Oracle VM VirtualBox®

Programming Guide and Reference

Version 4.0.30

http://www.virtualbox.org

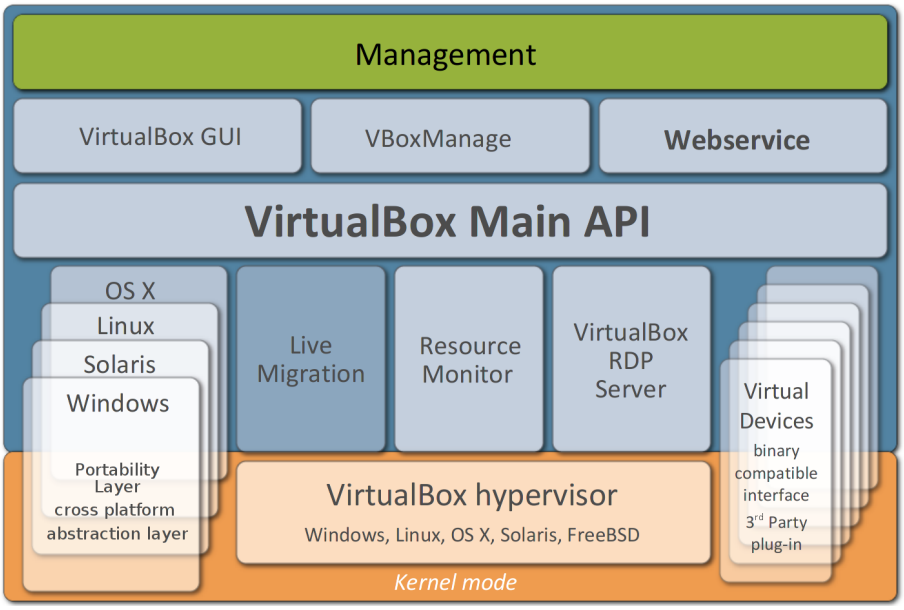
Contents

1 Introduction

VirtualBox comes with comprehensive support for third-party developers. This Software Development Kit (SDK) contains all the documentation and interface files that are needed to write code that interacts with VirtualBox.

1.1 Modularity: the building blocks of VirtualBox

VirtualBox is cleanly separated into several layers, which can be visualized like in the picture below:



The orange area represents code that runs in kernel mode, the blue area represents userspace code.

At the bottom of the stack resides the hypervisor – the core of the virtualization engine, controlling execution of the virtual machines and making sure they do not conflict with each other or whatever the host computer is doing otherwise.

On top of the hypervisor, additional internal modules provide extra functionality. For example, the RDP server, which can deliver the graphical output of a VM remotely to an RDP client, is a separate module that is only loosely tacked into the virtual graphics device. Live Migration and Resource Monitor are additional modules currently in the process of being added to VirtualBox.

What is primarily of interest for purposes of the SDK is the API layer block that sits on top of all the previously mentioned blocks. This API, which we call the **"Main API"**, exposes the entire feature set of the virtualization engine below. It is completely documented in this SDK Reference – see chapter 5, *Classes (interfaces)*, page 43 and chapter 6, *Enumerations (enums)*, page 250 – and available to anyone who wishes to control VirtualBox programmatically. We chose the name "Main API" to differentiate it from other programming interfaces of VirtualBox that may be publicly accessible.

With the Main API, you can create, configure, start, stop and delete virtual machines, retrieve performance statistics about running VMs, configure the VirtualBox installation in general, and

more. In fact, internally, the front-end programs `VirtualBox` and `VBoxManage` use nothing but this API as well – there are no hidden backdoors into the virtualization engine for our own front-ends. This ensures the entire Main API is both well-documented and well-tested. (The same applies to `VBoxHeadless`, which is not shown in the image.)

1.2 Two guises of the same "Main API": the web service or COM/XPCOM

There are several ways in which the Main API can be called by other code:

1. VirtualBox comes with a **web service** that maps nearly the entire Main API. The web service ships in a stand-alone executable (`vboxwebsrv`) that, when running, acts as an HTTP server, accepts SOAP connections and processes them.

   Since the entire web service API is publicly described in a web service description file (in WSDL format), you can write client programs that call the web service in any language with a toolkit that understands WSDL. These days, that includes most programming languages that are available: Java, C++, .NET, PHP, Python, Perl and probably many more.

   All of this is explained in detail in subsequent chapters of this book.

   There are two ways in which you can write client code that uses the web service:

   a) For Java as well as Python, the SDK contains easy-to-use classes that allow you to use the web service in an object-oriented, straightforward manner. We shall refer to this as the **"object-oriented web service (OOWS)"**.

      The OO bindings for Java are described in chapter 10, *Using Java API*, page 281, those for Python in chapter 2.1.2, *The object-oriented web service for Python*, page 23.

   b) Alternatively, you can use the web service directly, without the object-oriented client layer. We shall refer to this as the **"raw web service"**.

      You will then have neither native object orientation nor full type safety, since web services are neither object-oriented nor stateful. However, in this way, you can write client code even in languages for which we do not ship object-oriented client code; all you need is a programming language with a toolkit that can parse WSDL and generate client wrapper code from it.

      We describe this further in chapter 2.2, *Using the raw web service with any language*, page 24, with samples for Java and Perl.

2. Internally, for portability and easier maintenance, the Main API is implemented using the **Component Object Model (COM)**, an interprocess mechanism for software components originally introduced by Microsoft for Microsoft Windows. On a Windows host, VirtualBox will use Microsoft COM; on other hosts where COM is not present, it ships with XPCOM, a free software implementation of COM originally created by the Mozilla project for their browsers.

   So, if you are familiar with COM and the C++ programming language (or with any other programming language that can handle COM/XPCOM objects, such as Java, Visual Basic or C#), then you can use the COM/XPCOM API directly. VirtualBox comes with all necessary files and documentation to build fully functional COM applications. For an introduction, please see chapter 2.3, *Using COM/XPCOM directly*, page 30 below.

   The VirtualBox front-ends (the graphical user interfaces as well as the command line), which are all written in C++, use COM/XPCOM to call the Main API. Technically, the web service is another front-end to this COM API, mapping almost all of it to SOAP clients.

If you wonder which way to choose, here are a few comparisons:

Web service	COM/XPCOM
Pro: Easy to use with Java and Python with the object-oriented web service; extensive support even with other languages (C++, .NET, PHP, Perl and others)	**Con:** Usable from languages where COM bridge available (most languages on Windows platform, Python and C++ on other hosts)
Pro: Client can be on remote machine	**Con:** Client must be on the same host where virtual machine is executed
Con: Significant overhead due to XML marshalling over the wire for each method call	**Pro:** Relatively low invocation overhead

In the following chapters, we will describe the different ways in which to program VirtualBox, starting with the method that is easiest to use and then increase complexity as we go along.

1.3 About web services in general

Web services are a particular type of programming interface. Whereas, with "normal" programming, a program calls an application programming interface (API) defined by another program or the operating system and both sides of the interface have to agree on the calling convention and, in most cases, use the same programming language, web services use Internet standards such as HTTP and XML to communicate.[1]

In order to successfully use a web service, a number of things are required – primarily, a web service accepting connections; service descriptions; and then a client that connects to that web service. The connections are governed by the SOAP standard, which describes how messages are to be exchanged between a service and its clients; the service descriptions are governed by WSDL.

