

Parallels Virtualization SDK

Programmer's Guide

*Parallels Holdings, Ltd.
c/o Parallels International GmbH.
Parallels International GmbH
Vordergasse 49
CH8200 Schaffhausen
Switzerland
Tel: + 41 526320 411
Fax: + 41 52672 2010
www.parallels.com*

Copyright © 1999-2011 Parallels Holdings, Ltd. and its affiliates. All rights reserved.

This product is protected by United States and international copyright laws. The product's underlying technology, patents, and trademarks are listed at <http://www.parallels.com/trademarks>.

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.

Mac is a registered trademark of Apple, Inc.

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

Contents

Getting Started	5
Overview	5
System Requirements	6
Mac OS X Clients.....	6
Windows Clients	6
Linux Clients	7
Common Network Requirements	7
Parallels C API Concepts	8
Compiling Client Applications	8
Mac OS X	8
Windows.....	18
Linux	19
Handles	20
Synchronous Functions.....	21
Asynchronous Functions.....	22
Strings as Return Values.....	26
Error Handling	28
Parallels C API by Example	30
Obtaining Server Handle and Logging In	31
Host Operations	35
Retrieving Host Configuration Information	36
Managing Parallels Service Preferences.....	39
Searching for Parallels Servers	42
Managing Parallels Service Users	45
Managing Files In The Host OS	50
Managing Licenses.....	53
Obtaining a Problem Report.....	56
Virtual Machine Operations.....	58
Obtaining the Virtual Machines List	59
Searching for Virtual Machine by Name.....	62
Obtaining Virtual Machine Configuration Information.....	64
Determining Virtual Machine State.....	66
Starting, Stopping, Resetting a Virtual Machine	69
Suspending and Pausing a Virtual Machine	70
Creating a New Virtual Machine.....	72
Searching for Virtual Machines.....	75
Adding an Existing Virtual Machine	79
Cloning a Virtual Machine	82
Deleting a Virtual Machine	84
Modifying Virtual Machine Configuration.....	85
Managing User Access Rights.....	99
Working with Virtual Machine Templates	101
Events	110
Receiving and Handling Events.....	111
Responding to Parallels Service Questions	114
Performance Statistics.....	121
Obtaining Performance Report	122
Performance Monitoring.....	125

Encryption Plug-in..... 130

- Encryption Plug-in Basics 130
- The Encryption API Reference..... 131
- Implementing a Plug-in 134
- Building the Dynamic Library..... 140
- Plug-in Installation and Usage..... 141

Parallels Python API Concepts 142

Package and Modules 143

Classes 144

Class Methods..... 144

- Synchronous Methods 144
- Asynchronous Methods 145

Error Handling 147

Parallels Python API by Example 148

Creating a Basic Application 149

Connecting to Parallels Service and Logging In..... 152

Host Operations 155

- Retrieving Host Configuration Info..... 155
- Managing Parallels Service Preferences..... 157

Virtual Machine Operations..... 158

- Obtaining the Virtual Machine List..... 159
- Searching for a Virtual Machine..... 161
- Performing Power Operations 162
- Creating a New Virtual Machine..... 163
- Obtaining Virtual Machine Configuration Data 165
- Modifying Virtual Machine Configuration..... 168
- Adding an Existing Virtual Machine..... 175
- Removing an Existing Virtual Machine 176
- Cloning a Virtual Machine 177

Remote Desktop Access 178

- Creating a Simple OS Installation Program..... 179

Index 183

Getting Started

In This Chapter

Overview	5
System Requirements	6

Overview

Parallels Virtualization SDK is a development kit used to create and integrate custom software solutions with Parallels virtualization products. The SDK provides cross-platform ANSI C and Python APIs. The SDK can be used to develop software for any hypervisor-based Parallels virtualization product such as Parallels Server, Parallels Workstation, and Parallels Desktop.

The SDK comprises the following components:

- C header files.
- Dynamic libraries.
- Python package for developing client applications in Python.
- Parallels command line tools (`prlctl`, `prlsrvctl`) -- a command line utility that can be used to perform a full range of host and virtual machine operations.
- Parallels Virtualization SDK Programmer's Guide (this document).
- Parallels C API Reference Guide.
- Parallels Python API Reference Guide.
- Parallels Command Line Reference Guide.

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.
- Parallels Python API requires Python 2.5. Other versions of Python are not officially supported.

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.
- The Parallels Python API requires Python 2.5. Other versions of Python are not officially supported.

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

Parallels Server

When creating client applications for Parallels Server, 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 server via TCP/IP. The server is listening on port 6400. Please make sure that the port is not blocked by a firewall.

Parallels Desktop and Parallels Workstation

Remote connections to the Parallels Service are not allowed with Parallels Desktop or Parallels Workstation. With these products, you can run your client applications on the host computer only.

Parallels C API Concepts

In This Chapter

Compiling Client Applications	8
Handles.....	20
Synchronous Functions	21
Asynchronous Functions.....	22
Strings as Return Values	26
Error Handling	28

Compiling Client Applications

Mac OS X

Parallels Virtualization SDK for Mac OS X is provided as a framework. The framework is installed in the following directory:

```
/Library/Frameworks/ParallelsVirtualizationSDK.framework
```

You can use the framework just like any other Apple framework when creating development projects and compiling applications. Alternately, you can compile and build your applications without using the framework. In such a case, you will have to specify all the necessary paths to the SDK source files manually.

When using the framework, the dynamic library, which is supplied with the SDK, will be directly linked to the application. If you would like to load the dynamic library at runtime, the Parallels Virtualization SDK includes a convenient `dlopen` wrapper for this purpose called *SdkWrap*. Using the wrapper, you can load and unload the library symbols at any time with one simple call. Please note that in order to use *SdkWrap*, you must compile your application without using the framework. The wrapper source files are located in the `Helpers/SdkWrap` directory, which is located in the main SDK installation directory.

The following subsections describe various compilation scenarios in detail and provide code samples.

Compiling with SdkWrap

When using SdkWrap, your program must contain the following:

- The `#include "SdkWrap.h"` directive. This header file defines the wrapper functions.
- The `#define SDK_LIB_NAME "libprl_sdk.dylib"` directive. This is the name of the dynamic library included in the SDK.
- The `SdkWrap_Load(SDK_LIB_NAME)` function call that will load the dynamic library symbols.
- The `SdkWrap_Unload()` function call that will unload the dynamic library when it is no longer needed.

To compile a program, the following compiler options and instructions must be used:

- The `DYN_API_WRAP` preprocessor macro must be defined.
- Full paths to the `Headers` and the `Helpers/SdkWrap` directories must be specified. Both directories are located in the main SDK installation directory.
- The `SdkWrap.cpp` file must be included in the project and must be built together with the main target.
- The `libdl` library must be linked to the application. This is the standard dynamic linking interface library needed to load the SDK library.

Using Makefile

The following is a sample Makefile that demonstrates the implementation of the requirements described above. To compile a program and to build an executable, type `make` in the Terminal window. To clean up the project, type `make clean`. Please note that the `SOURCE` variable must contain the name of your source file name.

```
# Source file name.
# Substitute the file name with your own.
SOURCE = HelloWorld

# Target executable file name.
# Here we are using the same name as the source file name.
TARGET = $(SOURCE)

# Path to the Parallels Virtualization SDK files.
SDK_PATH = /Library/Frameworks/ParallelsVirtualizationSDK.framework

# Relative path to the SdkWrap directory containing
# the SDK helper files. The files are used to load
# the dynamic library.
SDK_WRAP_PATH = Helpers/SdkWrap

OBJS = SdkWrap.o $(SOURCE).o
CXX = g++
CXXFLAGS = -DDYN_API_WRAP -I$(SDK_PATH)/Headers -I$(SDK_PATH)/$(SDK_WRAP_PATH)
LDFLAGS = -ldl

all : $(TARGET)

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

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

```

SdkWrap.o : $(SDK_PATH)/$(SDK_WRAP_PATH)/SdkWrap.cpp
             $(CXX) -c -o $@ $(CXXFLAGS) $(SDK_PATH)/$(SDK_WRAP_PATH)/SdkWrap.cpp

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

.PHONY : all clean

```

Using Xcode IDE

If you are using the Xcode IDE, follow these steps to set up your project:

- 1 Add the `SdkWrap.h` and the `SdkWrap.cpp` files to your project.
- 2 In the Search Paths collection, specify:
 - a full path to the `Helpers/SdkWrap` directory (contains the wrapper source files)
 - a full path to the `Headers` directory (contains the SDK header files)
 - a full path to the `Libraries` directory (contains the dynamic library)
- 3 In the Preprocessor collection, add the `DYN_API_WRAP` preprocessor macro.

Example

The following is a complete sample program that demonstrates the usage of the `SdkWrap` wrapper. The program loads the dynamic library, initializes the API, and then logs in to the local Parallels Service. You can copy the entire program into a file on your Mac and try building and then running it. The program uses a cross-platform approach, so it can also be compiled on Windows and Linux machines.

```

#include "SdkWrap.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

PRL_RESULT LoginLocal(PRL_HANDLE &hServer);
PRL_RESULT LogOff(PRL_HANDLE &hServer);

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

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 err = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Log in to Parallels Service.
    err = LoginLocal(hServer);

    // Log off.
    err = LogOff(hServer);

```

```

printf( "\nEnd of program.\n\n" );
printf("Press Enter to exit...");
getchar();

exit(0);
}

// Initializes the SDK library and
// logs in to the local Parallels Service.
//
PRL_RESULT LoginLocal(PRL_HANDLE &hServer)
{
    // Variables for handles.
    PRL_HANDLE hJob = PRL_INVALID_HANDLE; // job handle
    PRL_HANDLE hJobResult = PRL_INVALID_HANDLE; // job result

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

    // Use the correct dynamic library depending on the platform.
    #ifdef _WIN_
    #define SDK_LIB_NAME "prl_sdk.dll"
    #elif defined(_LIN_)
    #define SDK_LIB_NAME "libprl_sdk.so"
    #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. In this example, we are initializing the
    // API for Parallels Desktop.
    // To initialize in the Parallels Workstation mode, pass PAM_WORKSTATION
    // as the second parameter.
    // To initialize for Parallels Server, pass PAM_SERVER.
    // See the PRL_APPLICATION_MODE enumeration for all possible options.
    err = PrlApi_InitEx(PARALLELS_API_VER, PAM_DESKTOP, 0, 0);

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

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

    // Log in (asynchronous call).

```

```

hJob = PrlSrv_LoginLocal(hServer, NULL, NULL, PSL_NORMAL_SECURITY);

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

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

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

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

return 0;
}

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

    PRL_RESULT err = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

    // Log off.
    hJob = PrlSrv_Logoff(hServer);
    err = PrlJob_Wait(hJob, 1000);

```

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

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

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

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

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

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

return 0;
}
```

Compiling with Framework

If you are using the ParallelsVirtualizationSDK framework, the program must contain the following include directive:

```
#include "ParallelsVirtualizationSDK/Parallels.h"
```

Parallels.h is the main SDK header file. Please note the framework name in front of the SDK header file name. This is a common requirement when using a framework.

Note: The difference between the SdkWrap scenario (described in the previous subsection) and the framework scenario is that `Parallels.h` must be included when using the framework, while `SdkWrap.h` must be included when using `SdkWrap`. The two files must never be included together. Please also note that you don't have to load the dynamic library manually in your program when using the framework.

The only compiler option that must be specified when using the framework is:

```
-framework ParallelsVirtualizationSDK
```

Using Makefile

The following sample Makefile can be used to compile a program using the ParallelsVirtualizationSDK framework:

```
# Source file name.
# Substitute the file name with your own.
SOURCE = HelloWorld

# Target executable file name.
# Here we are using the same name as the source file name.
TARGET = $(SOURCE)

CXX = g++
LDFLAGS = -framework ParallelsVirtualizationSDK

all : $(TARGET)

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

$(SOURCE).o : $(SOURCE).cpp
    $(CXX) -c -o $@ $(SOURCE).cpp

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

.PHONY : all clean
```

Using Xcode IDE

When setting up an Xcode project, the only thing that you have to do is add the ParallelsVirtualizationSDK framework to the project. No other project modifications are necessary.

Sample

The following is a complete sample program that demonstrates the usage of the ParallelsVirtualizationSDK framework.

```
#include "ParallelsVirtualizationSDK/Parallels.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

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

PRL_RESULT LoginLocal(PRL_HANDLE &hServer);
PRL_RESULT LogOff(PRL_HANDLE &hServer);

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

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

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

    // Log in.
    err = LoginLocal(hServer);

    // Log off
    err = LogOff(hServer);

    printf( "\nEnd of program.\n\n" );
    printf("Press Enter to exit...");
    getchar();

    exit(0);
}

// Intializes the SDK library and
// logs in to the local Parallels Service.
//
PRL_RESULT LoginLocal(PRL_HANDLE &hServer)
{
    // Variables for handles.
    PRL_HANDLE hJob = PRL_INVALID_HANDLE; // job handle

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

    // Initialize the API. In this example, we are initializing the
    // API for Parallels Workstation.
    // To initialize in the Parallels Desktop mode, pass PAM_DESKTOP
    // as the second parameter.
    // To initialize for Parallels Server, pass PAM_SERVER.
    // See the PRL_APPLICATION_MODE enumeration for all possible options.
    err = PrlApi_InitEx(PARALLELS_API_VER, PAM_DESKTOP, 0, 0);

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

```

    return -1;
}

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

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

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

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

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

return 0;
}

// Log off the Parallels Service and
// deinitializes the SDK library.
//

```

```
PRL_RESULT LogOff(PRL_HANDLE &hServer)
{
    PRL_HANDLE hJob = PRL_INVALID_HANDLE;

    PRL_RESULT err = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

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

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

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

    return 0;
}
```

Windows

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

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

```
SdkWrap.h
```

```
SdkWrap.cpp
```

These are the helper files that provide a set of methods for loading and unloading dynamic libraries. You can use the included source code file to customize this functionality if you wish.

- 4** Add the following `#include` directive to your program:

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

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

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

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

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

Linux

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

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

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

# Set the current folder name
TARGET = Example

all : $(TARGET)

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

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

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

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

.PHONY : all clean
```

Handles

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

Handle Types

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

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

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

Obtaining a Handle

A handle is usually obtained by calling a function that operates on another (we may call it "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 C API Reference* guide contains a description of every available handle and explains how each particular handle type can be obtained. The examples in this guide also demonstrate how to obtain handles of different types.

Freeing a Handle

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

Multithreading

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

Example

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

```
PRL_HANDLE hServer = PRL_INVALID_HANDLE;
PRL_RESULT ret;

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

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

// Free the handle when it is no longer needed.
PrlHandle_Free(hServer);
```

Synchronous Functions

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

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

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

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

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

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

Asynchronous Functions

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

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

The Callback Function (Event Handler)

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

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

The following is an example of the callback function implementation:

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

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

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

To handle system events within a callback function:

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

To handle jobs within a callback function:

- 1 Get the job type using `PrlJob_GetType`. A job type can be used to identify the function that started the job and to determine the type of the result it contains. For example, a job of type `PJOC_SRV_GET_VM_LIST` is started by `PrlSrv_GetVmList` function call, which returns a list of virtual 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. In such cases, use `PrlResult_GetParamCount` and `PrlResult_GetParamByIndex`.
- 6 Implement code to use the handle obtained in step 5.

