code>	Functions that provide statistics data about processes that are running.
<code>PHT_SYSTEM_STATISTICS_USER_SESSION</code>	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 the server 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 machine 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;
nServerStatistics = PrlJob_GetResult(hServerStatisticsJob, &hResult);
if (PRL_FAILED(nServerStatistics))
{
    printf("PrlJob_GetResult returned error: %s\n",
        prl_result_to_string(nServerStatistics));
    PrlHandle_Free(hResult);
    PrlHandle_Free(hServerStatisticsJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

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

PRL_HANDLE hCpuStatistics;
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;
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);
```

Performance Monitoring

To monitor a host server or a virtual machine 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 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 machine, instead of the host computer, by using `PrlVm_SubscribeToGuestStatistics` instead of `PrlSrv_SubscribeToHostStatistics`, and passing it a handle to a virtual machine that is running. This would also require using `PrlVm_UnsubscribeFromGuestStatistics` to stop receiving statistics data for the virtual machine.

```
#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");
            printf("-----\n");
        }

        PRL_HANDLE hEventParameters;
        PRL_HANDLE hServerStatistics;
        // 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;
        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;
        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_Init(PARALLELS_API_VER).
// 3. Create a PRL_SERVER handle using PrlSrv_Create.
// 4. Perform a server login using PrlSrv_Login.
// 5. Register our event handler (OurCallback function).
// 6. Subscribe to server (host) statistics events.
// 7. Keep receiving events until user presses <enter> key.
// 8. Unsubscribe from server (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_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 Parallels API.
    ret = PrlApi_Init(PARALLELS_API_VER);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlApi_Init 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, "PrlSrv_Create failed, error: %s",
            prl_result_to_string(ret));
        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

    // Perform a server login (PrlSrv is asynchronous).
    PRL_HANDLE hJob = PrlSrv_Login(

```

```

        hServer,          // PRL_HANDLE of type PHT_SERVER.
        szServer,        // Server hostname 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 server (host) statistics events.
// 3. Keep receiving events until user presses <enter> key.
// 4. Unsubscribe from server (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);

// -----

// Logoff the Parallels server.
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 Parallels API, and unload the SDK.
PrlApi_Deinit();
SdkWrap_Unload();

return 0;
}
```

Index

A

Adding an Existing Virtual Machine • 77
 API Basics • 12
 Asynchronous Functions • 18

C

Cloning a Virtual Machine • 79
 Common Network Requirements • 8
 Compiling Client Applications • 9
 Converting a Regular Virtual Machine to a Template • 98
 Converting a Template to a Regular Virtual Machine • 99
 Creating a New Virtual Machine • 70
 Creating a New Virtual Machine From a Template. • 100
 Creating a Template From Scratch • 96

D

Deleting a Virtual Machine • 81
 Determining Virtual Machine State • 62

E

Error Codes • 24
 Events • 102

G

Getting Started • 6

H

Handles • 13
 Hard Disks • 87
 Host Server Operations • 33
 Host-only and Shared Networking • 90

I

Installation • 8
 Installing Parallels Server SDK on Linux • 9
 Installing Parallels Server SDK on Mac OS X • 8
 Installing Parallels Server SDK on Windows • 8

L

Linux • 11

Linux Clients • 7

M

Mac OS X • 9
 Mac OS X Clients • 7
 Managing Files on Host Server • 48
 Managing Parallels Server Preferences • 37
 Managing Parallels Server Users • 43
 Managing User Access Rights • 91
 Modifying Virtual Machine Configuration • 82

N

Name, Description, Boot Options • 85
 Network Adapters • 89

O

Obtaining a List of Templates • 94
 Obtaining a List of Virtual Machines • 57
 Obtaining a PHT_VM_CONFIGURATION handle • 84
 Obtaining a Problem Report • 54
 Obtaining Performance Report • 114
 Obtaining Server Handle and Logging In • 28
 Obtaining Virtual Machine Configuration Data • 60
 Overview • 6

P

Parallels Server License • 51
 Performance Monitoring • 117
 Performance Statistics • 113
 PrlVm_BeginEdit and PrlVm_Commit Functions • 83

R

RAM Size • 86
 Receiving and Handling Events • 103
 Responding to Server Questions • 106
 Retrieving Host Server Configuration Information • 34

S

Searching for Parallels Servers • 40
 Searching for Virtual Machines • 73
 Starting, Stopping, Restarting a Virtual Machine • 64

Strings as Return Values • 23
Suspending and Pausing a Virtual Machine •
68
System Requirements • 7

V

Virtual Machine Operations • 56

W

Windows • 10
Windows Clients • 7
Working with Results • 15
Working with Virtual Machine Templates • 93