In the case of VirtualBox, this translates into the following three components:

1. The VirtualBox web service (the "server"): this is the `vboxwebsrv` executable shipped with VirtualBox. Once you start this executable (which acts as a HTTP server on a specific TCP/IP port), clients can connect to the web service and thus control a VirtualBox installation.

2. VirtualBox also comes with WSDL files that describe the services provided by the web service. You can find these files in the `sdk/bindings/webservice/` directory. These files are understood by the web service toolkits that are shipped with most programming languages and enable you to easily access a web service even if you don't use our object-oriented client layers. VirtualBox is shipped with pregenerated web service glue code for several languages (Python, Perl, Java).

3. A client that connects to the web service in order to control the VirtualBox installation.

   Unless you play with some of the samples shipped with VirtualBox, this needs to be written by you.

[1] In some ways, web services promise to deliver the same thing as CORBA and DCOM did years ago. However, while these previous technologies relied on specific binary protocols and thus proved to be difficult to use between diverging platforms, web services circumvent these incompatibilities by using text-only standards like HTTP and XML. On the downside (and, one could say, typical of things related to XML), a lot of standards are involved before a web service can be implemented. Many of the standards invented around XML are used one way or another. As a result, web services are slow and verbose, and the details can be incredibly messy. The relevant standards here are called SOAP and WSDL, where SOAP describes the format of the messages that are exchanged (an XML document wrapped in an HTTP header), and WSDL is an XML format that describes a complete API provided by a web service. WSDL in turn uses XML Schema to describe types, which is not exactly terse either. However, as you will see from the samples provided in this chapter, the VirtualBox web service shields you from these details and is easy to use.

1.4 Running the web service

The web service ships in an stand-alone executable, `vboxwebsrv`, that, when running, acts as a HTTP server, accepts SOAP connections and processes them – remotely or from the same machine.

> **Note:** The web service executable is not contained with the VirtualBox SDK, but instead ships with the standard VirtualBox binary package for your specific platform. Since the SDK contains only platform-independent text files and documentation, the binaries are instead shipped with the platform-specific packages.

The `vboxwebsrv` program, which implements the web service, is a text-mode (console) program which, after being started, simply runs until it is interrupted with Ctrl-C or a kill command.

Once the web service is started, it acts as a front-end to the VirtualBox installation of the user account that it is running under. In other words, if the web service is run under the user account of `user1`, it will see and manipulate the virtual machines and other data represented by the VirtualBox data of that user (e.g., on a Linux machine, under `/home/user1/.VirtualBox`; see the VirtualBox User Manual for details on where this data is stored).

1.4.1 Command line options of vboxwebsrv

The web service supports the following command line options:

- `--help` (or `-h`): print a brief summary of command line options.
- `--background` (or `-b`): run the web service as a background daemon. This option is not supported on Windows hosts.
- `--host` (or `-H`): This specifies the host to bind to and defaults to "localhost".
- `--port` (or `-p`): This specifies which port to bind to on the host and defaults to 18083.
- `--timeout` (or `-t`): This specifies the session timeout, in seconds, and defaults to 300 (five minutes). A web service client that has logged on but makes no calls to the web service will automatically be disconnected after the number of seconds specified here, as if it had called the `IWebSessionManager::logoff()` method provided by the web service itself.

  It is normally vital that each web service client call this method, as the web service can accumulate large amounts of memory when running, especially if a web service client does not properly release managed object references. As a result, this timeout value should not be set too high, especially on machines with a high load on the web service, or the web service may eventually deny service.
- `--check-interval` (or `-i`): This specifies the interval in which the web service checks for timed-out clients, in seconds, and defaults to 5. This normally does not need to be changed.
- `--verbose` (or `-v`): Normally, the webservice outputs only brief messages to the console each time a request is served. With this option, the webservice prints much more detailed data about every request and the COM methods that those requests are mapped to internally, which can be useful for debugging client programs.
- `--logfile` (or `-F`) `<file>`: If this is specified, the webservice not only prints its output to the console, but also writes it to the specified file. The file is created if it does not exist; if it does exist, new output is appended to it. This is useful if you run the webservice unattended and need to debug problems after they have occurred.

1.4.2 Authenticating at web service logon

As opposed to the COM/XPCOM variant of the Main API, a client that wants to use the web service must first log on by calling the `IWebsessionManager::logon()` API (see chapter 5.103.3, *logon*, page 249) that is specific to the web service. Logon is necessary for the web service to be stateful; internally, it maintains a session for each client that connects to it.

The `IWebsessionManager::logon()` API takes a user name and a password as arguments, which the web service then passes to a customizable authentication plugin that performs the actual authentication.

For testing purposes, it is recommended that you first disable authentication with this command:

```
VBoxManage setproperty websrvauthlibrary null
```

> **Warning:** This will cause all logons to succeed, regardless of user name or password. This should of course not be used in a production environment.

Generally, the mechanism by which clients are authenticated is configurable by way of the `VBoxManage` command:

```
VBoxManage setproperty websrvauthlibrary default|null|<library>
```

This way you can specify any shared object/dynamic link module that conforms with the specifications for VirtualBox external authentication modules as laid out in section **VRDE authentication** of the VirtualBox User Manual; the web service uses the same kind of modules as the VirtualBox VRDE server. For technical details on VirtualBox external authentication modules see chapter 9, *VirtualBox external authentication modules*, page 279

By default, after installation, the web service uses the VBoxAuth module that ships with VirtualBox. This module uses PAM on Linux hosts to authenticate users. Any valid username/password combination is accepted, it does not have to be the username and password of the user running the webservice daemon. Unless `vboxwebsrv` runs as root, PAM authentication can fail, because sometimes the file `/etc/shadow`, which is used by PAM, is not readable. On most Linux distribution PAM uses a suid root helper internally, so make sure you test this before deploying it. One can override this behavior by setting the environment variable `VBOX_PAM_ALLOW_INACTIVE` which will suppress failures when unable to read the shadow password file. Please use this variable carefully, and only if you fully understand what you're doing.

1.4.3 Solaris host: starting the web service via SMF

On Solaris hosts, the VirtualBox web service daemon is integrated into the SMF framework. You can change the parameters, but don't have to if the defaults below already match your needs:

```
svccfg -s svc:/application/virtualbox/webservice:default setprop config/host=localhost
svccfg -s svc:/application/virtualbox/webservice:default setprop config/port=18083
svccfg -s svc:/application/virtualbox/webservice:default setprop config/user=root
```

If you made any change, don't forget to run the following command to put the changes into effect immediately:

```
svcadm refresh svc:/application/virtualbox/webservice:default
```

If you forget the above command then the previous settings will be used when enabling the service. Check the current property settings with:

```
svcprop -p config svc:/application/virtualbox/webservice:default
```

When everything is configured correctly you can start the VirtualBox webservice with the following command:

```
svcadm enable svc:/application/virtualbox/webservice:default
```

For more information about SMF, please refer to the Solaris documentation.

2 Environment-specific notes

The Main API described in chapter 5, *Classes (interfaces)*, page 43 and chapter 6, *Enumerations (enums)*, page 250 is mostly identical in all the supported programming environments which have been briefly mentioned in the introduction of this book. As a result, the Main API's general concepts described in chapter 3, *Basic VirtualBox concepts; some examples*, page 38 are the same whether you use the object-oriented web service (OOWS) for JAX-WS or a raw web service connection via, say, Perl, or whether you use C++ COM bindings.