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

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

```
static PRL_RESULT OurCallbackFunction(PRL_HANDLE hHandle, PRL_VOID_PTR
pUserData)
{
    PRL_JOB_OPERATION_CODE nJobType = PJOC_UNKNOWN; // job type
    PRL_HANDLE_TYPE nHandleType = PHT_ERROR; // handle type
    PRL_HANDLE hVm = PRL_INVALID_HANDLE; // virtual 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 = PRL_INVALID_HANDLE;
            PrlEvent_GetParam(hHandle, 0, &hEventParameters);

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

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

```

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

    // Check if the job type is PJOC_SRV_GET_VM_LIST.
    if (nJobType == PJOC_SRV_GET_VM_LIST)
    {
        // Check the job return code.
        PRL_RESULT nJobRetCode;
        PrlJob_GetRetCode(hHandle, &nJobRetCode);
        if (PRL_FAILED(nJobRetCode))
        {
            fprintf(stderr, "[B]%.8X: %s\n", nJobRetCode,
                prl_result_to_string(nJobRetCode));
            PrlHandle_Free(hHandle);
            return nJobRetCode;
        }

        err = PrlJob_GetResult(hHandle, &hJobResult);

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

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

        // Iterate through the parameter list.
        for(nParamIndex = 0; nParamIndex < nParamsCount ; nParamIndex++)
        {
            // Obtain a virtual 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 of the events/jobs regardless of their origin. It is the responsibility of the client application to identify the type of the event and to handle each one accordingly.

```

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

// Make a call to an asynchronous function here.
// OurCallbackFunction will be called by the background thread

```

```

// as soon as the job is completed, and code within
// OurCallbackFunction can populate the ReturnDataClass instance.
// For example, we can make the following call here:

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

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

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

```

Calling Asynchronous Functions Synchronously

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

```

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

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

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

```

Strings as Return Values

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

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

Consider the following function:

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

The `PrlVmCfg_GetName` function above is a typical Parallels 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 Handling

Synchronous Functions

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

Error Codes for Asynchronous Functions

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

Analyzing Return Values

Parallels C API provides the following 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 the host and analyzes the return value (error code) of asynchronous function `PrlSrv_CreateDir`.

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

// Wait for a maximum of 5 seconds for asynchronous
// function PrlSrv_FsCreateDir to complete.
PRL_RESULT resWaitForCreateDir = PrlJob_Wait(hJob, 5000);
if (PRL_FAILED(resWaitForCreateDir))
{
    fprintf(stderr, "PrlJob_Wait for 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 is that sometimes errors contain dynamic parameters. The following example demonstrates how to obtain descriptive error information:

```
PrlJob_GetRetCode(hJob, &nJobResult);

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

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

Parallels C API by Example

In This Chapter

Obtaining Server Handle and Logging In	31
Host Operations.....	35
Virtual Machine Operations.....	58
Events.....	110
Performance Statistics.....	121
Encryption Plug-in	130

Obtaining Server Handle and Logging In

The following steps are required in any client application using the Parallels C API:

- 1 Load the Parallels Virtualization SDK library using the `SdkWrap_Load` function.
- 2 Initialize the API using the `PrlApi_InitEx` function. The API must be initialized properly for the given Parallels product, such as Parallels Server, Parallels Workstation, Parallels Desktop, etc. The initialization mode is determined by the value of the `nAppMode` parameter passed to the `PrlApi_InitEx` function. The value must be one of the enumerators from the `PRL_APPLICATION_MODE` enumeration.
- 3 Create a server handle using the `PrlSrv_Create` function.
- 4 Call `PrlSrv_LoginLocal` or `PrlSrv_Login` to login to the *Parallels Virtualization Service* (or simply Parallels Service). Parallels Service is a combination of processes running on the host machine that comprise the Parallels virtualization product. The first function is used when the client program and the target Parallels Service are running on the same host. The second function (`PrlSrv_Login`) is used to log in to a remote Parallels Service. Please note that remote login is supported in Parallels Server-based virtualization products only.

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

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

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

Example

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

```
// Intializes the SDK library and
// logs in to the local Parallels Service.
// Obtains a handle of type PHT_SERVER identifying
// the Parallels Service.
PRL_RESULT LoginLocal(PRL_HANDLE &hServer)
```

```

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

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

    // Use the correct dynamic library depending on the platform.
    #ifdef _WIN_
    #define SDK_LIB_NAME "prl_sdk.dll"
    #elif defined(_LIN_)
    #define SDK_LIB_NAME "libprl_sdk.so"
    #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. In this example, we are initializing the
    // API for Parallels Workstation.
    // To initialize in the Parallels Desktop mode, pass PAM_DESKTOP
    // as the second parameter.
    // To initialize for Parallels Server, pass PAM_SERVER.
    // See the PRL_APPLICATION_MODE enumeration for all possible options.
    //
    err = PrlApi_InitEx(PARALLELS_API_VER, PAM_WORKSTATION, 0, 0);

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

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

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

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

```

```

        PrlHandle_Free(hServer);
        PrlApi_Deinit();
        SdkWrap_Unload();
        return -1;
    }

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

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

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

    return 0;
}

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

    PRL_RESULT err = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

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

```

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

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

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

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

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

return 0;
}
```

Host Operations

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

In This Chapter

Retrieving Host Configuration Information.....	36
Managing Parallels Service Preferences	39
Searching for Parallels Servers	42
Managing Parallels Service Users.....	45
Managing Files In The Host OS.....	50
Managing Licenses	53
Obtaining a Problem Report.....	56

Retrieving Host Configuration Information

The Parallels C API provides a set of functions to retrieve detailed information about a host machine. 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 Service preferences, setting up devices inside virtual machines, or whenever you need to obtain information about the resources 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](#) (p. 31).

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

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

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

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

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

    // Get job result.
    ret = PrlJob_GetResult(hJob, &hJobResult);
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    // Get adapter type (physical, virtual).
```

```
PRL_HW_INFO_NET_ADAPTER_TYPE nNetAdapterType;
ret = PrlSrvCfgNet_GetNetAdapterType(phDevice,
                                     &nNetAdapterType);
printf("Type: %d\n", nNetAdapterType);

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

PrlHandle_Free(hHostConfig);

return 0;
}
```

Managing Parallels Service Preferences

Parallels Service preferences is a set of parameters that control its default behaviour. Some of the important parameters are:

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

Parallels Service preferences are managed 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 C 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 Service 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](#) (p. 31).

```
PRL_RESULT GetSetServicePrefs(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;
}

printf("Parallels Service Preferences \n\n");
printf("Default VM Directory: %s\n", sDefaultDir);

// Get the recommended virtual machine memory size.
PRL_UINT32 nMemSize = 0;
ret = PrlDispCfg_GetRecommendMaxVmMem(hDispConfig, &nMemSize);
if (PRL_FAILED(ret))
{
    fprintf(stderr, "Error: %s\n",
            prl_result_to_string(ret));
    PrlHandle_Free(hDispConfig);
    return -1;
}

printf("Recommended VM memory size: %d\n", nMemSize);

// Modify some of the Parallels Service 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.
    nMemSize = 512;
    ret = PrlDispCfg_SetRecommendMaxVmMem(hDispConfig, nMemSize);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "Error: %s\n",
            prl_result_to_string(ret));
        PrlHandle_Free(hDispConfig);
        return -1;
    }

    // Commit the changes.
    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;
    }

    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;
    }

    printf("The recommended VM memory size changed to: %d\n", nMemSize);

    PrlHandle_Free(hDispConfig);

    return 0;
}

```

Searching for Parallels Servers

This topic applies to Parallels Server only.

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 Parallels Server. The information includes host name, port number, version of the OS that a host is running, Parallels Server version number, and the global ID (UUID). This information can then be used to establish a connection with the Parallels 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 first. 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 Parallels Server found and a handle of type `PHT_SERVER_INFO` containing the Parallels Server information will be passed to it. Searching an entire local area network can take a significant time, so using a callback 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.

Note: The `PrlSrv_LookupParallelsServers` function can be executed without being logged in to a Parallels Service. For example, if you are writing an application with a user interface, you can search the network for available Parallels Servers and present the list to the user so that he/she can select a server to connect to.

The following sample functions demonstrate how to search local network for Parallels Servers. The first sample function calls the `PrlSrv_LookupParallelsServers` function synchronously. The second function takes an asynchronous approach.

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

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

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

    if (PRL_SUCCEEDED(ret))
    {
        printf("Found Parallels 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 (we'll call it `ourCallback`):

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

    // Get the host name.
    PRL_UINT32 nBufSize = 1024;
    PRL_CHAR sBuf[nBufSize];
    PrlSrvInfo_GetHostName(hEvent, sBuf, &nBufSize);

    // Get the other server properties and process them here, if needed...

    // 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 Service Users

This topic applies to Parallels Server only.

Parallels Service 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 Service 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 user as soon as he/she logs in to a Parallels Service for the very first time.

There are two API handles that are used to obtain information about Parallels Service users and to modify some of the user 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 is used to obtain information about currently logged in user while PHT_USER_INFO is used to obtain information about a specified user. There are also some differences in the type of the information provided.

Getting the information about the currently logged in user

The information about the currently logged in user can be retrieved using functions of the PHT_USER_PROFILE handle. The following sample demonstrates how to obtain the handle and how to use its functions to retrieve user information. The sample also shows how to set up a default virtual machine directory for the user. Parallels Service automatically assigns a default virtual machine directory (the directory where new virtual machines are created) for every new user. If needed, a user can specify a different directory for his/her virtual machines. At the time of this writing, this is the only property of the Parallels Service user profile that can be modified. Every user profile modification 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 Service 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 Service 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 Service 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 Parallels Service.
    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 In The Host OS

The following file management operations can be performed using the Parallels C API on the host machine:

- 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 "PrISrv_Fs".

Obtaining the host OS directory listing

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

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

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

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

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

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

```

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

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

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

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

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

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

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

```

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

return PRL_ERR_SUCCESS;
}
```

Managing Licenses

The Parallels 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, use the `PrlSrv_UpdateLicense` function.

The following sample function demonstrates how to obtain license information and how to install a new license.

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

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

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobReturnCode = PRL_ERR_UNINITIALIZED;

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

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

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

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

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

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

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

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

return 0;
}
```

Virtual Machine Operations

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

In This Chapter

Obtaining the Virtual Machines List.....	59
Searching for Virtual Machine by Name	62
Obtaining Virtual Machine Configuration Information	64
Determining Virtual Machine State	66
Starting, Stopping, Resetting a Virtual Machine.....	69
Suspending and Pausing a Virtual Machine.....	70
Creating a New Virtual Machine	72
Searching for Virtual Machines	75
Adding an Existing Virtual Machine	79
Cloning a Virtual Machine.....	82
Deleting a Virtual Machine.....	84
Modifying Virtual Machine Configuration.....	85
Managing User Access Rights	99
Working with Virtual Machine Templates.....	101

Obtaining the Virtual Machines List

Any virtual machine operation begins with obtaining a handle of type `PHT_VIRTUAL_MACHINE` identifying the virtual machine. Once a handle identifying a virtual machine is obtained, its functions can be used to perform a full range of virtual machine operations. This section describes how to obtain a list of handles, each identifying an individual virtual machine registered with a Parallels Service. If you would like to search for *unregistered* virtual machines on the host computer, please refer to the [Searching for Virtual Machines](#) section (p. 75).

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

- 1 Log in to the Parallels Service and obtain a handle of type `PHT_SERVER`. See [Obtaining Server Handle and Logging In](#) (p. 31) for more info and code samples.
- 2 Call `PrlSrv_GetVmList`. This is an asynchronous function that returns a handle of type `PHT_JOB`.
- 3 Call `PrlJob_GetResults` passing the `PHT_JOB` object obtained in step 2. This function returns a handle of type `PHT_RESULT` containing the virtual machine list.
- 4 Free the job handle using `PrlSrv_GetVmList` as it is no longer needed.
- 5 Call `PrlResult_GetParamsCount` to determine the number of virtual machines contained in the `PHT_RESULT` object.
- 6 Call the `PrlResult_GetParamByIndex` function in a loop passing an index from 0 to the total virtual machine count. The function obtains a handle of type `PHT_VIRTUAL_MACHINE` containing information about an individual virtual machine.
- 7 Use functions of the `PHT_VIRTUAL_MACHINE` object to obtain the virtual machine information. For example, use `PrlVmCfg_GetName` to obtain the virtual machine name.
- 8 Free the virtual machine handle using `PrlHandle_Free`.
- 9 Free the result handle using `PrlHandle_Free`.

The following sample function implements the steps described above.

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

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

    // Get the list of the available virtual 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(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);

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

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

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

    // Now that we have a handle of type PHT_VIRTUAL_MACHINE,
    // we can use its functions to retrieve or to modify the
    // virtual 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",
            prl_result_to_string(ret));
    }
    else
    {
        printf(" (%d) %s\n\n", i+1, szVmNameReturned);
    }
}

```

```
        // Free the virtual machine handle.  
        PrlHandle_Free(hVm);  
    }  
  
    return PRL_ERR_SUCCESS;  
}
```

Searching for Virtual Machine by Name

This section contains an example of how to obtain a handle of type `PHT_VIRTUAL_MACHINE` identifying the virtual machine using the virtual machine name as a search parameter. We will use the sample as a helper function in the later section of this guide that demonstrate how to perform operations on virtual machines. The sample is based on the code provided in the [Obtaining the Virtual Machine List](#) section (p. 59).

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

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

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

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

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

    PrlHandle_Free(hJob);

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

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

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

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

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

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

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

return PRL_ERR_NO_DATA;
}
```

Obtaining Virtual Machine Configuration Information

The virtual machine configuration information is obtained using functions of the PHT_VM_CONFIGURATION object. The functions are prefixed with `PrlVmCfg_`. To use the functions, a handle of type `PHT_VM_CONFIGURATION` must first be obtained from the virtual machine object (a handle of type `PHT_VIRTUAL_MACHINE`) using the `PrlVm_GetConfig` function. The following example shows how to obtain the virtual machine name, guest operating system type and version, RAM size, HDD size, and CPU count. To obtain the virtual machine handle (`hVm` input parameter), use the helper function described in the [Searching for Virtual Machine by Name](#) section (p. 62).

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

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

    ret = PrlVm_GetConfig(hVm, &hVmCfg);

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

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

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

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

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

    free(szVmName);

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

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

```
        break;
    default:
        printf("OS Type: %s: %d\n", "Other OS Type: ", nOsType);
    }

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

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

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

    // Get CPU count.
    PRL_UINT32 nCpuCount = 0;
    PrlVmCfg_GetCpuCount(hVmCfg, &nCpuCount);
    printf("Number of CPUs: %d\n", nCpuCount);

    return PRL_ERR_SUCCESS;
}
```

Determining Virtual 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 (p. 59). 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 that defines every possible state and transition applicable to a virtual machine. The following table lists the available states and transitions:

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

The following example demonstrates how obtain state/transition 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;
}
printf("Status: ");

switch (vm_state) {
    case VMS_UNKNOWN:
        printf("Unknown state\n");
}
```

```
        break;
    case VMS_STOPPED:
        printf("Stopped\n");
        break;
    case VMS_STARTING:
        printf("Starting...\n");
        break;
    case VMS_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_SUSPENDEED:
        printf("Suspended\n");
        break;
    default:
        printf("Unknown state\n");
    }
    printf("\n");

    PrlHandle_Free(hVmCfg);
    PrlHandle_Free(hVmInfo);

    return 0;
}
```

Starting, Stopping, Resetting a Virtual Machine

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

Stopping a virtual machine is not the same as performing a guest operating system shutdown operation. When a virtual machine is stopped, it is a *cold stop* (i.e. it is the same as turning off the power to a physical computer). Any unsaved data will be lost. However, if the OS in the virtual 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.