Some things are different depending on your environment, however. These differences are explained in this chapter.

2.1 Using the object-oriented web service (OOWS)

As explained in chapter 1.2, *Two guises of the same "Main API": the web service or COM/XPCOM*, page 16, VirtualBox ships with client-side libraries for Java, Python and PHP that allow you to use the VirtualBox web service in an intuitive, object-oriented way. These libraries shield you from the client-side complications of managed object references and other implementation details that come with the VirtualBox web service. (If you are interested in these complications, have a look at chapter 2.2, *Using the raw web service with any language*, page 24).

We recommend that you start your experiments with the VirtualBox web service by using our object-oriented client libraries for JAX-WS, a web service toolkit for Java, which enables you to write code to interact with VirtualBox in the simplest manner possible.

As "interfaces", "attributes" and "methods" are COM concepts, please read the documentation in chapter 5, *Classes (interfaces)*, page 43 and chapter 6, *Enumerations (enums)*, page 250 with the following notes in mind.

The OOWS bindings attempt to map the Main API as closely as possible to the Java, Python and PHP languages. In other words, objects are objects, interfaces become classes, and you can call methods on objects as you would on local objects.

The main difference remains with attributes: to read an attribute, call a "getXXX" method, with "XXX" being the attribute name with a capitalized first letter. So when the Main API Reference says that `IMachine` has a "name" attribute (see IMachine::name), call `getName()` on an IMachine object to obtain a machine's name. Unless the attribute is marked as read-only in the documentation, there will also be a corresponding "set" method.

2.1.1 The object-oriented web service for JAX-WS

JAX-WS is a powerful toolkit by Sun Microsystems to build both server and client code with Java. It is part of Java 6 (JDK 1.6), but can also be obtained separately for Java 5 (JDK 1.5). The VirtualBox SDK comes with precompiled OOWS bindings working with both Java 5 and 6.

The following sections explain how to get the JAX-WS sample code running and explain a few common practices when using the JAX-WS object-oriented web service.

2.1.1.1 Preparations

Since JAX-WS is already integrated into Java 6, no additional preparations are needed for Java 6.

If you are using Java 5 (JDK 1.5.x), you will first need to download and install an external JAX-WS implementation, as Java 5 does not support JAX-WS out of the box; for example, you can

download one from here: https://jax-ws.dev.java.net/2.1.4/JAXWS2.1.4-20080502.jar. Then perform the installation (`java -jar JAXWS2.1.4-20080502.jar`).

2.1.1.2 Getting started: running the sample code

To run the OOWS for JAX-WS samples that we ship with the SDK, perform the following steps:

1. Open a terminal and change to the directory where the JAX-WS samples reside.[1] Examine the header of `Makefile` to see if the supplied variables (Java compiler, Java executable) and a few other details match your system settings.

2. To start the VirtualBox web service, open a second terminal and change to the directory where the VirtualBox executables are located. Then type:

   ```
   ./vboxwebsrv -v
   ```

   The web service now waits for connections and will run until you press Ctrl+C in this second terminal. The -v argument causes it to log all connections to the terminal. (See chapter 1.4, *Running the web service*, page 18 for details on how to run the web service.)

3. Back in the first terminal and still in the samples directory, to start a simple client example just type:

   ```
   make run16
   ```

   if you're on a Java 6 system; on a Java 5 system, run `make run15` instead.

   This should work on all Unix-like systems such as Linux and Solaris. For Windows systems, use commands similar to what is used in the Makefile.

   This will compile the `clienttest.java` code on the first call and then execute the resulting `clienttest` class to show the locally installed VMs (see below).

The `clienttest` sample imitates a few typical command line tasks that `VBoxManage`, VirtualBox's regular command-line front-end, would provide (see the VirtualBox User Manual for details). In particular, you can run:

- `java clienttest show vms`: show the virtual machines that are registered locally.
- `java clienttest list hostinfo`: show various information about the host this VirtualBox installation runs on.
- `java clienttest startvm <vmname|uuid>`: start the given virtual machine.

The `clienttest.java` sample code illustrates common basic practices how to use the VirtualBox OOWS for JAX-WS, which we will explain in more detail in the following chapters.

2.1.1.3 Logging on to the web service

Before a web service client can do anything useful, two objects need to be created, as can be seen in the `clienttest` constructor:

1. An instance of IWebsessionManager, which is an interface provided by the web service to manage "web sessions" – that is, stateful connections to the web service with persistent objects upon which methods can be invoked.

   In the OOWS for JAX-WS, the IWebsessionManager class must be constructed explicitly, and a URL must be provided in the constructor that specifies where the web service (the server) awaits connections. The code in `clienttest.java` connects to "http://localhost:18083/", which is the default.

   The port number, by default 18083, must match the port number given to the `vboxwebsrv` command line; see chapter 1.4.1, *Command line options of vboxwebsrv*, page 18.

[1] In `sdk/bindings/webservice/java/jax-ws/samples/`.

2. After that, the code calls IWebsessionManager::logon(), which is the first call that actually communicates with the server. This authenticates the client with the web service and returns an instance of IVirtualBox, the most fundamental interface of the VirtualBox web service, from which all other functionality can be derived.

   If logon doesn't work, please take another look at chapter 1.4.2, *Authenticating at web service logon*, page 19.

2.1.1.4 Object management

The current OOWS for JAX-WS has certain memory management related limitations. When you no longer need an object, call its IManagedObjectRef::release() method explicitly, which frees appropriate managed reference, as is required by the raw webservice; see chapter 2.2.3.3, *Managed object references*, page 28 for details. This limitation may be reconsidered in a future version of the VirtualBox SDK.

2.1.2 The object-oriented web service for Python

VirtualBox comes with two flavors of a Python API: one for web service, discussed here, and one for the COM/XPCOM API discussed in chapter 2.3.1, *Python COM API*, page 30. The client code is mostly similar, except for the initialization part, so it is up to the application developer to choose the appropriate technology. Moreover, a common Python glue layer exists, abstracting out concrete platform access details, see chapter 2.3.2, *Common Python bindings layer*, page 30.

As indicated in chapter 1.2, *Two guises of the same "Main API": the web service or COM/XPCOM*, page 16, the COM/XPCOM API gives better performance without the SOAP overhead, and does not require a web server to be running. On the other hand, the COM/XPCOM Python API requires a suitable Python bridge for your Python installation (VirtualBox ships the most important ones for each platform[2]), and you cannot connect to VirtualBox remotely. On Windows, you can use the Main API from Python if the Win32 extensions package for Python[3] is installed.

The VirtualBox OOWS for Python relies on the Python ZSI SOAP implementation (see `http://pywebsvcs.sourceforge.net/zsi.html`), which you will need to install locally before trying the examples. Most Linux distributions come with package for ZSI, such as `python-zsi` in Ubuntu.

To get started, open a terminal and change to the `bindings/glue/python/sample` directory, which contains an example of a simple interactive shell able to control a VirtualBox instance. The shell is written using the API layer, thereby hiding different implementation details, so it is actually an example of code share among XPCOM, MSCOM and web services. If you are interested in how to interact with the webservices layer directly, have a look at `install/vboxapi/__init__.py` which contains the glue layer for all target platforms (i.e. XPCOM, MSCOM and web services).

To start the shell, perform the following commands:

```
/opt/VirtualBox/vboxwebsrv -t 0
           # start webservice with object autocollection disabled
export VBOX_PROGRAM_PATH=/opt/VirtualBox
           # your VirtualBox installation directory
export VBOX_SDK_PATH=/home/youruser/vbox-sdk
           # where you've extracted the SDK
./vboxshell.py -w
```

See chapter 4, *The VirtualBox shell*, page 41 for more details on the shell's functionality. For you, as a VirtualBox application developer, the vboxshell sample could be interesting as an example of how to write code targeting both local and remote cases (COM/XPCOM and SOAP). The common

[2] On On Mac OS X only the Python versions bundled with the OS are officially supported. This means Python 2.3 for 10.4, Python 2.5 for 10.5 and Python 2.5 and 2.6 for 10.6.

[3] See `http://sourceforge.net/project/showfiles.php?group_id=78018`.

part of the shell is the same – the only difference is how it interacts with the invocation layer. You can use the `connect` shell command to connect to remote VirtualBox servers; in this case you can skip starting the local webserver.

2.1.3 The object-oriented web service for PHP

VirtualBox also comes with object-oriented web service (OOWS) wrappers for PHP5. These wrappers rely on the PHP SOAP Extension[4], which can be installed by configuring PHP with `--enable-soap`.

2.2 Using the raw web service with any language

The following examples show you how to use the raw web service, without the object-oriented client-side code that was described in the previous chapter.

Generally, when reading the documentation in chapter 5, *Classes (interfaces)*, page 43 and chapter 6, *Enumerations (enums)*, page 250, due to the limitations of SOAP and WSDL lined out in chapter 2.2.3.1, *Fundamental conventions*, page 26, please have the following notes in mind:

1. Any COM method call becomes a **plain function call** in the raw web service, with the object as an additional first parameter (before the "real" parameters listed in the documentation). So when the documentation says that the `IVirtualBox` interface supports the `createMachine()` method (see IVirtualBox::createMachine()), the web service operation is `IVirtualBox_createMachine(...)`, and a managed object reference to an `IVirtualBox` object must be passed as the first argument.

2. For **attributes** in interfaces, there will be at least one "get" function; there will also be a "set" function, unless the attribute is "readonly". The attribute name will be appended to the "get" or "set" prefix, with a capitalized first letter. So, the "version" readonly attribute of the `IVirtualBox` interface can be retrieved by calling `IVirtualBox_getVersion(vbox)`, with `vbox` being the VirtualBox object.

3. Whenever the API documentation says that a method (or an attribute getter) returns an **object**, it will returned a managed object reference in the web service instead. As said above, managed object references should be released if the web service client does not log off again immediately!

2.2.1 Raw web service example for Java with Axis

Axis is an older web service toolkit created by the Apache foundation. If your distribution does not have it installed, you can get a binary from `http://www.apache.org`. The following examples assume that you have Axis 1.4 installed.

The VirtualBox SDK ships with an example for Axis that, again, is called `clienttest.java` and that imitates a few of the commands of `VBoxManage` over the wire.

Then perform the following steps:

1. Create a working directory somewhere. Under your VirtualBox installation directory, find the `sdk/webservice/samples/java/axis/` directory and copy the file `clienttest.java` to your working directory.

2. Open a terminal in your working directory. Execute the following command:

```
java org.apache.axis.wsdl.WSDL2Java /path/to/vboxwebService.wsdl
```

[4] See `http://www.php.net/soap`.

The `vboxwebService.wsdl` file should be located in the `sdk/webservice/` directory.

If this fails, your Apache Axis may not be located on your system classpath, and you may have to adjust the CLASSPATH environment variable. Something like this:

```
export CLASSPATH="/path-to-axis-1_4/lib/*":$CLASSPATH
```

Use the directory where the Axis JAR files are located. Mind the quotes so that your shell passes the "*" character to the java executable without expanding. Alternatively, add a corresponding `-classpath` argument to the "java" call above.

If the command executes successfully, you should see an "org" directory with subdirectories containing Java source files in your working directory. These classes represent the interfaces that the VirtualBox web service offers, as described by the WSDL file.

This is the bit that makes using web services so attractive to client developers: if a language's toolkit understands WSDL, it can generate large amounts of support code automatically. Clients can then easily use this support code and can be done with just a few lines of code.

3. Next, compile the `clienttest.java` source:

   ```
   javac clienttest.java
   ```

   This should yield a "clienttest.class" file.

4. To start the VirtualBox web service, open a second terminal and change to the directory where the VirtualBox executables are located. Then type:

   ```
   ./vboxwebsrv -v
   ```

   The web service now waits for connections and will run until you press Ctrl+C in this second terminal. The -v argument causes it to log all connections to the terminal. (See chapter 1.4, *Running the web service*, page 18 for details on how to run the web service.)

5. Back in the original terminal where you compiled the Java source, run the resulting binary, which will then connect to the web service:

   ```
   java clienttest
   ```

   The client sample will connect to the web service (on localhost, but the code could be changed to connect remotely if the web service was running on a different machine) and make a number of method calls. It will output the version number of your VirtualBox installation and a list of all virtual machines that are currently registered (with a bit of seemingly random data, which will be explained later).

2.2.2 Raw web service example for Perl

We also ship a small sample for Perl. It uses the SOAP::Lite perl module to communicate with the VirtualBox web service.

The `sdk/bindings/webservice/perl/lib/` directory contains a pre-generated Perl module that allows for communicating with the web service from Perl. You can generate such a module yourself using the "stubmaker" tool that comes with SOAP::Lite, but since that tool is slow as well as sometimes unreliable, we are shipping a working module with the SDK for your convenience.

Perform the following steps:

1. If SOAP::Lite is not yet installed on your system, you will need to install the package first. On Debian-based systems, the package is called `libsoap-lite-perl`; on Gentoo, it's `dev-perl/SOAP-Lite`.

2. Open a terminal in the `sdk/bindings/webservice/perl/samples/` directory.

3. To start the VirtualBox web service, open a second terminal and change to the directory where the VirtualBox executables are located. Then type:

   ```
   ./vboxwebsrv -v
   ```

   The web service now waits for connections and will run until you press Ctrl+C in this second terminal. The -v argument causes it to log all connections to the terminal. (See chapter 1.4, *Running the web service*, page 18 for details on how to run the web service.)

4. In the first terminal with the Perl sample, run the clienttest.pl script:

   ```
   perl -I ../lib clienttest.pl
   ```

2.2.3 Programming considerations for the raw web service

If you use the raw web service, you need to keep a number of things in mind, or you will sooner or later run into issues that are not immediately obvious. By contrast, the object-oriented client-side libraries described in chapter 2.1, *Using the object-oriented web service (OOWS)*, page 21 take care of these things automatically and thus greatly simplify using the web service.

2.2.3.1 Fundamental conventions

If you are familiar with other web services, you may find the VirtualBox web service to behave a bit differently to accommodate for the fact that VirtualBox web service more or less maps the VirtualBox Main COM API. The following main differences had to be taken care of:

- Web services, as expressed by WSDL, are not object-oriented. Even worse, they are normally stateless (or, in web services terminology, "loosely coupled"). Web service operations are entirely procedural, and one cannot normally make assumptions about the state of a web service between function calls.

  In particular, this normally means that you cannot work on objects in one method call that were created by another call.