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

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

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

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

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

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

    return PRL_ERR_SUCCESS;
}
```

Suspending and Pausing a Virtual Machine

Suspending a Virtual Machine

When a virtual machine is suspended, the information about its state 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 called *Windows XP - 01*.

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

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, 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);
```

with:

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

Dropping Suspended State

A suspended virtual machine 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 the Parallels Service. 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 registered, it can be powered on and an operating system can be installed on it.

In this section, we will discuss how to create a typical virtual machine for a particular OS type using a sample configuration. By using this approach, you can easily create a virtual machine without knowing all of the little 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 new handle of type `PHT_VIRTUAL_MACHINE` using the `PrlSrv_CreateVm` function. The handle will identify our new virtual machine.
- 2** Obtain a handle of type `PHT_VM_CONFIGURATION` by calling the `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 in the virtual 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 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. The devices include floppy disk drive, CD drive, serial and parallel ports, sound card, etc. To connect the available host devices, obtain a handle of type `PHT_SERVER_CONFIG` (physical host configuration) using the `PrlSrv_GetSrvConfig` function. The handle should then be passed to `PrlVmCfg_SetDefaultConfig` together with OS information and other parameters. If you don't want to connect the devices, set the `hSrvConfig` parameter to `PRL_INVALID_HANDLE`.
- 4** Choose a name for the new virtual 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 using the `PrlVm_Reg` function. This step will create the necessary virtual machine files on the host and register the machine with the Parallels Service. The directory containing the virtual machine files will have the same name as the virtual machine name. The directory will be created in the default location for this Parallels Service. 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 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 hosts configuration handle from it.
ret = PrlJob_GetResult(hJob, &hResult);
PRL_HANDLE hSrvCfg = PRL_INVALID_HANDLE;
PrlResult_GetParam(hResult, &hSrvCfg);

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

// Now that we have the host configuration data,
// we can set the default configuration for the new virtual 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 = PRL_INVALID_HANDLE;
ret = PrlVmCfg_GetHardDisk(hVmCfg, 0, &hHDD);
ret = PrlVmDevHd_SetDiskSize(hHDD, 20000);

// Create and register the machine with the Parallels Service.
// 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 host computer may have virtual machines on its hard drive that are not currently registered with the Parallels Service. This can happen when a virtual machine is removed from the Parallels Service 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 C 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, or remove it from the hard drive.

Since the search operation may take a long time (depending on the size of the specified directory tree), the `PrlSrv_StartSearchVms` function should be executed using the callback functionality (p. 22). 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 (p. 22), a callback function can receive two types of objects: *jobs* (`PHT_JOB`) and *events* (`PHT_EVENT`). In this instance, the information is passed to the callback function as an event of type `PET_DSP_EVT_FOUND_LOST_VM_CONFIG`. To following steps are involved in processing the event inside the callback function:

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

The following is an implementation of the steps above:

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

    PrlHandle_GetType(hEvent, &nHandleType);

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

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

```

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

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

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

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

    // Get the virtual 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 unregistered virtual machines in the specified directory tree, then the function will never be called as a PET_DSP_EVT_FOUND_LOST_VM_CONFIG event (it will still be called at least once as a result of the started job and will receive the job object but this and possibly other callback invocations are irrelevant in the context of this example).

Receiving the search results synchronously

It is also possible to use this function synchronously using the `PrlJob_Wait` function (p. 22). 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

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

Note: The `PrlSrv_RegisterVm` function does NOT generate new MAC addresses for the virtual network adapters that already exist in the virtual machine. If the machine is a copy of another virtual machine then you should set new MAC addresses for its network adapters after you register it. The example at the end of this section demonstrates how this can be accomplished. For more information on modifying an existing virtual machine, please see the [Modifying Virtual Machine Configuration](#) section (p. 85).

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

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

    PRL_RESULT ret = PRL_ERR_UNINITIALIZED;
    PRL_RESULT nJobRetCode = PRL_ERR_UNINITIALIZED;

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

    ret = PrlJob_Wait(hJob, 10000);

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

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

    ret = PrlResult_GetParam(hJobResult, &hVm);
    if (PRL_FAILED(ret))
```

```

{
    // Handle the error...
    return ret;
}

PrlHandle_Free(hJob);
PrlHandle_Free(hJobResult);

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

// The following code will generate and set a new MAC address
// for every virtual network adapter that exists in the virtual machine.
// This step is optional and should normally be performed when the virtual
// machine is a copy of another virtual machine.

PRL_HANDLE hJobBeginEdit = PRL_INVALID_HANDLE;
PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;

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

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

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

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

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

```

```
        return ret;
    }
}

// Commit the changes to the virtual 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;
}

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

return 0;
}
```

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 Service. 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](#) (p. 101) section for more virtual machine info and examples.

The source virtual machine must be registered with the Parallels Service 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.

```
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.
    // 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 the virtual machines registered with the Service. 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 host computer, 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, and the directory (or just some of its files that you can specify) will be deleted.

The following example demonstrates un-registering a virtual machine. 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 C API provides a complete set of functions to modify the configuration parameters of an existing virtual machine. You can find the list of the available functions in the **Parallels C API Reference** guide by looking at the `PHT_VM_CONFIGURATION` group. Most of the get/set functions in the group allow to obtain and modify the virtual 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 the Parallels Service.
- you've already obtained a handle to the virtual machine that you would like to modify.

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 Parallels Service 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_INVALID_HANDLE;
PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
PRL_RESULT nJobRetCode = PRL_INVALID_HANDLE;

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

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

// Commits the changes to the virtual 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_INVALID_HANDLE;
PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
PRL_RESULT nJobRetCode = PRL_INVALID_HANDLE;

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

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

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

// Modify boot options.
// Set boot device list as follows:
// 0. CD/DVD drive.
// 1. Hard disk.
// 2. Network adapter.
// 3. Floppy disk drive.
// Remove all other devices (if any) from the
// boot devices list for this VM.

// Device count.
PRL_UINT32 nDevCount;

// A handle identifying the device.
PRL_HANDLE hDevice = PRL_INVALID_HANDLE;

// Device type.
PRL_DEVICE_TYPE devType;

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

// Iterate through the device list.
```

```

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

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

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

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

```

RAM Size

The size of the memory available to the virtual 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_INVALID_HANDLE;
PRL_UINT32 nCount;

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

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

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

Adding a new hard disk

In this example, we will add a hard disk to a virtual 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_INVALID_HANDLE;
PRL_HANDLE hJobCommit = PRL_INVALID_HANDLE;
PRL_RESULT nJobRetCode = PRL_INVALID_HANDLE;

// Timestamp the beginning of the configuration changes operation.
// The hVm specifies the virtual 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 = PRL_INVALID_HANDLE;
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 = PRL_INVALID_HANDLE;
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 sample function illustrates how to add virtual network adapters using the host-only and shared networking (both types are created similarly). The steps are:

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

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

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

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

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

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

    // For host-only networking, set the device emulation type
```

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

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

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

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

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

return PRL_ERR_SUCCESS;
}
```

Bridged Networking

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

- 1 Obtain a list of network adapters installed on the host. This step is performed using the `PrlSrvCfg_GetNetAdaptersCount`, `PrlSrvCfg_GetNetAdapter`, and `PrlSrvCfgDev_GetName` functions.
- 2 Begin the virtual machine editing operation and create a new network adapter handle as described in the [Host-only and Shared Networking](#) section (p. 93).
- 3 Bind the new virtual network adapter to the desired host machine adapter using the `PrlVmDevNet_SetBoundAdapterName` function.
- 4 Finalize the changes by calling the `PrlVm_Commit` function.

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

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

Example 1:

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

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

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

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

    // Get job results.
    ret = PrlJob_GetResult(hJob, &hJobResult);
    if (PRL_FAILED(ret))
```

```

{
    // Handle the error.
}

// server config handle.
PRL_HANDLE hSrvCfg = PRL_INVALID_HANDLE;

// counter.
PRL_UINT32 nCount = PRL_INVALID_HANDLE;

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

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

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

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

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

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

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

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

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

// Create a virtual network adapter device handle.
PRL_HANDLE hNet = PRL_INVALID_HANDLE;
ret = PrlVmCfg_CreateVmDev(
    hVmCfg, // The virtual machine configuration handle.

```

```

        PDE_GENERIC_NETWORK_ADAPTER, // Device type.
        &hNet); // Device handle.

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

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

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

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

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

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

    return PRL_ERR_SUCCESS;
}

```

Example 2:

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

```

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

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

```

```

    PrlHandle_Free(hJobBeginEdit);
    return nJobRetCode;
}

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

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

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

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

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

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

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

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

return PRL_ERR_SUCCESS;
}

```

Managing User Access Rights

This topic applies to Parallels Server only.

User authorization (determining user access rights) 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 Service 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 C API.

The Parallels C API contains a `PHT_ACCESS_RIGHTS` object that is used to manage user access rights. A handle to it is obtained using the `PrlVmCfg_GetAccessRights` or the `PrlVmInfo_GetAccessRights` function. The difference between the two function is that `PrlVmInfo_GetAccessRights` takes an additional step: obtaining a handle of type `PHT_VM_INFO` which will also contain the virtual 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 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 host). The host 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 Service user registry (p. 45)). 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 C 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;
}

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

```

```
        return -1;
    }

    // Determine if the current user has the right to
    // start the virtual 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 C API allows to perform the following template-related operations:

- Obtaining a list of the available virtual machine templates.
- 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

A list of virtual machines and virtual machine templates are obtained from the server using the same function: `PrlSrv_GetVmList` (p. 59). 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 (p. 59) 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(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)
{
    // Virtual machine handle
    PRL_HANDLE hVm = PRL_INVALID_HANDLE;

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

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

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

    // If this is not a template, proceed to the next
    // virtual 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 with the Parallels Service, 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 (p. 72) 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 (p. 72) 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 Service 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 registered virtual machine 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 (p. 82). 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 (p. 82) 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.
    // 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;
}
```

Events

In This Chapter

Receiving and Handling Events	111
Responding to Parallels Service Questions	114

Receiving and Handling Events

Parallels Service and all running virtual machines are constantly monitored for any changes in their state and status. When something important changes, 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 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](#) (p. 75) 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 (p. 22). If you haven't read it yet, please do so now. To subscribe to event notifications, you must register your event handler with the Service. 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 Service. The code inside the event handler can then handle the event according to the application logic.

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

- 1 Determine if the notification received is an event (not a *job*, because event handlers are also called when an asynchronous job begins). This can be accomplished using the `PrlHandle_GetType` function (determines the type of the handle received) and then checking if the handle is of type `PHT_EVENT` (not `PHT_JOB`).
- 2 Determine the type of the event using the `PrlEvent_GetType` function. Check the event type against the `PRL_EVENT_TYPE` enumeration. If it is relevant, continue to the next step.
- 3 If needed, you can use the `PrlEvent_GetIssuerType` or `PrlEvent_GetIssuerId` function to find out what part of the system triggered the event. This could be a host, a virtual 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 C API Reference](#).
- 7 When finished, release the received event handle. This step is necessary regardless of if you actually used the handle or not. Failure to release the handle will result in a memory leak.

The following is a simple event handler function that illustrates the implementation of the steps described above. We are not including an example of how to register an event handler here, please see the [Asynchronous Functions](#) section (p. 22) 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 Service configuration has been modified. \n");
        break;
    default:
        printf("Unhandled event: %d\n", eventType);
}
}
```

Responding to Parallels Service Questions

One of the event types in the `PRL_EVENT_TYPE` enumeration deserves special attention. This event type is `PET_DSP_EVT_VM_QUESTION`. While processing a task, a Parallels Service 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 host. Since virtual hard drives can dynamically allocate disk space, this is not necessarily a reason to abort the operation. In such a case, the Service 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 Service. This is performed in two steps: first, the `PrlEvent_CreateAnswerEvent` function is used to properly format the answer; second, the answer is sent to the Service using the `PrlSrv_SendAnswer` function.

The following is a complete example that demonstrates how to handle events of type `PET_DSP_EVT_VM_QUESTION` and how to answer Service 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 Service. We then send an answer and the operation continues normally.

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

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

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

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

// A helper function that will attempt to create a hard drive larger
```

```

// than the free space available, thus triggering an event on the
// server.
static PRL_RESULT create_big_hdd(PRL_HANDLE hVm);

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

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

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

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

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

    // Initialize API library. In this example, we are initializing the
    // API for Parallels Server.
    // To initialize in the Parallels Desktop mode, pass PAM_DESKTOP
    // as the second parameter.
    // To initialize for Parallels Workstation, pass PAM_WORKSTATION.
    // See the PRL_APPLICATION_MODE enumeration for other options.
    err = PrlApi_InitEx(PARALLELS_API_VER, PAM_SERVER, 0, 0);
    if (PRL_FAILED(err))
    {
        // Error handling goes here...
        return -1;
    }

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

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

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

    if (PRL_FAILED(ret))

```

```

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

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

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

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

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

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

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

    // We can now clean up and exit the program.
    // Unregister the event handler and log off.
    PrlSrv_UnregEventHandler(hServer, &callback, NULL);
    hJob = PrlSrv_Logoff(hServer);
    PrlJob_Wait(hJob, MY_JOB_TIMEOUT);
    PrlHandle_Free( hJob );
    PrlHandle_Free( hServer );
    PrlApi_Deinit();
    SdkWrap_Unload();
    return 0;
}

```

```

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

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

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

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

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

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

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

        // Now, 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_INVALID_HANDLE;
            PRL_RESULT err = PRL_ERR_UNINITIALIZED;

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

```

```

        prl_result_to_string(err));
        PrlHandle_Free(hParam);
        PrlHandle_Free(hEvent);
        return err;
    }

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

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

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

// Obtain a server handle. We need it to send an answer.
PRL_HANDLE hServer = PRL_INVALID_HANDLE;
PrlEvent_GetServer(hEvent, &hServer);

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

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

return PRL_ERR_SUCCESS;
}

```

```

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

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

    // Create a new device handle.
    // This will be our new virtual hard disk.
    PRL_HANDLE hHDD = PRL_INVALID_HANDLE;
    err = PrlVmCfg_CreateVmDev(
        hVm, // Target virtual 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;
}
```

Performance Statistics

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

- 1 Using `PrlSrv_GetStatistics` (for host statistics) or `PrlVm_GetStatistics` (for virtual machine statistics) to obtain a report containing the latest performance data.
- 2 Using `PrlSrv_SubscribeToHostStatistics` (for host 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	122
Performance Monitoring	125

Obtaining Performance Report

The first step required to obtain a 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 be only one parameter returned).