- By contrast, the VirtualBox Main API, being expressed in COM, is object-oriented and works entirely on objects, which are grouped into public interfaces, which in turn have attributes and methods associated with them.

For the VirtualBox web service, this results in three fundamental conventions:

1. All **function names** in the VirtualBox web service consist of an interface name and a method name, joined together by an underscore. This is because there are only functions ("operations") in WSDL, but no classes, interfaces, or methods.

   In addition, all calls to the VirtualBox web service (except for logon, see below) take a **managed object reference** as the first argument, representing the object upon which the underlying method is invoked. (Managed object references are explained in detail below; see chapter 2.2.3.3, *Managed object references*, page 28.)

   So, when one would normally code, in the pseudo-code of an object-oriented language, to invoke a method upon an object:

   ```
   IMachine machine;
   result = machine.getName();
   ```

   In the VirtualBox web service, this looks something like this (again, pseudo-code):

   ```
   IMachineRef machine;
   result = IMachine_getName(machine);
   ```

2. To make the web service stateful, and objects persistent between method calls, the VirtualBox web service introduces a **session manager** (by way of the IWebsessionManager interface), which manages object references. Any client wishing to interact with the web service must first log on to the session manager and in turn receives a managed object reference to an object that supports the IVirtualBox interface (the basic interface in the Main API).

In other words, as opposed to other web services, **the VirtualBox web service is both object-oriented and stateful.**

2.2.3.2 Example: A typical web service client session

A typical short web service session to retrieve the version number of the VirtualBox web service (to be precise, the underlying Main API version number) looks like this:

1. A client logs on to the web service by calling IWebsessionManager::logon() with a valid user name and password. See chapter 1.4.2, *Authenticating at web service logon*, page 19 for details about how authentication works.

2. On the server side, `vboxwebsrv` creates a session, which persists until the client calls IWebsessionManager::logoff() or the session times out after a configurable period of inactivity (see chapter 1.4.1, *Command line options of vboxwebsrv*, page 18).

   For the new session, the web service creates an instance of IVirtualBox. This interface is the most central one in the Main API and allows access to all other interfaces, either through attributes or method calls. For example, IVirtualBox contains a list of all virtual machines that are currently registered (as they would be listed on the left side of the VirtualBox main program).

   The web service then creates a managed object reference for this instance of IVirtualBox and returns it to the calling client, which receives it as the return value of the logon call. Something like this:

   ```
   string oVirtualBox;
   oVirtualBox = webservice.IWebsessionManager_logon("user", "pass");
   ```

   (The managed object reference "oVirtualBox" is just a string consisting of digits and dashes. However, it is a string with a meaning and will be checked by the web service. For details, see below. As hinted above, IWebsessionManager::logon() is the *only* operation provided by the web service which does not take a managed object reference as the first argument!)

3. The VirtualBox Main API documentation says that the `IVirtualBox` interface has a version attribute, which is a string. For each attribute, there is a "get" and a "set" method in COM, which maps to according operations in the web service. So, to retrieve the "version" attribute of this `IVirtualBox` object, the web service client does this:

   ```
   string version;
   version = webservice.IVirtualBox_getVersion(oVirtualBox);

   print version;
   ```

   And it will print "4.0.30".

4. The web service client calls IWebsessionManager::logoff() with the VirtualBox managed object reference. This will clean up all allocated resources.

2.2.3.3 Managed object references

To a web service client, a managed object reference looks like a string: two 64-bit hex numbers separated by a dash. This string, however, represents a COM object that "lives" in the web service process. The two 64-bit numbers encoded in the managed object reference represent a session ID (which is the same for all objects in the same web service session, i.e. for all objects after one logon) and a unique object ID within that session.

Managed object references are created in two situations:

1. When a client logs on, by calling IWebsessionManager::logon().

   Upon logon, the websession manager creates one instance of IVirtualBox and another object of ISession representing the web service session. This can be retrieved using IWebsessionManager::getSessionObject().

   (Technically, there is always only one IVirtualBox object, which is shared between all sessions and clients, as it is a COM singleton. However, each session receives its own managed object reference to it. The ISession object, however, is created and destroyed for each session.)

2. Whenever a web service clients invokes an operation whose COM implementation creates COM objects.

   For example, IVirtualBox::createMachine() creates a new instance of IMachine; the COM object returned by the COM method call is then wrapped into a managed object reference by the web server, and this reference is returned to the web service client.

Internally, in the web service process, each managed object reference is simply a small data structure, containing a COM pointer to the "real" COM object, the web session ID and the object ID. This structure is allocated on creation and stored efficiently in hashes, so that the web service can look up the COM object quickly whenever a web service client wishes to make a method call. The random session ID also ensures that one web service client cannot intercept the objects of another.

Managed object references are not destroyed automatically and must be released by explicitly calling IManagedObjectRef::release(). This is important, as otherwise hundreds or thousands of managed object references (and corresponding COM objects, which can consume much more memory!) can pile up in the web service process and eventually cause it to deny service.

To reiterate: The underlying COM object, which the reference points to, is only freed if the managed object reference is released. It is therefore vital that web service clients properly clean up after the managed object references that are returned to them.

When a web service client calls IWebsessionManager::logoff(), all managed object references created during the session are automatically freed. For short-lived sessions that do not create a lot of objects, logging off may therefore be sufficient, although it is certainly not "best practice".

2.2.3.4 Some more detail about web service operation

SOAP messages Whenever a client makes a call to a web service, this involves a complicated procedure internally. These calls are remote procedure calls. Each such procedure call typically consists of two "message" being passed, where each message is a plain-text HTTP request with a standard HTTP header and a special XML document following. This XML document encodes the name of the procedure to call and the argument names and values passed to it.

To give you an idea of what such a message looks like, assuming that a web service provides a procedure called "SayHello", which takes a string "name" as an argument and returns "Hello" with a space and that name appended, the request message could look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
```

```
 xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:test="http://test/">
<SOAP-ENV:Body>
   <test:SayHello>
    <name>Peter</name>
   </test:SayHello>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A similar message – the "response" message – would be sent back from the web service to the client, containing the return value "Hello Peter".

Most programming languages provide automatic support to generate such messages whenever code in that programming language makes such a request. In other words, these programming languages allow for writing something like this (in pseudo-C++ code):

```
webServiceClass service("localhost", 18083); // server and port
string result = service.SayHello("Peter");  // invoke remote procedure
```

and would, for these two pseudo-lines, automatically perform these steps:

1. prepare a connection to a web service running on port 18083 of "localhost";
2. for the `SayHello()` function of the web service, generate a SOAP message like in the above example by encoding all arguments of the remote procedure call (which could involve all kinds of type conversions and complex marshalling for arrays and structures);
3. connect to the web service via HTTP and send that message;
4. wait for the web service to send a response message;
5. decode that response message and put the return value of the remote procedure into the "result" variable.

Service descriptions in WSDL In the above explanations about SOAP, it was left open how the programming language learns about how to translate function calls in its own syntax into proper SOAP messages. In other words, the programming language needs to know what operations the web service supports and what types of arguments are required for the operation's data in order to be able to properly serialize and deserialize the data to and from the web service. For example, if a web service operation expects a number in "double" floating point format for a particular parameter, the programming language cannot send to it a string instead.

For this, the Web Service Definition Language (WSDL) was invented, another XML substandard that describes exactly what operations the web service supports and, for each operation, which parameters and types are needed with each request and response message. WSDL descriptions can be incredibly verbose, and one of the few good things that can be said about this standard is that it is indeed supported by most programming languages.

So, if it is said that a programming language "supports" web services, this typically means that a programming language has support for parsing WSDL files and somehow integrating the remote procedure calls into the native language syntax – for example, like in the Java sample shown in chapter 2.2.1, *Raw web service example for Java with Axis*, page 24.

For details about how programming languages support web services, please refer to the documentation that comes with the individual languages. Here are a few pointers:

1. For **C++,** among many others, the gSOAP toolkit is a good option. Parts of gSOAP are also used in VirtualBox to implement the VirtualBox web service.
2. For **Java,** there are several implementations already described in this document (see chapter 2.1.1, *The object-oriented web service for JAX-WS*, page 21 and chapter 2.2.1, *Raw web service example for Java with Axis*, page 24).

3. **Perl** supports WSDL via the SOAP::Lite package. This in turn comes with a tool called `stubmaker.pl` that allows you to turn any WSDL file into a Perl package that you can import. (You can also import any WSDL file "live" by having it parsed every time the script runs, but that can take a while.) You can then code (again, assuming the above example):

   ```
   my $result = servicename->sayHello("Peter");
   ```

   A sample that uses SOAP::Lite was described in chapter 2.2.2, *Raw web service example for Perl*, page 25.

2.3 Using COM/XPCOM directly

If you do not require *remote* procedure calls such as those offered by the VirtualBox web service, and if you know Python or C++ as well as COM, you might find it preferable to program VirtualBox's Main API directly via COM.

COM stands for "Component Object Model" and is a standard originally introduced by Microsoft in the 1990s for Microsoft Windows. It allows for organizing software in an object-oriented way and across processes; code in one process may access objects that live in another process.

COM has several advantages: it is language-neutral, meaning that even though all of VirtualBox is internally written in C++, programs written in other languages could communicate with it. COM also cleanly separates interface from implementation, so that external programs need not know anything about the messy and complicated details of VirtualBox internals.

On a Windows host, all parts of VirtualBox will use the COM functionality that is native to Windows. On other hosts (including Linux), VirtualBox comes with a built-in implementation of XPCOM, as originally created by the Mozilla project, which we have enhanced to support interprocess communication on a level comparable to Microsoft COM. Internally, VirtualBox has an abstraction layer that allows the same VirtualBox code to work both with native COM as well as our XPCOM implementation.

2.3.1 Python COM API

On Windows, Python scripts can use COM and VirtualBox interfaces to control almost all aspects of virtual machine execution. As an example, use the following commands to instantiate the VirtualBox object and start a VM:

```
vbox = win32com.client.Dispatch("VirtualBox.VirtualBox")
session = win32com.client.Dispatch("VirtualBox.Session")
mach = vbox.findMachine("uuid or name  of machine to start")
progress = mach.launchVMProcess(session, "gui", "")
progress.waitForCompletion(-1)
```

Also, see `/bindings/glue/python/samples/vboxshell.py` for more advanced usage scenarios. However, unless you have specific requirements, we strongly recommend to use the generic glue layer described in the next section to access MS COM objects.

2.3.2 Common Python bindings layer

As different wrappers ultimately provide access to the same underlying API, and to simplify porting and development of Python application using the VirtualBox Main API, we developed a common glue layer that abstracts out most platform-specific details from the application and allows the developer to focus on application logic. The VirtualBox installer automatically sets up this glue layer for the system default Python install. See below for details on how to set up the glue layer if you want to use a different Python installation.

In this layer, the class `VirtualBoxManager` hides most platform-specific details. It can be used to access both the local (COM) and the webservice-based API. The following code can be used by an application to use the glue layer.

```
# This code assumes vboxapi.py from VirtualBox distribution
# being in PYTHONPATH, or installed system-wide
from vboxapi import VirtualBoxManager

# This code initializes VirtualBox manager with default style
# and parameters
virtualBoxManager = VirtualBoxManager(None, None)

# Alternatively, one can be more verbose, and initialize
# glue with webservice backend, and provide authentication
# information
virtualBoxManager = VirtualBoxManager("WEBSERVICE",
                                      {'url':'http://myhost.com::18083/',
                                      'user':'me',
                                      'password':'secret'})
```

We supply the `VirtualBoxManager` constructor with 2 arguments: style and parameters. Style defines which bindings style to use (could be "MSCOM", "XPCOM" or "WEBSERVICE"), and if set to `None` defaults to usable platform bindings (MS COM on Windows, XPCOM on other platforms). The second argument defines parameters, passed to the platform-specific module, as we do in the second example, where we pass username and password to be used to authenticate against the web service.

After obtaining the `VirtualBoxManager` instance, one can perform operations on the IVirtualBox class. For example, the following code will a start virtual machine by name or ID:

```
vbox = virtualBoxManager.vbox
mgr = virtualBoxManager.mgr
print "Version is",vbox.version

def machById(id):
    mach = None
    for m in virtualBoxManager.getArray(vbox, 'machines'):
        if m.name == id or mach.id == id:
            mach = m
            break
    return mach

name = "Linux"
mach = machById(name)
if mach is None:
    print "cannot find machine",name
else:
    session = mgr.getSessionObject(vbox)
    # one can also start headless session with "headless" instead of "gui"
    progress = mach.launchVMProcess(session, mach.id, "gui", "")
    progress.waitForCompletion(-1)
    session.close()
```

This code also shows cross-platform access to array properties (certain limitations prevent one from using `vbox.machines` to access a list of available virtual machines in case of XPCOM), and a mechanism of uniform session creation (`virtualBoxManager.mgr.getSessionObject()`).

In case you want to use the glue layer with a different Python installation, use these steps in a shell to add the necessary files:

```
    # cd VBOX_INSTALL_PATH/sdk/installer
    # PYTHON vboxapisetup.py install
```

2.3.3 C++ COM API

C++ is the language that VirtualBox itself is written in, so C++ is the most direct way to use the Main API – but it is not necessarily the easiest, as using COM and XPCOM has its own set of complications.

VirtualBox ships with sample programs that demonstrate how to use the Main API to implement a number of tasks on your host platform. These samples can be found in the `/bindings/xpcom/samples` directory for Linux, Mac OS X and Solaris and `/bindings/mscom/samples` for Windows. The two samples are actually different, because the one for Windows uses native COM, whereas the other uses our XPCOM implementation, as described above.

Since COM and XPCOM are conceptually very similar but vary in the implementation details, we have created a "glue" layer that shields COM client code from these differences. All VirtualBox uses is this glue layer, so the same code written once works on both Windows hosts (with native COM) as well as on other hosts (with our XPCOM implementation). It is recommended to always use this glue code instead of using the COM and XPCOM APIs directly, as it is very easy to make your code completely independent from the platform it is running on.

In order to encapsulate platform differences between Microsoft COM and XPCOM, the following items should be kept in mind when using the glue layer:

1. **Attribute getters and setters.** COM has the notion of "attributes" in interfaces, which roughly compare to C++ member variables in classes. The difference is that for each attribute declared in an interface, COM automatically provides a "get" method to return the attribute's value. Unless the attribute has been marked as "readonly", a "set" attribute is also provided.