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

C API Reference Section	Description
<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 can be obtained using <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 host statistics (PHT_SYSTEM_STATISTICS handle), and wait for a
// maximum of 10 seconds for the asynchronous call PrlSrv_GetStatistics to
// complete.
// Note: PrlVm_GetStatistics(hVm) could be used instead of
// PrlSrv_GetStatistics(hServer) if statistics are required for a
// virtual 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 = PRL_INVALID_HANDLE;
nServerStatistics = PrlJob_GetResult(hServerStatisticsJob, &hResult);
if (PRL_FAILED(nServerStatistics))
{
    printf("PrlJob_GetResult returned error: %s\n",
        prl_result_to_string(nServerStatistics));
    PrlHandle_Free(hServerStatisticsJob);
    PrlHandle_Free(hServer);
    PrlApi_Deinit();
    SdkWrap_Unload();
    return -1;
}

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

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

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

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

// Get disk statistics.
PRL_UINT64 nFreeDiskSpace, nUsedDiskSpace;
PRL_HANDLE hDiskStatistics = PRL_INVALID_HANDLE;
PrlStat_GetDiskStat(hServerStatistics, 0, &hDiskStatistics);
PrlStatDisk_GetFreeDiskSpace(hDiskStatistics, &nFreeDiskSpace);
PrlStatDisk_GetUsageDiskSpace(hDiskStatistics, &nUsedDiskSpace);
printf("Used Disk Space: %I64d MB\nFree Disk Space: %I64d MB\n",
    nUsedDiskSpace/1024/1024, nFreeDiskSpace/1024/1024);

```

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

Performance Monitoring

To monitor the host 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");
        }

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

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

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

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

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

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

PrlHandle_Free(handle);

return PRL_ERR_SUCCESS;
}

```

```

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

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

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

    // Initialize the Parallels API. In this example, we are initializing the
    // API for Parallels Server.
    // To initialize in the Parallels Desktop mode, pass PAM_DESKTOP
    // as the second parameter.
    // To initialize for Parallels Workstation, pass PAM_WORKSTATION.
    // See the PRL_APPLICATION_MODE enumeration for other options.
    ret = PrlApi_InitEx(PARALLELS_API_VER, PAM_SERVER, 0, 0);
    if (PRL_FAILED(ret))
    {
        fprintf(stderr, "PrlApi_InitEx returned with error: %s.\n",
            prl_result_to_string(ret));
        PrlApi_Deinit();
        SdkWrap_Unload();
        return ret;
    }

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

```

```

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

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

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

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

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

// -----

// Log off.
hJob = PrlSrv_Logoff(hServer);
ret = PrlJob_Wait(hJob, 1000);

```

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

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

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

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

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

return 0;
}
```

Encryption Plug-in

A Parallels virtual machine can be encrypted to make its files unreadable to anyone except those authorized to do so. When configuring a virtual machine, you have an option to encrypt its files if desired. When this option is selected, the user is asked to enter a password and then the virtual machine files are encrypted using an encryption algorithm. Once this is done, the virtual machine cannot be started without providing the correct password and its files cannot be read using standard Parallels or third party utilities. An encryption module is provided by Parallels and is built into the Parallels Desktop. The built-in module uses a sufficiently strong and fast encryption algorithm. If, however, you would like to use your own (or third-party) encryption module, the Parallels Virtualization SDK provides an API that allows to custom built an encryption plug-in and then use it instead of the built-in one. The following sub-sections describes how to build and use such a plug-in.

Encryption Plug-in Basics

To build and use an encryption plug-in you have to perform the following steps:

- 1** Develop a plug-in according to specifications provided here and build it as a dynamic library.
- 2** Place the dynamic library file into designated directory, which is created automatically when you install Parallels Desktop; then set the necessary permissions for the file.
- 3** Enable the third-party plug-in support in Parallels Desktop preferences.
- 4** The Parallels Desktop will recognize and load the plug-in. The plug-in will then appear in the list of the encryption engines available for encrypting a virtual machine.

All of the steps above are discussed in detail in the following sub-sections.

Please note that developing an encryption plug-in is different than developing client applications described earlier in this guide. All you need to know about developing an encryption plug-in is contained in this portion of the guide.

The Encryption API Reference

The encryption plug-in is designed using the Component Object Model (COM) principles. A plug-in is a component (class) identified by class ID, which is a globally unique identifier (GUID). The component exposes its functionality through interfaces. Each interface provided by the component is also identified by a GUID. The access to the component is done through methods of the interfaces. The encryption plug-in interfaces are defined in the `PrlPluginClasses.h` header file, which is included in the Parallels Virtualization SDK. Developing a plug-in involves implementing the component, the interfaces, and a number of functions (some optional, some mandatory) that have to be exported from the dynamic library. This section provides a reference information. For a sample plug-in implementation please refer to the [Implementing a Plug-in](#) section (p. 134).

Constants

`PRL_CLS_NULL` is a constant that defines the NULL GUID used as a terminator in the interfaces array (the array is used to hold the list of the interfaces supported by the component and is declared and populated in the plug-in implementation).

```
static const PRL_GUID PRL_CLS_NULL = { 0, 0, 0, {0, 0, 0, 0, 0, 0, 0, 0} };
```

`PRL_CLS_BASE` is a constant that defines the GUID of the base interface. The base interface provides access to the base plug-in functionality, such as creating objects, getting the plug-in info, and memory management. The base interface name is `PrlPlugin` and it's described later in this section.

```
static const PRL_GUID PRL_CLS_BASE = { 0x823067ea, 0x8e15, 0x474f,
    { 0xa3, 0xae, 0xc6, 0xa0, 0x68, 0x46, 0x12, 0x56 } };
#define GUID_CLS_BASE_STR "{823067ea-8e15-474f-a3ae-c6a068461256}"
```

`PRL_CLS_ENCRYPTION` is a constant that defines the encryption interface GUID. The encryption interface provides access to the data encryption functionality. The encryption component name is `PrlCrypt` and it's described later in this section.

```
static const PRL_GUID PRL_CLS_ENCRYPTION = { 0x564820fc, 0xe265, 0x4d69,
    { 0x9f, 0xb0, 0x3d, 0x18, 0x39, 0x6f, 0x1f, 0x8d } };
#define GUID_CLS_ENCRYPTION_STR "{564820fc-e265-4d69-9fb0-3d18396f1f8d}"
```

Structures

`IPluginInfo` is a structure that's used to hold general plug-in information.

```
typedef struct _IPluginInfo
{
    PRL_STR Vendor; // Vendor info.
    PRL_STR Copyright; // Copyright info.
    PRL_STR DescShort; // Short description.
    PRL_STR DescLong; // Long description.
    PRL_UINT32 Major; // Major version number.
    PRL_UINT32 Minor; // Minor version number.
    PRL_UINT32 Build; // Build number.
    PRL_GUID Type; // Plug-in GUID.
} PRL_STRUCT(IPluginInfo);
typedef IPluginInfo* IPluginInfoPtr;
```

`ICryptInfo` is a structure that's used to hold the plug-in info (`PluginInfo`) together with the key and block size values used for data encryption.

```
typedef struct _ICryptInfo
{
```

```

    IPluginInfo PluginInfo;
    PRL_UINT32 KeySize;
    PRL_UINT32 BlockSize;
} PRL_STRUCT(ICryptInfo);
typedef ICryptInfo* ICryptInfoPtr;

```

Interfaces

PrIPlugin is an interface that provides access to the base plug-in functionality, such as creating objects, getting the plug-in info, and memory management.

```

typedef struct _PrIPlugin
{
    // Releases the memory occupied by the structure.
    void (PRL_CALL *Release)(struct _PrIPlugin* _self);

    // Creates a specified interface.
    // This method is called by Parallels Service to create
    // the base and encryption interfaces.
    // The Class parameter specifies the GUID of
    // the interface to create: PRL_CLS_BASE or PRL_CLS_ENCRYPTION.
    // The _obj parameter receives a reference to the created interface.
    PRL_RESULT (PRL_CALL *QueryInterface)(struct _PrIPlugin* _self,
        PRL_GUID* Class, PRL_VOID_PTR_PTR _obj);

    // Obtains a reference to the IPluginInfo structure containing the
    // plug-in info.
    PRL_RESULT (PRL_CALL *GetInfo)(struct _PrIPlugin* _self, IPluginInfoPtr
Info);
} PRL_STRUCT( PrIPlugin )

```

PrICrypt is an interface that provides access to the data encryption functionality.

```

typedef struct _PrICrypt
{
    // Inheritance from PrIPlugin.
    struct _PrIPlugin Plugin;

    // Initializes the encryption engine.
    PRL_RESULT (PRL_CALL *Init)(struct _PrICrypt* _self);

    // Encrypts the supplied data block.
    PRL_RESULT (PRL_CALL *Encrypt)(struct _PrICrypt* _self, PRL_VOID_PTR Data,
        const PRL_UINT32 Size, const PRL_UINT8_PTR
v);

    // Decrypts the supplied data block.
    PRL_RESULT (PRL_CALL *Decrypt)(struct _PrICrypt* _self, PRL_VOID_PTR Data,
        const PRL_UINT32 Size, const PRL_UINT8_PTR
v);

    // Sets the encryption key. The key size must be equal to or
    // larger than the one specified in the ICryptInfo structure.
    PRL_RESULT (PRL_CALL *SetKey)(struct _PrICrypt* _self, const PRL_UINT8_PTR
Key);

    // Sets the initial initialization vector. The vector size must be
    // equal to or larger than the one specified in the ICryptInfo structure
    // as the block size.
    PRL_RESULT (PRL_CALL *SetInitIV)(struct _PrICrypt* _self, const
PRL_UINT8_PTR v);

    // Obtains a reference to the ICryptInfo structure containing the plug-in
    // information.
    PRL_RESULT (PRL_CALL *GetInfo)(struct _PrICrypt* _self, ICryptInfoPtr
Info);

```

```
} PRL_STRUCT( PrlCrypt );
```

Exported functions

PrlInitPlugin is a function that is called on plug-in loading and should contain the code to do some global plug-in initialization. This function is optional.

```
PRL_RESULT PRL_CALL PrlInitPlugin()
```

PrlFiniPlugin is a function that is called on plug-in unloading and should contain the clean-up code. This function is optional.

```
PRL_RESULT PRL_CALL PrlFiniPlugin()
```

PrlGetObjectInfo is a function that is called after plug-in initialization in order to enumerate the plug-in interfaces. The `InterfacesList` parameter is used as output and should contain the list of GUIDs of public interfaces provided by the plug-in.

```
PRL_RESULT PRL_CALL PrlGetObjectInfo(PRL_UINT32 Number, PRL_GUID* Uid,  
PRL_GUID** InterfacesList)
```

PrlCreateObject is a function that is called in order to instantiate the corresponding plug-in object. The function should contain a code that will properly initialize and populate the `PrlPlugin` structure a reference to which is passed back using the `Out` parameter. The Parallels Service will then use the returned reference to request the necessary interfaces (e.g. `PrlCrypt`) using the `PrlPlugin::QueryInterface()` call.

```
PRL_RESULT PRL_CALL PrlCreateObject(PRL_GUID* Uid, PrlPlugin** Out)
```

The following subsection provides a sample plug-in implementation.

Implementing a Plug-in

This section goes step by step through a sample plug-in implementation and describes each step in detail. This should give you an understanding of how to implement the plug-in and its interfaces. An encryption engine used in this example is a simple XOR cipher. This is a simple algorithm that can be easily broken, so we use it as a demonstration only. When developing your own plug-in, you should use a stronger encryption algorithm according to your needs.

Include directives

A plug-in program must have the following include directives:

```
#include "PrlTypes.h"
#include "PrlErrors.h"
#include "PrlPluginClasses.h"
```

All of the above header files are provided in the Parallels Virtualization SDK. You have to make sure that your compiler can find them on your computer. See the [Installation](#) chapter for the default SDK installation location.

This sample program also uses the following standard includes:

```
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <new>
```

Plug-in GUID

A plug-in must have its own unique ID (GUID) in order to be properly identified on the Parallels Service side. The following is an example of GUID declaration and initialization. You will have to generate a GUID for your own plug-in.

```
static PRL_GUID Obj = { 0x801ecba5, 0x909a, 0x42fc, { 0x81, 0x62, 0x1e, 0x7c,
0x51, 0x9e, 0x84, 0x6e } };
```

Components (classes) array

Declare and populate an array containing the list of GUIDs identifying the available interfaces. The array is terminated with `PRL_CLS_NULL`, which is a NULL GUID.

```
static PRL_GUID Classes[] = { PRL_CLS_BASE, PRL_CLS_ENCRYPTION, PRL_CLS_NULL
};
```

Encryption block size

Define the encryption data block size for the XOR cipher.

```
#define BLOCK_SIZE (16)
```

Define the encryption interface

`PrlCrypt` is the interface that contains pointers to the data encryption functions. We will implement individual functions later in this example. Right now we will declare the component as a member of a structure. We will also add two more members to the structure to hold the initialization vector and the encryption key.

```
typedef struct _IXorCrypt
{
```

```

PrlCrypt I;

// Initialization vector
PRL_UINT8 m_IV[BLOCK_SIZE];
// Key
PRL_UINT8 m_Key[BLOCK_SIZE];
} IXorCrypt;

```

Provide the plug-in description

Here we populate the plug-in information structure.

```

static const ICryptInfo PlugInfo =
{
    {
        (PRL_STR)"Sample",
        (PRL_STR)"(c)Sample",
        (PRL_STR)"XOR encryption engine",
        (PRL_STR)"This engine utilizes the XOR cipher",
        1, 0, 0,
        Obj
    },
    BLOCK_SIZE, BLOCK_SIZE
};

```

Free the memory occupied by the base interface

`PrlPlugin` is the base interface. It contains an important function that must be implemented. The function frees the memory occupied by the `PrlPlugin` structure. This has to be done because the caller will have no knowledge of how this structure was allocated. We implement it as a standalone function and will use a reference to it in the `PrlPlugin` structure declaration.

```

void PRL_CALL Release(PrlPlugin* _self)
{
    free(_self);
}

```

Initialize the encryption engine

In this step we implement a function that will initialize the encryption engine. The function prototype is defined in the `PrlCrypt` structure.

```

PRL_RESULT PRL_CALL Init(PrlCrypt* _self)
{
    IXorCrypt* I = (IXorCrypt*)_self;

    memset(I->m_IV, 0, BLOCK_SIZE);
    memset(I->m_Key, 0, BLOCK_SIZE);
    return PRL_ERR_SUCCESS;
};

```

Main encryption engine

In this step we create a function that implements the actual encryption algorithm. The function uses a simple XOR encryption where a bitwise XOR operator is applied to a block of data using the specified key.

```

static void XorFunc(PRL_UINT8_PTR a, PRL_UINT8_PTR b)
{
    PRL_UINT64* p1 = (PRL_UINT64*)a;
    PRL_UINT64* p2 = (PRL_UINT64*)b;

    p1[0] ^= p2[0];
    p1[1] ^= p2[1];
}

```

```
}

```

This function encrypts the provided data block.

```
PRL_RESULT PRL_CALL Encrypt(PrlCrypt* _self, PRL_VOID_PTR Data,
                            const PRL_UINT32 Size, const PRL_UINT8_PTR v)
{
    if ((Size % BLOCK_SIZE) ||
        !Data)
        return PRL_ERR_INVALID_ARG;

    IXorCrypt* I = (IXorCrypt*)_self;
    PRL_UINT32 Count = Size / BLOCK_SIZE;
    PRL_UINT8_PTR Cur = (PRL_UINT8_PTR)Data;

    for(PRL_UINT32 i = 0; i < Count; i++, Cur += BLOCK_SIZE)
    {
        XorFunc(Cur, I->m_Key);
        XorFunc(Cur, I->m_IV);
        if (v)
            XorFunc(Cur, v);
    }

    return PRL_ERR_SUCCESS;
}
```

This function decrypts the provided data block.