   To illustrate, the IVirtualBox interface has a "version" attribute, which is read-only and of the "wstring" type (the standard string type in COM). As a result, you can call the "get" method for this attribute to retrieve the version number of VirtualBox.

   Unfortunately, the implementation differs between COM and XPCOM. Microsoft COM names the "get" method like this: `get_Attribute()`, whereas XPCOM uses this syntax: `GetAttribute()` (and accordingly for "set" methods). To hide these differences, the VirtualBox glue code provides the `COMGETTER(attrib)` and `COMSETTER(attrib)` macros. So, `COMGETTER(version)()` (note, two pairs of brackets) expands to `get_Version()` on Windows and `GetVersion()` on other platforms.

2. **Unicode conversions.** While the rest of the modern world has pretty much settled on encoding strings in UTF-8, COM, unfortunately, uses UCS-16 encoding. This requires a lot of conversions, in particular between the VirtualBox Main API and the Qt GUI, which, like the rest of Qt, likes to use UTF-8.

   To facilitate these conversions, VirtualBox provides the `com::Bstr` and `com::Utf8Str` classes, which support all kinds of conversions back and forth.

3. **COM autopointers.** Possibly the greatest pain of using COM – reference counting – is alleviated by the `ComPtr<>` template provided by the `ptr.h` file in the glue layer.

2.3.4 Event queue processing

Both VirtualBox client programs and frontends should periodically perform processing of the main event queue, and do that on the application's main thread. In case of a typical GUI Windows/Mac OS application this happens automatically in the GUI's dispatch loop. However, for CLI only application, the appropriate actions have to be taken. For C++ applications, the VirtualBox SDK provided glue method

```
int EventQueue::processEventQueue(uint32_t cMsTimeout)
```

can be used for both blocking and non-blocking operations. For the Python bindings, a common layer provides the method

```
VirtualBoxManager.waitForEvents(ms)
```

with similar semantics.

Things get somewhat more complicated for situations where an application using VirtualBox cannot directly control the main event loop and the main event queue is separated from the event queue of the programming librarly (for example in case of Qt on Unix platforms). In such a case, the application developer is advised to use a platform/toolkit specific event injection mechanism to force event queue checks either based on periodical timer events delivered to the main thread, or by using custom platform messages to notify the main thread when events are available. See the VBoxSDL and Qt (VirtualBox) frontends as examples.

2.3.5 Visual Basic and Visual Basic Script (VBS) on Windows hosts

On Windows hosts, one can control some of the VirtualBox Main API functionality from VBS scripts, and pretty much everything from Visual Basic programs.[5]

VBS is scripting language available in any recent Windows environment. As an example, the following VBS code will print VirtualBox version:

```
set vb = CreateObject("VirtualBox.VirtualBox")
Wscript.Echo "VirtualBox version " & vb.version
```

See `bindings/mscom/vbs/sample/vboxinfo.vbs` for the complete sample.

Visual Basic is a popular high level language capable of accessing COM objects. The following VB code will iterate over all available virtual machines:

```
Dim vb As VirtualBox.IVirtualBox

vb = CreateObject("VirtualBox.VirtualBox")
machines = ""
For Each m In vb.Machines
   m = m & " " & m.Name
Next
```

See `bindings/mscom/vb/sample/vboxinfo.vb` for the complete sample.

2.3.6 C binding to XPCOM API

Note: This section currently applies to Linux hosts only.

Starting with version 2.2, VirtualBox offers a C binding for the XPCOM API.

The C binding provides a layer enabling object creation, method invocation and attribute access from C.

[5] The difference results from the way VBS treats COM safearrays, which are used to keep lists in the Main API. VBS expects every array element to be a `VARIANT`, which is too strict a limitation for any high performance API. We may lift this restriction for interface APIs in a future version, or alternatively provide conversion APIs.

2.3.6.1 Getting started

The following sections describe how to use the C binding in a C program.

For Linux, a sample program is provided which demonstrates use of the C binding to initialize XPCOM, get handles for VirtualBox and Session objects, make calls to list and start virtual machines, and uninitialize resources when done. The program uses the VBoxGlue library to open the C binding layer during runtime.

The sample program `tstXPCOMCGlue` is located in the bin directory and can be run without arguments. It lists registered machines on the host along with some additional information and ask for a machine to start. The source for this program is available in `sdk/bindings/xpcom/cbinding/samples/` directory. The source for the VBoxGlue library is available in the `sdk/bindings/xpcom/cbinding/` directory.

2.3.6.2 XPCOM initialization

Just like in C++, XPCOM needs to be initialized before it can be used. The `VBoxCAPI_v2_5.h` header provides the interface to the C binding. Here's how to initialize XPCOM:

```
#include "VBoxCAPI_v2_5.h"
...
PCVBOXXPCOM g_pVBoxFuncs = NULL;
IVirtualBox *vbox        = NULL;
ISession *session        = NULL;

/*
 * VBoxGetXPCOMCFunctions() is the only function exported by
 * VBoxXPCOMC.so and the only one needed to make virtualbox
 * work with C. This functions gives you the pointer to the
 * function table (g_pVBoxFuncs).
 *
 * Once you get the function table, then how and which functions
 * to use is explained below.
 *
 * g_pVBoxFuncs->pfnComInitialize does all the necessary startup
 * action and provides us with pointers to vbox and session handles.
 * It should be matched by a call to g_pVBoxFuncs->pfnComUninitialize()
 * when done.
 */

g_pVBoxFuncs = VBoxGetXPCOMCFunctions(VBOX_XPCOMC_VERSION);
g_pVBoxFuncs->pfnComInitialize(&vbox, &session);
```

If either `vbox` or `session` is still `NULL`, initialization failed and the XPCOM API cannot be used.

2.3.6.3 XPCOM method invocation

Method invocation is straightforward. It looks pretty much like the C++ way, augmented with an extra indirection due to accessing the vtable and passing a pointer to the object as the first argument to serve as the `this` pointer.

Using the C binding, all method invocations return a numeric result code.

If an interface is specified as returning an object, a pointer to a pointer to the appropriate object must be passed as the last argument. The method will then store an object pointer in that location.

In other words, to call an object's method what you need is

```
IObject *object;
nsresult rc;
...
/*
 * Calling void IObject::method(arg, ...)
 */
```

```
rc = object->vtbl->Method(object, arg, ...);

...
IFoo *foo;
/*
 * Calling IFoo IObject::method(arg, ...)
 */
rc = object->vtbl->Method(object, args, ..., &foo);
```

As a real-world example of a method invocation, let's call IMachine::launchVMProcess which returns an IProgress object. Note again that the method name is capitalized.

```
IProgress *progress;
...
rc = vbox->vtbl->LaunchVMProcess(
    machine,       /* this  */
    session,       /* arg 1 */
    sessionType,   /* arg 2 */
    env,           /* arg 3 */
    &progress      /* Out   */
);
```

2.3.6.4 XPCOM attribute access

A construct similar to calling non-void methods is used to access object attributes. For each attribute there exists a getter method, the name of which is composed of `Get` followed by the capitalized attribute name. Unless the attribute is read-only, an analogous `Set` method exists. Let's apply these rules to read the IVirtualBox::revision attribute.

Using the `IVirtualBox` handle `vbox` obtained above, calling its `GetRevision` method looks like this:

```
PRUint32   rev;

rc = vbox->vtbl->GetRevision(vbox, &rev);
if (NS_SUCCEEDED(rc))
{
    printf("Revision: %u\n", (unsigned)rev);
}
```

All objects with their methods and attributes are documented in chapter 5, *Classes (interfaces)*, page 43.