```
PRL_RESULT PRL_CALL Decrypt(PrlCrypt* _self, PRL_VOID_PTR Data,
                             const PRL_UINT32 Size, const PRL_UINT8_PTR v)
{
    if ((Size % BLOCK_SIZE) ||
        !Data)
        return PRL_ERR_INVALID_ARG;

    IXorCrypt* I = (IXorCrypt*)_self;
    PRL_UINT32 Count = Size / BLOCK_SIZE;
    PRL_UINT8_PTR Cur = (PRL_UINT8_PTR)Data;

    for(PRL_UINT32 i = 0; i < Count; i++, Cur += BLOCK_SIZE)
    {
        if (v)
            XorFunc(Cur, v);
        XorFunc(Cur, I->m_IV);
        XorFunc(Cur, I->m_Key);
    }

    return PRL_ERR_SUCCESS;
}
```

This function is used to set a key for the encryptor. The key size must be equal to or larger than the one specified in the PlugInfo structure (declared above).

```
PRL_RESULT PRL_CALL SetKey(PrlCrypt* _self, const PRL_UINT8_PTR Key)
{
    IXorCrypt* I = (IXorCrypt*)_self;

    if (!Key)
    {
        memset(I->m_Key, 0, BLOCK_SIZE);
        return PRL_ERR_SUCCESS;
    }

    memcpy(I->m_Key, Key, BLOCK_SIZE);

    return PRL_ERR_SUCCESS;
}
```

This function sets the initial initialization vector. The vector size must be equal to the one specified in the `PlugInfo` structure as the block size.

```
PRL_RESULT PRL_CALL SetInitIV(PrlCrypt* _self, const PRL_UINT8_PTR v)
{
    IXorCrypt* I = (IXorCrypt*)_self;

    if (!v)
    {
        memset(I->m_IV, 0, BLOCK_SIZE);
        return PRL_ERR_SUCCESS;
    }

    memcpy(I->m_IV, v, BLOCK_SIZE);

    return PRL_ERR_SUCCESS;
}
```

Functions to obtain the plug-in information

The first function obtains a reference to the structure containing the plug-in description.

```
PRL_RESULT PRL_CALL GetBaseInfo(PrlPlugin* _self, IPluginInfoPtr Info)
{
    (void)_self;

    if (!Info)
        return PRL_ERR_INVALID_ARG;

    *Info = PlugInfo.PluginInfo;

    return PRL_ERR_SUCCESS;
}
```

The second function obtains a reference to the structure that contains the plug-in description together with the data encryption key and block block sizes.

```
PRL_RESULT PRL_CALL GetInfo(PrlCrypt* _self, ICryptInfoPtr Info)
{
    (void)_self;

    if (!Info)
        return PRL_ERR_INVALID_ARG;

    *Info = PlugInfo;

    return PRL_ERR_SUCCESS;
}
```

The QueryInterface function

This function accepts the GUID of an interface and passes back a reference to the interface instance.

```
PRL_RESULT PRL_CALL QueryInterface(PrlPlugin* _self, PRL_GUID* Class,
PRL_VOID_PTR_PTR _obj)
{
    if (!_self || !_obj)
        return PRL_ERR_INVALID_ARG;

    // Want to instantiate the base or encryption interface?
    if (memcmp(Class, &Classes[0], sizeof(PRL_GUID)) &&
        memcmp(Class, &Classes[1], sizeof(PRL_GUID)))
    {
        return PRL_ERR_OBJECT_NOT_FOUND;
    }
}
```

```

}

// The interface pointer is the same for both objects
*_obj = (PRL_VOID_PTR)_self;

return PRL_ERR_SUCCESS;
}

```

Functions that must be exported

All exported functions must have C style declaration to properly resolve names at loading..

```

extern "C"
{
/*
 * This function will be called immediately after the plug-in is loaded.
 * You can put any code here that might be needed in your implementation.
 * The function is optional.
 */
__attribute__((visibility("default"))) PRL_RESULT PRL_CALL PrlInitPlugin()
{
    return PRL_ERR_SUCCESS;
}
/*
 * This function will be called immediately before the plug-in is unloaded.
 * You can put any code here that might be needed in your implementation.
 * The function is optional.
 */
__attribute__((visibility("default"))) PRL_RESULT PRL_CALL PrlFiniPlugin()
{
    return PRL_ERR_SUCCESS;
}

/*
 * This function is used to obtain all available interfaces provided by the
plug-in.
 * The Number parameter is an interface number; the Uid and Classes parameters
 * are object information that should be filled to caller.
 */
__attribute__((visibility("default"))) PRL_RESULT PRL_CALL
PrlGetObjectInfo(PRL_UINT32 Number,
                 PRL_GUID* Uid, PRL_GUID** InterfacesList)
{
    if (Number != 0)
        return PRL_ERR_OBJECT_NOT_FOUND;

    if (!Uid || !InterfacesList)
        return PRL_ERR_INVALID_ARG;

    *Uid = Obj;
    *InterfacesList = Classes;

    return PRL_ERR_SUCCESS;
}

/*
 * This function is used to create a specified object and put a
 * reference to it into the Out variable.
 */
__attribute__((visibility("default"))) PRL_RESULT PRL_CALL
PrlCreateObject(PRL_GUID* Uid, PrlPlugin** Out)
{
    if (!Uid || !Out)
        return PRL_ERR_INVALID_ARG;

    if (memcmp(Uid, &Obj, sizeof(PRL_GUID)))

```

```
        return PRL_ERR_OBJECT_NOT_FOUND;

    IXorCrypt* I = (IXorCrypt*)malloc(sizeof(IXorCrypt));

    if (!I)
        return PRL_ERR_OUT_OF_MEMORY;

    // Cleanup memory
    memset(I, 0, sizeof(IXorCrypt));

    /*
     * Fill functions table. If you skip something, you'll get
     * SEGFAULT if there was no memset, or unpredictable errors
     * if function address is equal to NULL.
     */
    PrlPlugin* Base = (PrlPlugin*)I;
    Base->Release = &Release;
    Base->GetInfo = &GetBaseInfo;
    Base->QueryInterface = &QueryInterface;

    PrlCrypt* Crypt = (PrlCrypt*)I;
    Crypt->Init = &Init;
    Crypt->Encrypt = &Encrypt;
    Crypt->Decrypt = &Decrypt;
    Crypt->SetKey = &SetKey;
    Crypt->SetInitIV = &SetInitIV;
    Crypt->GetInfo = &GetInfo;

    *Out = (PrlPlugin*)I;

    return PRL_ERR_SUCCESS;
}
}
```

Building the Dynamic Library

The plug-in must be compiled as a dynamic library. You can use the following sample Makefile to compile the sample program on Mac OS X.

```
#####  
# Makefile for building: libsampler_plugin.dylib  
#  
# Copyright (c) 2005-2011 Parallels Software International, Inc.  
# All rights reserved.  
# http://www.parallels.com  
#####  
  
CC = g++  
RM = rm -f  
  
CFLAGS = -c -Wall -arch i386 -arch x86_64 -O2 -gdwarf-2 -fvisibility=hidden -  
O2 -fPIC -mmacosx-version-min=10.5  
  
INCPATH = -I../Library/Frameworks/ParallelsVirtualizationSDK.framework/Headers  
  
LDFLAGS = -arch x86_64 -mmacosx-version-min=10.5 -arch i386 -single_module -  
dynamiclib -compatibility_version 1.0 -current_version 1.0.0 -  
install_name libsampler_plugin.1.dylib  
  
LIBRARY = libsampler_plugin.dylib  
  
SRCFILE = Plugin.cpp  
  
OBJFILE = $(SRCFILE:.cpp=.o)  
  
.cpp.o:  
    $(CC) $(CFLAGS) $(INCPATH) $<  
  
$(LIBRARY): $(OBJFILE)  
    $(CC) $(LDFLAGS) $(OBJFILE) -o $@  
  
all: $(LIBRARY)  
  
clean:  
    $(RM) *.o *.dylib
```

Plug-in Installation and Usage

Plug-in installation directory and permissions

After building the plug-in:

- 1 copy the dynamic library to the `/usr/lib/parallels/extensions` directory on your Mac. The directory is created when you install Parallels Desktop.
- 2 Modify the plug-in file permissions. The owner of the file should be the `root` user. All users and groups, including `root`, should have *read-only* access (`-r-xr-xr-x`) to the file. If anybody has a write access, the plug-in will fail to load!

Enabling third-party plug-in support

Before you can use the plug-in, you have to turn on third-party plug-in support in Parallels Desktop preferences. Select **Preferences** from the Parallels Desktop menu. On the Preferences window go to **Advanced** and select the **Allow third-party plug-ins** option. When this option is selected, Parallels Desktop will scan the plug-in directory and will load the new plug-in (if you have more than one plug-in, it will load all of them).

Encrypting a virtual machine

To encrypt a virtual machine using the encryption plug-in, open the virtual machine configuration and go to **Options/Security**. Press the **Encryption Turn On...** button. Select the encryption plug-in in the **Encryption Engine** list. Choose and type a password and click **OK**.

Parallels Python API Concepts

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

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

In This Chapter

Package and Modules	143
Classes	144
Class Methods.....	144
Error Handling	147

Package and Modules

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

<code>prlsdkapi</code>	This is the main package containing the majority of the classes.
<code>prlsdkapi.prlsdk</code>	<i>This is an internal module. Do not use it in your applications.</i>
<code>prlsdkapi.prlsdk.consts</code>	<p>This module contains constants that are used throughout the API. Most of the constants are combined into groups which are used as pseudo enumerations. Constants that belong to the same group have the same prefix in their names. For example, constants with a <code>PDE_</code> prefix identify device types: <code>PDE_GENERIC_DEVICE</code>, <code>PDE_HARD_DISK</code>, <code>PDE_GENERIC_NETWORK_ADAPTER</code>, etc.</p> <p>In this guide, and in the Parallels Python API Reference guide, we identify individual groups of constants using these prefixes. For example, we might say, "for the complete list of device types, see the <code>PDE_xxx</code> constants".</p>
<code>prlsdkapi.prlsdk.errors</code>	This module contains error code constants. There's a very large amount of error codes but the majority of them are used internally. The error code constant are also grouped using prefixes in their names.

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

Classes

Compared to the Parallels C API, a Python class is an equivalent of a C handle. In most cases, an instance of a class must be obtained using a method of another class. Instances of particular classes are obtained in a certain order. A typical program must first obtain an instance of the `prlsdkapi.Server` class identifying the Parallels Service. If the intention is to work with a virtual machine, an instance of the `prlsdkapi.Vm` class identifying the virtual machine must then be obtained using the corresponding methods of the `prlsdkapi.Server` class. To view or modify the virtual machine configuration setting, an instance of the `prlsdkapi.VmConfig` class must be obtained using a method of the `prlsdkapi.Vm` class, and so forth. The examples in this guide provide information on how to obtain the most important and commonly used objects (server, virtual machine, devices, etc.). In general, an instance of a class is obtained using a method of a class to which the first class logically belongs. For example, a virtual machine belongs to a server, so the `Server` class must be used to obtain the virtual machine object. A virtual device belongs to a virtual machine, so the virtual machine object must be used to obtain a device object, and so on. In some cases an object must be created manually, but these cases are rare. The most notable one is the `prlsdkapi.Server` class, which is created using the `server = prlsdkapi.Server()` statement in the very beginning of a typical program.

Class Methods

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

Synchronous Methods

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

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

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

Synchronous methods throw the `prlsdkapi.PrlSDKError` exception. For more information on exceptions, see the **Error Handling** section (p. 147).

Asynchronous Methods

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

Checking the job status

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

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

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

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

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

```
# Start the virtual machine.
job = vm.start()

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

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

Suspending the main thread

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

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

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

# The job is now finished and our program continues...
vm_config = vm.get_config()
```

```
print vm_config.get_name() + " was started."
```

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

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

Obtaining the job results

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

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

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

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

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

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

Other useful Job methods

The `Job` class provides other useful methods:

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

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

Asynchronous methods throw the `prlsdkapi.PrlSDKError` exception. For more information on exceptions, see the [Error Handling](#) section (p. 147).

Error Handling

Error handling in the Parallels Python API is done through the use of exceptions that are caught in `try` blocks and handled in `except` blocks. All methods in the API throw the `prlsdkapi.PrlSDKError` exception. The `PrlSDKError` object itself contains the error message. To obtain the error code, examine the `prlsdkapi.PrlSDKError.error_code` argument. The error code can be evaluated against standard Parallels API errors, which can be found in the `prlsdkapi.prlsdk.errors` module. The most common error codes are `PRL_ERR_SUCCESS`, `PRL_ERR_INVALID_ARG`, `PRL_ERR_OUT_OF_MEMORY`. For the complete list of errors, see the `prlsdkapi.prlsdk.errors` module documentation or the [Parallels Python API Reference guide](#).

The following code sample illustrates the exception handling:

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

Parallels Python API by Example

This chapter provides code samples and descriptions of how to perform the most common tasks using the Parallels Python API. Each sample is provided as a complete function that can be used in your own program. Each sample function accepts a parameter -- usually an instance of the `prlsdkapi.Server` class identifying the Parallels Service or an instance of the `prlsdkapi.Vm` class identifying a virtual machine. The **Creating a Basic Application** section (p. 149) shows how to create and initialize the `prlsdkapi.Server` object and contains a skeleton program that can be used to run individual examples provided later in this chapter. To run the examples, simply paste a sample function into the program and then call it from `main()` passing the correct object and/or other required values.

In This Chapter

Creating a Basic Application	149
Connecting to Parallels Service and Logging In	152
Host Operations.....	155
Virtual Machine Operations.....	158
Remote Desktop Access.....	178

Creating a Basic Application

The following steps are required in any client application using the Parallels Python API:

- 1 Import the `prlsdkapi` package. This is the main Parallels Python API package containing the majority of the classes and additional modules.
- 2 Initialize the API. This step must be performed correctly for the type of the Parallels virtualization product that you are going to be connecting to. To initialize for Parallels Server-based products (Parallels Server, Parallels Server Bare Metal) use the `prlsdkapi.init_server_sdk()` function. For Parallels Desktop use the `init_desktop_sdk()` function. For Parallels Workstation use `init_workstation_sdk()`. For Parallels Desktop for Windows and Linux use `init_desktop_wl_sdk()`. For Parallels Player use `init_player_sdk()`. To verify that the API was initialized successfully, you may use the `is_sdk_initialized()` function.
- 3 Create an instance of the `prlsdkapi.Server` class. The `Server` class provides methods for logging in and for obtaining other object references (a virtual machine object in particular).
- 4 Perform the login operation using the `Server.login()` or `Server.login_local()` method. Use the proper method depending on the Parallels product type and your application specifics. Parallels Server allows local and remote logins, so both methods can be used. All other products accept local logins only, so only the `Server.login_local()` method is applicable. We will discuss login operations in greater detail in the [Connecting and Logging In to a Host](#) section (p. 152).

To exit gracefully, the client application should perform the following steps:

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

Example

The following is a complete Python program that illustrates an implementation of the steps above. The program assumes that it will be run on a client machine connecting to a remote Parallels Server. If you are using any other Parallels virtualization product, uncomment the appropriate lines in the `main()` function where the library is initialized and where the `login_server()` function is called and comment out the Parallels Server-specific lines.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
# (c) Parallels Software International, Inc. 2005-2009
#
#
# Example of prlsdkapi usage.
#
# Import the main Parallels Python API package.
import prlsdkapi
# Import some of the standard Python modules.
```

```

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

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

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

"""
Parallels Service login.

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

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

    # Obtain a LoginResponse object contained in the Result object.
    # LoginResponse contains the results of the login operation.
    login_response = result.get_param()

    # Get the Parallels virtualization product version number.
    product_version = login_response.get_product_version()

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

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

    print ""
    print "Login successful"
    print ""
    print "Parallels product version: " + product_version
    print "Host OS versions:          " + host_os_version
    print "Host UUID:                    " + host_uuid
    print ""

#####
#####

def main():

```

```
# Initialize the library for Parallels Server.
prlsdkapi.init_server_sdk()

# Create a Server object and log in to a remote Parallels Server.
server = prlsdkapi.Server()
login_server(server, "10.30.18.99", "root", "secret",
consts.PSL_NORMAL_SECURITY);

# Initialize the library for Parallels Desktop.
# prlsdkapi.init_desktop_sdk()

# Initialize the library for Parallels Desktop for Windows and Linux.
# prlsdkapi.init_desktop_wl_sdk()

# Initialize the library for Parallels Workstation.
# prlsdkapi.init_workstation_sdk()

# Initialize the library for Parallels Player.
# prlsdkapi.init_player_sdk()

# Create a Server object and log in to a local
# Parallels Desktop/Workstation/Desktop for Windows and Linux/Player.
#
# server = prlsdkapi.Server()
# login_server(server, "localhost", "", "", consts.PSL_NORMAL_SECURITY);

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

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

Connecting to Parallels Service and Logging In

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

Parallels Server and Parallels Server Bare Metal accept both local and remote connections. This means that your client program can run anywhere on the network and connect to the Parallels Server remotely. Parallels Server allows multiple connections. If running a client program locally, you have an option to login as the current user or as a specified user. If a program is running on a remote client, you always have to specify the user information.

All other Parallels virtualization products accept local connections only. This means that you can only run your client program on the same computer that hosts the Parallels Virtualization Service. Multiple connections are not allowed, so the only option available is to connect as the current user.

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

The `login_local` method is used to establish a *local connection* as a *current user*. It can be used with Parallels Server and other Parallels products.

The `login()` method is used to establish a local or a remote connection as a *specified user*. It can be used with Parallels Server and Parallels Server Bare Metal only.

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

`prlsdkapi.Server.login_local`

Parameter	Type	Description
<code>sPrevSessionUui d</code>	string	[optional] Previous session ID. This parameter can be used in recovering from a lost connection. The ID will be used to restore references to the asynchronous jobs that were started in the previous session and are still running on the server. If you are not restoring a connection, omit this parameter or pass an empty string. The default value is empty string.
<code>port</code>	integer	[optional] Port number at which Parallels Service is listening for incoming requests. To use the default port number, pass 0. If the default port number was changed by the administrator of your system, specify the correct value. The default value is 0.

security_level	integer	<p>[optional] Communication security level to use for the session. The value must be specified using one of the constants with the PSL_ prefix. The following options are currently available (for possible changes, consult the Parallels Python API Reference guide):</p> <p>PSL_HIGH_SECURITY - using SSL. PSL_LOW_SECURITY - no encryption. PSL_NORMAL_SECURITY - mixed.</p> <p>Parallels Service configuration have a setting specifying the minimum security level (the setting can be modified). You must specify here an equal or a higher level to establish a connection. To find out the minimum level, use the <code>get_min_security_level</code> method of the <code>prlsdkapi.DispConfig</code> class.</p> <p>The default value is PSL_HIGH_SECURITY.</p>
----------------	---------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

prlsdkapi.Server.login

Parameter	Type	Description
host	string	The IP address or name of the host.
user	string	User name.
passwd	string	User password.
sPrevSessionUui d	string	<p>[optional] Previous session ID. This parameter can be used in recovering from a lost connection. The ID will be used to restore references to asynchronous jobs that were started in the previous session and are still running on the server. If you are not restoring a connection, omit this parameter or pass an empty string value.</p> <p>The default value is empty string.</p>
port_cmd	integer	<p>[optional] Port number on which Parallels Service is listening for incoming requests. To use the default port number, pass 0. If the default port number was changed by the administrator, specify the correct value.</p> <p>The default value is 0.</p>
timeout	integer	<p>[optional] Timeout value in milliseconds. The operation will be automatically interrupted if a connection is not established within this timeframe. Specify 0 (zero) for infinite timeout.</p> <p>The default value is 0.</p>
security_level	integer	<p>[optional] Communication security level to use for the session. The value must be specified using one of the constants with the PSL_ prefix. The following options are currently available (for possible changes, consult the Parallels Python API Reference guide):</p>

		<p>PSL_HIGH_SECURITY - using SSL. PSL_LOW_SECURITY - no encryption. PSL_NORMAL_SECURITY - mixed.</p> <p>Parallels Service configuration have a setting specifying the minimum security level (the setting can be modified). You must specify here an equal or a higher level to establish a connection. To find out the minimum level, use the <code>get_min_security_level</code> method of the <code>prlsdkapi.DispConfig</code> class.</p> <p>The default value is PSL_HIGH_SECURITY.</p>
--	--	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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

Host Operations

Retrieving Host Configuration Info

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

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

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

The information is obtained using the `get_srv_config` method of the `prlsdkapi.Server` class. This is an asynchronous method, so the information is returned via the `Job` and `Result` objects (see the [Asynchronous Methods](#) section (p. 145) for more information). The name of the class containing the host configuration information is `prlsdkapi.ServerConfig`.

Example

```
"""
    This function demonstrates how to obtain the
    host computer configuration information.
    @param server: An instance of prlsdkapi.Server identifying the
                   Parallels Service.
"""
def get_host_configuration_info(server):
    print ""
    print "Host Configuration Information"
    print "======"

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

    srv_config = result.get_param()

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

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

    # Get the network adapter info.
    # The type of the netd object is prlsdkapi.SrvCfgNet.
    print ""
```

```
print "Network adapters"
print ""
print "No.  Type                Status  System Index"
print "-----"

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

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

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

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

Managing Parallels Service Preferences

Parallels Service preferences is a set of configuration parameters that control its behavior. The most important parameters are:

- Parallels Service memory limits.
- Virtual machine memory limits and recommended values.
- Virtual network adapter information.
- Default virtual machine directory (the directory where all new virtual machines are created by default).
- Minimum allowed communication security level.

To obtain the preferences information use the `prlsdkapi.Server.get_common_prefs` method. This is an asynchronous method. The preferences information is obtained from the `Result` object and is returned as an instance of the `prlsdkapi.DispConfig` class. Once you obtain the instance, you can use its methods to view and modify individual settings. Parallels Service preferences modifications are performed in a transactional manner. First you have to invoke the `Server.common_prefs_begin_edit` method to mark the beginning of the modification operation (i.e "begin a transaction"). This will timestamp the operation to prevent a conflict with other clients trying to make modifications to the same data at the same time. When you are done making the changes, invoke the `Server.common_prefs_commit` method to commit the changes to the Parallels Service. If there's a conflict, the method will throw an exception and your commit will be aborted. In such a case, you will have to re-evaluate the data and repeat the steps from the beginning. On success, the changes will be saved on the Parallels Service side.

Example

```
"""
    This function shows how to view and modify Parallels Service preferences.
    @param server: An instance of prlsdkapi.Server identifying the
                   Parallels Service.
"""
def srv_preferences_management(server):

    print ""
    print "Parallels Service Preferences Management"
    print "-----"

    # The preferences info is obtained as a prlsdkapi.DispConfig object.
    try:
        result = server.get_common_prefs().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    disp_config = result.get_param()

    # Obtain the default virtual machine directory.
    print "Default virtual machine directory: " + \
          disp_config.get_default_vm_dir()

    # The minimum allowed security level.
    # This setting ensures that the communication security level
    # specified at login satisfies the necessary requirements.
    security_level = disp_config.get_min_security_level()

    if security_level == consts.PSL_LOW_SECURITY:
```

```
        security_level = "Low"
    elif security_level == consts.PSL_NORMAL_SECURITY:
        security_level = "Normal"
    elif security_level == consts.PSL_HIGH_SECURITY:
        security_level = "High"

    print "Currently set minimum security level: " + security_level

    # Modify the minimum security level.
    # First, mark the beginning of the editing operation (required step).
    try:
        server.common_prefs_begin_edit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Set the new security level value.
    new_level = consts.PSL_HIGH_SECURITY
    disp_config.set_min_security_level(new_level)

    # Commit the changes.
    try:
        server.common_prefs_commit(disp_config).wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    print "Minimum security was set to High."
```

Virtual Machine Operations

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

In order to perform operations on a virtual machine, an instance of the `prlsdkapi.Vm` class identifying the virtual machine must be obtained. Once you have the instance, you can use its methods to perform some of the virtual machine management operations (start, stop, pause, create snapshot, clone, and many others) and to obtain other objects that allow to perform additional virtual machine management functions, such as modifying the virtual machine configuration. This section begins with a demonstration of how to obtain the virtual machine list from the Parallels Service (including obtaining a `Vm` object identifying an individual machine) and then describes how to perform various virtual machine management tasks.

Obtaining the Virtual Machine List

Before a virtual machine can be powered on, it must be registered with the Parallels Service. All new virtual machines created with Parallels management tools are registered by default. Some virtual machines can exist on the host without being registered. This can happen if the virtual machine files were copied from another location or computer or if the virtual machine was intentionally removed from the Parallels Service registry. The list of the machines that are registered with Parallels Service can be retrieved using the `get_vm_list` method of the `prlsdkapi.Server` class. The method obtains a `prlsdkapi.Result` object containing a list of `prlsdkapi.Vm` objects, each of which can be used to obtain a complete information about an individual virtual machine. Once we obtain the `Result` object, we will have to extract individual `Vm` objects from it using `Result.get_params_count` and `Result.get_param` methods. The first method returns the `Vm` object count. The second method returns a `Vm` object specified by its index inside the container.

The following example shows how obtain the virtual machine list. The sample functions accepts a `prlsdkapi.Server` object. Before passing it to the function, the object must be properly created, the API library must be initialized, and a connection with the Parallels Service must be established. Please see the [Creating a Basic Application](#) section (p. 149) for more information and code samples.

Example

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

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

    print "Virtual Machine" + " " + "State"
    print "-----"

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

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

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

        # Obtain the VmInfo object containing the
        # virtual machine state info.
        # The object is obtained from the Result object returned by
        # the vm.get_state() method.
        try:
            state_result = vm.get_state().wait()
```

```
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e
    return

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

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

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

# Print the virtual machine name and status on the screen.
vm_name = vm_name + " "
print vm_name[:25] + "\t" + state_desc

print "-----"
```

Searching for a Virtual Machine

The example provided in this section does not really show anything new but it can be useful when testing the sample code provided in later sections. The sample function below accepts a virtual machine name as a parameter (the name can be partial) and searches for it in the virtual machine list retrieved from the host. If it finds it, it returns the `prlsdkapi.Vm` object identifying the virtual machine to the caller. The function uses the same approach that was used in the Obtaining the Virtual Machine List section (p. 159). It obtains the list of virtual machine from the Parallels Service, then iterates through it comparing a virtual machine name to the specified name.

Example

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

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

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

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

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

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

vm = search_vm(server, search_name)

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

Performing Power Operations

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

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

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

Examples

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

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

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

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

# Restart the virtual machine.
try:
    vm.restart().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e

# Reset the virtual machine. This operation is an equivalent of
# Stop and Start performed in succession.
# The stop operation will NOT use ACPI, so the entire reset
# operation will resemble the "Reset" button pressed on a physical box.
try:
    vm.reset().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e
```

Creating a New Virtual Machine

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

In this section, we will discuss how to create a typical virtual machine for a particular OS type using a sample configuration. By using this approach, you can easily create a virtual machine without knowing all of the little 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 `prlsdkapi.SrvConfig` object containing the host machine configuration information. This information is needed to configure the new virtual machine, so it will run properly on the given host.
- 2 Obtain a new `prlsdkapi.Vm` object that will identify the new virtual machine. This must be performed using the `prlsdkapi.Server.create_vm` method.
- 3 Obtain an instance of the `prlsdkapi.VmConfig` object that will contain the new virtual machine configuration information. This step must be performed using the `prlsdkapi.Vm.get_config` method.
- 4 Set the default configuration based on the version of the OS that you will later install in the machine. This step is performed using the `prlsdkapi.VmConfig.set_default_config()` method. You supply the version of the target OS and the method will generate the appropriate configuration parameters automatically. The OS version is specified using predefined constants that have the `PVS_GUEST_VER_` prefix in their names.
- 5 Choose a name for the virtual machine and set it using the `VmConfig.set_name` method.
- 6 Modify the default configuration parameters if needed. For example, you may want to modify the hard disk image type and size, the amount of memory available to the machine, and the networking options. When modifying a device, an object identifying it must first be obtained and then its methods and properties can be used to make the modifications. The code sample provided in this section shows how to modify some of the default configuration values.
- 7 Create and register the new machine using the `Vm.reg()` method. This step will create the necessary virtual machine files on the host and register the machine with Parallels Service. The virtual machine directory will have the same name as the name you've chosen for your virtual machine and will be created in the default location for this Parallels Service. You may specify a different virtual machine directory name and path if you wish.

Sample

```
"""
    Create a new virtual machine.
"""
def create_vm(server):

    # Obtain the prlsdkapi.ServerConfig object.
    # The object contains the host machine configuration
```

```
# information.
try:
    result = server.get_srv_config().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e

srv_config = result.get_param()

# Obtain a new prlsdkapi.Vm object.
vm = server.create_vm()

# Obtain a prlsdkapi.VmConfig object.
# The new virtual machine configuration will be performed
# using this object. At this time the object will be empty.
vm_config = vm.get_config()

# Use the default configuration.
# Parameters of the set_default_config method:
# param_1: The host machine configuration object.
# param_2: Target OS type and version.
# param_3: Specifies to create the virtual machine devices using
#          default values (the settings can be modified
#          later if needed).
vm_config.set_default_config(srv_config, \
                             consts.PVS_GUEST_VER_WIN_XP, True)

# Set the virtual machine name and description.
vm_config.set_name("My New XP machine")
vm_config.set_description("Parallels Python API sample")

# Modify the default RAM size and HDD size.
# These two steps are optional. If you omit them, the
# default values will be used.
vm_config.set_ram_size(256)
# Set HDD size to 10 gig.
# The get_device method obtains a prlsdkapi.VmHardDisk object.
# The index 0 is used because the default configuration has a
# single hard disk.
dev_hdd = vm.get_hard_disk(0)
dev_hdd.set_disk_size(10000)

# Register the virtual machine with the Parallels Service.
# The first parameter specifies to create the machine in the
# default directory on the host computer.
# The second parameter specifies that non-interactive mode
# should be used.
print "Creating a virtual machine..."
try:
    vm.reg("", True).wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e
    return

print "Virtual machine was created successfully."
```

Obtaining Virtual Machine Configuration Data

The virtual machine configuration information includes the machine name, guest operating system type and version, RAM size, disk drive and network adapter information, and other settings. To obtain this information, a `prlsdkapi.VmConfig` object must be obtained from the `prlsdkapi.Vm` object (the object that identifies the virtual machine). The object methods can then be used to extract the data. The examples in this section show how to obtain the most commonly used configuration data. We will talk about modifying configuration parameters in the [Modifying Virtual Machine Configuration](#) section (p. 168).

All sample functions below accept a single parameter -- an instance of `prlsdkapi.Vm` class. To obtain the object, you can use the helper function `search_vm()` that we've created in the [Searching for a Virtual Machine](#) section (p. 161).

Obtaining the RAM size

```
def get_vm_ram_size(vm):

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

    # Get the virtual machine RAM size.
    ram_size = vm_config.get_ram_size()
    print "RAM size: " + str(ram_size)
```

Obtaining the OS type and version

```
def get_vm_os_info(vm):

    print ""

    # Virtual machine name.
    print "Virtual machine name: " + vm.get_name()

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

    # Obtain the guest OS type and version.
    # OS types are defined as PVS_GUEST_TYPE_xxx constants.
    # For the complete list, see the documentation for
    # the prlsdkapi.prlsdk.consts module or
    # the Parallels Python API Reference guide.
    os_type = vm_config.get_os_type()
    if os_type == consts.PVS_GUEST_TYPE_WINDOWS:
        osType = "Windows"
    elif os_type == consts.PVS_GUEST_TYPE_LINUX:
        osType = "Linux"
    else:
        osType = "Other type (" + str(os_type) + ")"

    # OS versions are defined as PVS_GUEST_VER_xxx constants.
    os_version = vm_config.get_os_version()
    if os_version == consts.PVS_GUEST_VER_WIN_XP:
        osVersion = "XP"
    elif os_version == consts.PVS_GUEST_VER_WIN_2003:
        osVersion = "2003"
    elif os_version == consts.PVS_GUEST_VER_LIN_FEDORA_5:
        osVersion = "Fedora 5"
    else:
        osVersion = "Other version (" + str(os_version) + ")"
```

```
print "Guest OS: " + osType + " " + osVersionRAM size
```

Obtaining optical disk drive information

```
def get_optical_drive_info(vm):

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

    print ""
    print "Optical Drives:"
    print "-----"

    # Iterate through the existing optical drive devices.
    count = vm_config.get_optical_disks_count()
    for i in range(count):
        print ""
        print "Drive " + str(i)

        # Obtain an instance of VmDevice containing the optical drive info.
        device = vm_config.get_optical_disk(i)

        # Get the device emulation type.
        # In case of optical disks, this value specifies whether the virtual
device
        # is using a real disk drive or an image file.
        emulated_type = device.get_emulated_type()

        if emulated_type == consts.PDT_USE_REAL_DEVICE:
            print "Uses physical device"
        elif emulated_type == consts.PDT_USE_IMAGE_FILE:
            print "Uses image file " + "'" + device.get_image_path() + "'"
        else:
            print "Unknown emulation type"

        if device.is_enabled():
            print "Enabled"
        else:
            print "Disabled"

        if device.is_connected():
            print "Connected"
        else:
            print "Disconnected"
```

Obtaining hard disk information

```
def get_hdd_info(vm):

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

    print ""
    print "Virtual Hard Disks:"
    print "-----"

    count = vm_config.get_hard_disks_count()
    for i in range(count):
        print ""
        print "Disk " + str(i)

        hdd = vm_config.get_hard_disk(i)
        emulated_type = hdd.get_emulated_type()

        if emulated_type == consts.PDT_USE_REAL_DEVICE:
```

```

        print "Uses Boot Camp: Disk " + hdd.get_friendly_name()
    elif emulated_type == consts.PDT_USE_IMAGE_FILE:
        print "Uses image file " + "'" + hdd.get_image_path() + "'"

    if hdd.get_disk_type() == consts.PHD_EXPANDING_HARD_DISK:
        print "Expanding disk"
    elif hdd.get_disk_type() == consts.PHD_PLAIN_HARD_DISK:
        print "Plain disk"

    print "Disk size:" + str(hdd.get_disk_size()) + " Mbyte"
    print "Size on physical disk: " + str(hdd.get_size_on_disk()) + "
Mbyte"

```

Obtaining network adapter information

```

def get_net_adapter_info(vm):

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

    # Obtain the network interface info.
    # The vm.net_adapters sequence contains objects of type VmNetDev.
    print ""
    print "Network Adapters:"

    count = vm_config.get_net_adapters_count()
    for i in range(count):
        print ""
        print "Adapter " + str(i)

        net_adapter = vm_config.get_net_adapter(i)

        emulated_type = net_adapter.get_emulated_type()

        if emulated_type == consts.PNA_HOST_ONLY:
            print "Uses host-only networking"
        elif emulated_type == consts.PNA_SHARED:
            print "Uses shared networking"
        elif emulated_type == consts.PNA_BRIDGED_ETHERNET:
            print "Uses bridged ethernet (bound to " +
net_adapter.get_bound_adapter_name() + ")"

        print "MAC address " + str(net_adapter.get_mac_address())

```

Modifying Virtual Machine Configuration

`Vm.begin_edit` and `Vm.commit` Methods

All virtual machine configuration changes must begin with the `prlsdkapi.Vm.begin_edit` and end with the `prlsdkapi.Vm.commit` call. These two methods are used to detect collisions with other clients trying to modify the configuration settings of the same virtual machine at the same time.

The `Vm.begin_edit` method timestamps the beginning of the editing operation. It does not lock the machine, so other clients can still make changes to the same virtual machine. The method will also automatically update your local virtual machine object with the current virtual machine configuration information. This is done in order to ensure that the local object contains the changes that might have happened since you obtained the virtual machine object. When you are done making the changes, you must invoke the `Vm.commit` method. The first thing that the method will do is verify that the virtual machine configuration has not been modified by other client(s) since you began making your changes. If a collision is detected, your changes will be rejected and `Vm.commit` will throw an exception. In such a case, you will have to reapply the changes. In order to do that, you will have to get the latest configuration using the `Vm.refresh_config` method and re-evaluate it. Please note that `Vm.refresh_config` method will update the configuration data in your local virtual machine object and will overwrite all existing data, including the changes that you've made so far. Furthermore, the `Vm.begin_edit` method will also overwrite all existing data (see above). If you don't want to lose your data, save it locally before invoking any of the two methods.

Name, Description, Boot Options, RAM size

The virtual machine name, description, and RAM size modifications are simple. They are performed by invoking a corresponding method of a `Vm` object. To modify the boot options (boot device priority), obtain an instance of the `prlsdkapi.BootDevice` class representing each device. This step is performed using the `VmConfig.get_boot_dev_count` method to determine the total number of boot devices, then iterating through the list and obtaining a `BootDevice` object using the `VmConfig.get_boot_dev` method. To place a device at the specified position in the boot device priority list, use the `BootDevice.set_sequence_index` method passing a value of 0 to the first device, 1 to the second device, and so forth. If you have more than one instance of a particular device type in the boot list (i.e. more than one CD/DVD drive), you will have to set a sequence index for each instance individually. An instance is identified by an index that can be obtained using the `VmBootDev.get_index` method. A device in the boot priority list can be enabled or disabled using the `BootDevice.set_in_use` method. Disabling the device does not remove it from the boot device list. To remove a device from the list, use the `BootDevice.remove` method.

Example

```
"""
    Modify the virtual machine name, RAM size, and boot options.
"""
def vm_edit(vm):

    # Begin the virtual machine editing operation.
    try:
        vm.begin_edit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

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

    vm.set_name(vm.get_name() + "_modified")
    vm.set_ram_size(256)
    vm.set_description("SDK Test Machine")

    # Modify boot device priority using the following order:.
    # CD > HDD > Network > FDD.
    # Remove all other devices from the boot priority list (if any).
    count = vm_config.get_boot_dev_count()
    for i in range(count):

        # Obtain an instance of the prlsdkapi.BootDevice class
        # containing the boot device information.
        boot_dev = vm_config.get_boot_dev(i)

        # Enable the device.
        boot_dev.set_in_use(True)

        # Set the device sequence index.
        dev_type = boot_dev.get_type()

        if dev_type == consts.PDE_OPTICAL_DISK:
            boot_dev.set_sequence_index(0)
        elif dev_type == consts.PDE_HARD_DISK:
            boot_dev.set_sequence_index(1)
        elif dev_type == consts.PDE_GENERIC_NETWORK_ADAPTER:
```

```
        boot_dev.set_sequence_index(2)
    elif dev_type == consts.PDE_FLOPPY_DISK:
        boot_dev.set_sequence_index(3)
    else:
        boot_dev.remove()

# Commit the changes.
try:
    vm.commit().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e
    return
```

Adding a Hard Disk Drive

The following sample function demonstrates how to create a new image file and to add a new virtual hard disk to a virtual machine. The steps are:

- 1 Mark the beginning of the editing operation.
- 2 Create a device object representing the new disk.
- 3 Set the device properties, including emulation type (image file or real device), disk type (expanding or fixed), disk size, and disk name.
- 4 Create the image file.
- 5 Commit the changes.

Example

```
"""
    Add a new virtual hard disk to the virtual machine.
"""
def add_hdd(vm):

    # Begin the virtual machine configuration editing.
    try:
        vm.begin_edit().wait
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

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

    # Create an instance of the prlsdkapi.VmHardDisk class.
    hdd_dev = vm_config.create_vm_dev(consts.PDE_HARD_DISK)

    # Populate the object.
    # Set emulated type (image file or real device).
    hdd_dev.set_emulated_type(consts.PDT_USE_IMAGE_FILE)

    # Set disk type (expanding or fixed)
    hdd_dev.set_disk_type(consts.PHD_EXPANDING_HARD_DISK)

    # Set disk size to 20 Gig.
    hdd_dev.set_disk_size(20000)

    # Choose and set a name for the new image file.
    # Both the friendly_name and the sys_name properties must be
    # populated and must contain the same value.
    # The new image file will be created in
    # the virtual machine directory.
    # To create the file in a different directory,
    # the name must contain the full directory path and
    # the hard disk name.
    hdd_name = vm_config.get_name() + "_hdd_sample.hdd"
    hdd_dev.set_friendly_name(hdd_name)
    hdd_dev.set_sys_name(hdd_name)

    # Enable the disk.
    hdd_dev.set_enabled(True)

    # Create the image file.
    # First parameter - Overwrite the image file if it exists.
    # Second parameter - Use non-interactive mode.
```

```
try:
    hdd_dev.create_image(True, True)
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e
    return

# Commit the changes.
try:
    vm.commit().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e
    return

print("New hard disk was created successfully.")
```

Adding a Network Adapter

To add a new virtual network adapter to a virtual machine, the following steps must be taken:

- 1 Mark the beginning of the editing operation.
- 2 Create a device object representing the new adapter.
- 3 Set the emulation type (host-only, shared, or bridged).
- 4 If creating a bridged adapter, select the host adapter to bind the new adapter to.
- 5 Commit the changes.

Example

```
"""
Add a network adapter to the virtual machine.
@param vm: An instance of prlsdkapi.Vm class identifying the
          virtual machine.
@param networking_type: Host-only/shared/bridged. Use one of the
                       consts.PNA_XXX constants.
@param bound_default: Used with bridged networking only.
                       Specify True to bound a new adapter to the
                       default physical adapter. If False is passed,
                       the adapter will be bound to a specific physical
                       adapter (in this example, the adapter is
                       chosen randomly).
"""
def add_net_adapter(server, vm, networking_type, bound_default = True):

    # Begin the virtual machine editing operation.
    try:
        vm.begin_edit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

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

    # Create an instance of the prlsdkapi.VmNet class.
    net_adapter = vm_config.create_vm_dev(consts.PDE_GENERIC_NETWORK_ADAPTER)

    # Set the emulation type to the specified value.
    net_adapter.set_emulated_type(networking_type)

    # For bridged networking mode, we'll have to bind the
    # new adapter to a network adapter on the host machine.
    if networking_type == consts.PNA_BRIDGED_ETHERNET:

        # To use the default adapter, simply set the
        # adapter index to -1.
        if bound_default == True:
            net_adapter.set_bound_adapter_index(-1)
        else:
            # To use a specific adapter, first obtain the
            # list of the adapters from the host machine.

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

```
    srv_config = result.get_param()

    # Iterate through the list of the host network adapters.
    # In this example, we are simply selecting the first
    # adapter in the list and binding the virtual adapter to it.
    # The adapter is identified by its name.
    for i in range(srv_config.get_net_adapters_count()):
        hw_net_adapter = srv_config.get_net_adapter(i)
        hw_net_adapter_name = hw_net_adapter.get_name()
        net_adapter.set_bound_adapter_name(hw_net_adapter_name)
        exit

    # Connect and enable the new virtual adapter.
    net_adapter.set_connected(True)
    net_adapter.set_enabled(True)

    # Commit the changes.
    try:
        vm.commit().wait()
    except prl_sdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    print("Virtual network adapter created successfully")
```

Adding an Existing Virtual Machine

A host may have virtual machines that are not registered with the Parallels Service. This can happen if a virtual machine was previously removed from the Parallels Service registry or if the virtual machine files were manually copied from a different location. If you know the location of such a virtual machine, you can easily register it with the Parallels Service.

Note: When adding an existing virtual machine, the MAC addresses of its virtual network adapters are kept unchanged. If the machine is a copy of another virtual machine, then you should set new MAC addresses for its network adapters after you register it. The example below demonstrates how this can be accomplished.

Example:

The following sample function demonstrates how to register an existing virtual machine. The function takes a `Server` object identifying the Parallels Service and a string specifying the name and path of the virtual machine directory (on Mac OS X it is the name of a bundle). It registers the virtual machine and then modifies the MAC address of every virtual network adapter installed in it.

```
"""
    Add an existing virtual machine.
    @param path: Name and path of the virtual machine directory or bundle.
                  The name normally has the .PVM extension.
"""
def register_vm(server, path):

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

    vm = result.get_param()
    vm_config = vm.get_config()
    print vm_config.get_name() + " was registered."

    # Generate a new MAC addresses for all virtual network adapters.
    # This should be done when a virtual machine was copied from another host.
    try:
        vm.begin_edit()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    # Iterate through the network adapter list and
    # generate a new MAC address.
    # The get_net_adapter(i) method returns an instance of the VmNet class.
    for i in range(vm_config.get_net_adapters_count()):
        net_adapter = vm_config.get_net_adapter(i)
        net_adapter.generate_mac_addr()

    # Commit the changes.
    try:
        vm.commit().wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return
```

Removing an Existing Virtual Machine

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

- Un-register the virtual machine without deleting its files. You can re-register the virtual machine later if needed.
- Delete the virtual machine from the host completely. The virtual machine files will be permanently deleted and cannot be recovered if this option is used.

Example

The following sample function illustrates how to implement both options. The function takes a `Vm` object identifying the virtual machine and a boolean value indicating whether the virtual machine files should be deleted from the host computer.

```
"""
    Remove an existing virtual machine.
    @param vm: An instance of prlsdkapi.Vm class identifying
               the virtual machine.
    @param delete: A boolean value indicating whether the
                  virtual machine files should be permanently deleted
                  from the host.
"""
def remove_vm(vm, delete):

    if delete == False:
        # Unregister the virtual machine but don't delete its files.
        try:
            vm.unreg()
        except prlsdkapi.PrlSDKError, e:
            print "Error: %s" % e
            return
    else:
        # Unregister the machine and delete its files from the hard drive.
        try:
            vm.delete()
        except prlsdkapi.PrlSDKError, e:
            print "Error: %s" % e
            return

    print vm.get_config().get_name() + " has been removed."
```

Cloning a Virtual Machine

A new virtual machine can 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 Service. The cloning operation is performed using the `prlsdkapi.Vm.clone` method. The following parameters must be specified when cloning a virtual machine:

- 1 A unique name for the new virtual machine (the new name is NOT generated automatically).
- 2 The name of the directory where the new virtual machine files should be created or an empty string to create the files in the default directory.
- 3 A boolean value specifying whether to create a regular virtual machine or a virtual machine template. Virtual machine templates are used to create other virtual machines from them. You cannot run a template.

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

Sample

```
"""
Clone a virtual machine.
@param vm: An instance of the prlsdkapi.Vm class identifying
          the source virtual machine.
"""
def clone_vm(vm):
    try:
        new_name = "Clone_2 of " + vm.get_config().get_name()
        print "Cloning is in progress..."
        # Second parameter - create a new machine in the
        # default directory.
        # Third parameter - create a virtual machine (not a template).
        vm.clone(new_name, "", False).wait()
    except prlsdkapi.PrlSDKError, e:
        print "Error: %s" % e
        return

    print "Cloning was successful. New virtual machine name: " + new_name
```

Remote Desktop Access

Remote Desktop Access is a functionality that allows to remotely capture screenshots of a virtual machine desktop and to send keyboard and mouse commands to it. With this functionality, you can programmatically connect to a remote virtual machine and run its applications as if you were sitting at the virtual machine's console. Typical uses of the Remote Desktop Access functionality include creating automation scripts for unattended operating system and other software installations, implementing automated test systems, or automating any other routine activity, which would otherwise require you to physically look at the virtual machine screen and operate with its keyboard and mouse.

The functionality is supported in both Parallels C and Python APIs. C API provides additional functions that can be used to create remote desktop applications with graphical user interfaces. Python API contains a simplified version of the C API functionality and is best suited for writing automation scripts.

The Remote Desktop Access functionality is provided by the `prlsdkapi.VmIO` class. There are three groups of methods in the class:

- **Primary display capture.** These functions allow to capture the primary display of the remote virtual machine. In scripts, you can take a snapshot of a screen of interest in advance and save it to a file. At runtime, you capture virtual machine screens after every interaction with it and perform a bit-by-bit comparison of the saved screen and a snapshot that you take each time. If the comparison operation determines that the screen currently displayed on the virtual machine desktop is the screen of interest, you can interact with user interface controls that it contains (keystrokes on text-based screens; visual controls on GUI screens) by sending mouse or keyboard commands to the virtual machine. The assumption here is that the desktop background is static, individual windows always open at the same coordinates, have a fixed size, and contain the same number of controls and data in the same exact default state.
- **Mouse control.** These functions provide mouse control in a virtual machine. You can change the position of the mouse pointer, press and release mouse buttons, and use a scroll wheel.
- **Keyboard control.** These functions send a key/action code combination to the virtual machine. In scripts, you can use these functions to interact with controls on a window opened inside a virtual machine (pressing buttons, selecting options, etc.). In a typical GUI application, essential visual controls usually have keyboard shortcuts (accelerator keys) assigned to them. For example, to click a button, you can send an accelerator key combination to the virtual machine; to select/deselect a check-box you similarly send a keyboard shortcut assigned to it, and so forth. If a control doesn't have a shortcut, then you will have to use mouse control functions to position a mouse pointer over it and clicking the mouse button (you will have to determine the control's coordinates in advance to properly position the mouse pointer).

The use of this functionality is not limited to the tasks described above. You can use it for anything that requires taking screenshots of a virtual machine desktop and controlling its keyboard and mouse input.

Creating a Simple OS Installation Program

In this section, we will write a simple program that can be used to automatically install a hypothetical operating system inside a brand new virtual machine.

Step 1 - Preparation

First, we have to capture all screens that the OS installation wizard displays to the user.

- 1 Create a blank virtual machine, mount a CD drive in it, insert the OS installation disk (or mount an ISO image of the disk), and start the virtual machine.
- 2 Using the Parallels API, programmatically connect to the virtual machine and begin a Remote Desktop Access session with it.
- 3 At this point, the first OS installation screen should be displayed, waiting for user interaction. Using the Remote Desktop Access API, capture this screen to a file on the client machine.
- 4 Go back to the virtual machine console and manually make the appropriate selections on it (for example, press the **Continue** button). Write down the accelerator key assigned to the control (**Alt-N** for instance, or **Enter** if this is a default button, or the appropriate keystroke if the screen is in a text mode).
- 5 When the next screen opens, capture it to a file using the API the same way you captured the first screen. Write down the controls that advance you to the next installation screen the same way we did in the previous step.
- 6 Repeat for all of the installation screens until the operating system is fully installed, or until a desired point in the installation process is reached.
- 7 In the end, you should have a collection of files containing images of installation screens and instructions for each screens describing the actions that should be taken on each one of them.

Step 2 - Writing the automated OS installation program

Now that we have screenshots and interaction instructions, we can write the program that will automatically install the OS on any blank virtual machine.

- 1 Every remote desktop access session must begin with the `VmIO.connect_to_vm` method invocation and end with `VmIO.disconnect_from_vm`. These step are necessary to properly initialize/deinitialize the remote desktop access library and protocol.
- 2 Capture the current virtual machine desktop screen to a file on the client machine.
- 3 Make a bit-by-bit comparison of the screen that you've just captured to every file that you saved in **Step 1 - Preparation** (above). Once you find the matching screen, continue to the next step (note: for bit-by-bit comparison to work, a lossless compression or no compression must be used when saving captured screen data).
- 4 Read the interaction instructions for the screen you've just found and send the appropriate keyboard (or mouse if necessary) commands to the virtual machine. For example, if the instruction says "press Enter on the keyboard", send the "enter key pressed" command (the actual key and mouse command codes are explained in the [Parallels Python API Reference guide](#) and some examples are provided below).

- 5 Wait a while for the next screen on the virtual machine desktop to open and repeat the capture and user interaction procedure the same exact way as explained in the step above.
- 6 Repeat until all saved screens are found and processed. Your operating system is installed.

Bit-by-bit comparison notes

In theory, the screen comparison procedure described above (comparing two full screens) should work. In reality, it is virtually impossible to achieve a state when an individual screen remains absolutely static. Such things as blinking cursor, messages from the program vendor, and other animations will make each capture different from another. Therefore, instead of comparing an entire screen, the following approach can be used:

- 1 In the **Preparation** step described above, capture an entire virtual machine desktop screen and then select a rectangular region on it that is guaranteed to be absolutely static and is unique enough to be used to identify the screen. This can be the screen name (such as "Step 1" or similar), a unique static picture or text displayed on it, and so forth. Copy the region from the image using an image editor of your choice and save it to a file.
- 2 When determining the identity of a screen captured at runtime, start at the beginning of the full screen image and see if a screen region that you saved earlier matches the region on the screen at that position. Since you know that the region never changes, you can safely use a bit-bit-comparison. If the two regions don't match, move one pixel forward and compare again. Repeat until the match is found or until the end of file is reached.

Example

The following sample program illustrates the implementation of the steps above. Please note that this is not a complete working program. The program does not include the implementation of the algorithm described in the **Bit-by-bit comparison** subsection (above). An implementation of the algorithm depends on the image format used and in any case should be simple and straightforward. Some of the steps in the program are simplified, specifically the `keys` array contains only three keys that are used to demonstrate how to send the key commands to the virtual machine. The main steps concerning the API usage are included and can be used as a starting point for your own implementation.

```
import sys
import prlsdkapi

consts = prlsdkapi.prlsdk.consts

if len(sys.argv) != 3:
    print "Usage: install_os <VM_name> <path_to_iso>"
    exit()

# Initialize the Parallels API library.
prlsdkapi.init_server_sdk()

# Create a server object.
server = prlsdkapi.Server()

# Log in.
try:
    "10.30.18.99", "root", "qawsed", consts.PSL_NORMAL_SECURITY
    result = server.login("10.30.18.99", "root", "qawsed", '', 0, 0,
consts.PSL_HIGH_SECURITY).wait()
except prlsdkapi.prlsdk.PrlSDKError, e:
    print "Login error: %s" % e
    exit()

# Get a list of virtual machines.
```

```
# Find the specified virtual machine and
# obtain an object identifying it.
try:
    result = server.get_vm_list().wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e
    exit()

found = False
for i in range(result.get_params_count()):
    VM = result.get_param_by_index(i)
    if VM.get_name() == sys.argv[1]:
        found = True
        break

if found == False:
    print "Specified virtual machine not found."
    exit()

# Obtain an object identifying the
# CD/DVD drive
cdrom = VM.get_optical_disk(0)

# Begin the virtual machine editing operation.
VM.begin_edit()

# Mount the OS installation ISO image.
cdrom.set_emulated_type(consts.PDT_USE_IMAGE_FILE)
cdrom.set_sys_name(sys.argv[2])
cdrom.set_image_path(sys.argv[2])

# Commit the changes to the virtual machine.
VM.commit()

# Start the virtual machine.
VM.start().wait()

# Instantiate the prlsdkapi.VmIO class.
vm_io = prlsdkapi.VmIO()

# Begin a Remote Desktop Access session.
try:
    vm_io.connect_to_vm(VM).wait()
except prlsdkapi.PrlSDKError, e:
    print "Error: %s" % e
    exit()

# Define the name of the file to save the
# captured screen data to.
current_screen = "current.bmp"

# Set the reference screenshot count to 0.
ref_count = 0

# Define the names of files containing
# reference screenshots.
ref_files = ['1.bmp', '2.bmp', '3.bmp']

# Define the keyboard keys that will be used
# to interact with the remote desktop.
keys = ['enter', 'f1', 'enter']

# Capture the virtual machine screen, find the matching
# screen in the list of reference files and send the
# appropriate keyboard command to the virtual machine.
# Repeat for all screens.
while True:
```

```
# Get a reference file name.
ref_screen = ref_files[ref_count]

# Capture the current virtual machine desktop screen and save it
# into a file as a BMP image.
# The parameters are:
#   Target file name
#   X coordinate
#   Y coordinate
#   Width (-1 for full screen)
#   Height (-1 for full screen)
#   Image format
vm_io.sync_capture_screen_region_to_file(VM, current_screen, \
                                         consts.PIF_BMP)

# Do a bit-by-bit comparison of the captured screen and
# the reference screen for this iteration.
# The actual comparison procedure depends on the data format used.
# The bb_cmp() function DOES NOT exist in this sample program.
# You will have to implement it yourself.
if bb_cmp(current_screen, ref_screen):
    print "%d screen valid" % ref_count

    # Press the appropriate key.
    press = consts.PKE_PRESS
    release = consts.PKE_RELEASE
    key = keys[ref_count]
    key = key.upper()

    # Determine the key scan code based on its name.
    # The codes are defined in the ScanCodesList constant.
    # For the complete list of codes, start Python from the command line,
    # import the prlSDKAPI module, and issue the
    # "print prlSDKAPI.prlSDK.consts.ScanCodesList" statement.
    scan_code = consts.ScanCodesList[key]

    # Send the key command to the virtual machine.
    vm_io.send_key_event(VM, scan_code, press)
    vm_io.send_key_event(VM, scan_code, release)

    # If still have reference files to process, continue, otherwise, exit.
    if ref_count < (len(ref_files) - 1):
        ref_count = ref_count + 1
    else:
        print "Os is installed."
        break

# End the Remote Desktop Access session.
vm_io.disconnect_from_vm(VM)

# Stop the virtual machine.
VM.stop().wait()

# Logoff and deinitialize the library.
server.logoff()
prlSDKAPI.deinit_sdk()
```

Index

A

Adding a Hard Disk Drive - 171
 Adding a Network Adapter - 173
 Adding an Existing Virtual Machine - 79, 175
 Asynchronous Functions - 22
 Asynchronous Methods - 145

B

Bridged Networking - 95
 Building the Dynamic Library - 140

C

Class Methods - 144
 Classes - 144
 Cloning a Virtual Machine - 82, 177
 Common Network Requirements - 7
 Compiling Client Applications - 8
 Compiling with Framework - 14
 Compiling with SdkWrap - 9
 Connecting to Parallels Service and Logging In - 152
 Converting a Regular Virtual Machine to a Template - 106
 Converting a Template to a Regular Virtual Machine - 107
 Creating a Basic Application - 149
 Creating a New Virtual Machine - 72, 163
 Creating a New Virtual Machine From a Template. - 108
 Creating a Simple OS Installation Program - 179
 Creating a Template From Scratch - 104

D

Deleting a Virtual Machine - 84
 Determining Virtual Machine State - 66

E

Encryption Plug-in - 130
 Encryption Plug-in Basics - 130
 Error Handling - 28, 147
 Events - 110

G

Getting Started - 5

H

Handles - 20
 Hard Disks - 90
 Host Operations - 35, 155
 Host-only and Shared Networking - 93

I

Implementing a Plug-in - 134

L

Linux - 19
 Linux Clients - 7

M

Mac OS X - 8
 Mac OS X Clients - 6
 Managing Files In The Host OS - 50
 Managing Licenses - 53
 Managing Parallels Service Preferences - 39, 157
 Managing Parallels Service Users - 45
 Managing User Access Rights - 99
 Modifying Virtual Machine Configuration - 85, 168

N

Name, Description, Boot Options - 88
 Name, Description, Boot Options, RAM size - 169
 Network Adapters - 92

O

Obtaining a List of Templates - 102
 Obtaining a PHT_VM_CONFIGURATION handle - 87
 Obtaining a Problem Report - 56
 Obtaining Performance Report - 122
 Obtaining Server Handle and Logging In - 31
 Obtaining the Virtual Machine List - 159
 Obtaining the Virtual Machines List - 59
 Obtaining Virtual Machine Configuration Data - 165
 Obtaining Virtual Machine Configuration Information - 64
 Overview - 5

P

Package and Modules - 143
Parallels C API by Example - 30
Parallels C API Concepts - 8
Parallels Python API by Example - 148
Parallels Python API Concepts - 142
Performance Monitoring - 125
Performance Statistics - 121
Performing Power Operations - 162
Plug-in Installation and Usage - 141
PrlVm_BeginEdit and PrlVm_Commit
 Functions - 86

R

RAM Size - 89
Receiving and Handling Events - 111
Remote Desktop Access - 178
Removing an Existing Virtual Machine - 176
Responding to Parallels Service Questions -
 114
Retrieving Host Configuration Info - 155
Retrieving Host Configuration Information -
 36

S

Searching for a Virtual Machine - 161
Searching for Parallels Servers - 42
Searching for Virtual Machine by Name - 62
Searching for Virtual Machines - 75
Starting, Stopping, Resetting a Virtual
 Machine - 69
Strings as Return Values - 26
Suspending and Pausing a Virtual Machine -
 70
Synchronous Functions - 21
Synchronous Methods - 144
System Requirements - 6

T

The Encryption API Reference - 131

V

Virtual Machine Operations - 58, 158
Vm.begin_edit and Vm.commit Methods - 168

W

Windows - 18
Windows Clients - 6
Working with Virtual Machine Templates -
 101