2.3.6.5 String handling

When dealing with strings you have to be aware of a string's encoding and ownership.

Internally, XPCOM uses UTF-16 encoded strings. A set of conversion functions is provided to convert other encodings to and from UTF-16. The type of a UTF-16 character is `PRUnichar`. Strings of UTF-16 characters are arrays of that type. Most string handling functions take pointers to that type. Prototypes for the following conversion functions are declared in `VBoxCAPI_v2_5.h`.

Conversion of UTF-16 to and from UTF-8

```
int (*pfnUtf16ToUtf8)(const PRUnichar *pwszString, char **ppszString);
int (*pfnUtf8ToUtf16)(const char *pszString, PRUnichar **ppwszString);
```

Ownership The ownership of a string determines who is responsible for releasing resources associated with the string. Whenever XPCOM creates a string, ownership is transferred to the caller. To avoid resource leaks, the caller should release resources once the string is no longer needed.

2.3.6.6 XPCOM uninitialization

Uninitialization is performed by `g_pVBoxFuncs->pfnComUninitialize()`. If your program can exit from more than one place, it is a good idea to install this function as an exit handler with Standard C's `atexit()` just after calling `g_pVBoxFuncs->pfnComInitialize()` , e.g.

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

...

/*
 * Make sure g_pVBoxFuncs->pfnComUninitialize() is called at exit, no
 * matter if we return from the initial call to main or call exit()
 * somewhere else. Note that atexit registered functions are not
 * called upon abnormal termination, i.e. when calling abort() or
 * signal(). Separate provisions must be taken for these cases.
 */

if (atexit(g_pVBoxFuncs->pfnComUninitialize()) != 0) {
    fprintf(stderr, "failed to register g_pVBoxFuncs->pfnComUninitialize()\n");
    exit(EXIT_FAILURE);
}
```

Another idea would be to write your own `void myexit(int status)` function, calling `g_pVBoxFuncs->pfnComUninitialize()` followed by the real `exit()`, and use it instead of `exit()` throughout your program and at the end of `main`.

If you expect the program to be terminated by a signal (e.g. user types CTRL-C sending SIGINT) you might want to install a signal handler setting a flag noting that a signal was sent and then calling `g_pVBoxFuncs->pfnComUninitialize()` later on (usually *not* from the handler itself .)

That said, if a client program forgets to call `g_pVBoxFuncs->pfnComUninitialize()` before it terminates, there is a mechanism in place which will eventually release references held by the client. You should not rely on this, however.

2.3.6.7 Compiling and linking

A program using the C binding has to open the library during runtime using the help of glue code provided and as shown in the example `tstXPCOMCGlue.c`. Compilation and linking can be achieved, e.g., with a makefile fragment similar to

```
# Where is the XPCOM include directory?
INCS_XPCOM    = -I../../include
# Where is the glue code directory?
GLUE_DIR      = ..
GLUE_INC      = -I..

#Compile Glue Library
VBoxXPCOMCGlue.o: $(GLUE_DIR)/VBoxXPCOMCGlue.c
    $(CC) $(CFLAGS) $(INCS_XPCOM) $(GLUE_INC) -o $@ -c $<

# Compile.
program.o: program.c VBoxCAPI_v2_5.h
    $(CC) $(CFLAGS) $(INCS_XPCOM) $(GLUE_INC) -o $@ -c $<
```

```
# Link.
program: program.o VBoxXPCOMCGlue.o
    $(CC) -o $@ $^ -ldl
```

3 Basic VirtualBox concepts; some examples

The following explains some basic VirtualBox concepts such as the VirtualBox object, sessions and how virtual machines are manipulated and launched using the Main API. The coding examples use a pseudo-code style closely related to the object-oriented web service (OOWS) for JAX-WS. Depending on which environment you are using, you will need to adjust the examples.

3.1 Obtaining basic machine information. Reading attributes

Any program using the Main API will first need access to the global VirtualBox object (see IVirtualBox), from which all other functionality of the API is derived. With the OOWS for JAX-WS, this is returned from the IWebsessionManager::logon() call.

To enumerate virtual machines, one would look at the "machines" array attribute in the VirtualBox object (see IVirtualBox::machines). This array contains all virtual machines currently registered with the host, each of them being an instance of IMachine. From each such instance, one can query additional information, such as the UUID, the name, memory, operating system and more by looking at the attributes; see the attributes list in IMachine documentation.

As mentioned in the preceding chapters, depending on your programming environment, attributes are mapped to corresponding "get" and (if the attribute is not read-only) "set" methods. So when the documentation says that IMachine has a "name" attribute, this means you need to code something like the following to get the machine's name:

```
IMachine machine = ...;
String name = machine.getName();
```

Boolean attribute getters can sometimes be called `isAttribute()` due to JAX-WS naming conventions.

3.2 Changing machine settings. Sessions

As said in the previous section, to read a machine's attribute, one invokes the corresponding "get" method. One would think that to change settings of a machine, it would suffice to call the corresponding "set" method – for example, to set a VM's memory to 1024 MB, one would call `setMemorySize(1024)`. Try that, and you will get an error: "The machine is not mutable."

So unfortunately, things are not that easy. VirtualBox is a complicated environment in which multiple processes compete for possibly the same resources, especially machine settings. As a result, machines must be "locked" before they can either be modified or started. This is to prevent multiple processes from making conflicting changes to a machine: it should, for example, not be allowed to change the memory size of a virtual machine while it is running. (You can't add more memory to a real computer while it is running either, at least not to an ordinary PC.) Also, two processes must not change settings at the same time, or start a machine at the same time.

These requirements are implemented in the Main API by way of "sessions", in particular, the ISession interface. Each process which talks to VirtualBox needs its own instance of ISession. In the web service, you cannot create such an object, but `vboxwebsrv` creates one for you when you log on, which you can obtain by calling IWebsessionManager::getSessionObject().

This session object must then be used like a mutex semaphore in common programming environments. Before you can change machine settings, you must write-lock the machine by calling IMachine::lockMachine() with your process's session object.

After the machine has been locked, the ISession::machine attribute contains a copy of the original IMachine object upon which the session was opened, but this copy is "mutable": you can invoke "set" methods on it.

When done making the changes to the machine, you must call IMachine::saveSettings(), which will copy the changes you have made from your "mutable" machine back to the real machine and write them out to the machine settings XML file. This will make your changes permanent.

Finally, it is important to always unlock the machine again, by calling ISession::unlockMachine(). Otherwise, when the calling process end, the machine will receive the "aborted" state, which can lead to loss of data.

So, as an example, the sequence to change a machine's memory to 1024 MB is something like this:

```
IWebsessionManager mgr ...;
IVirtualBox vbox = mgr.logon(user, pass);
...
IMachine machine = ...;  // read-only machine
ISession session = mgr.getSessionObject();
machine.lockMachine(session, LockType.Write); // machine is now locked for writing
IMachine mutable = session.getMachine();      // obtain the mutable machine copy
mutable.setMemorySize(1024);
mutable.saveSettings();                       // write settings to XML
session.unlockMachine();
```

3.3 Launching virtual machines

To launch a virtual machine, you call IMachine::launchVMProcess(). In doing so, the caller instructs the VirtualBox engine to start a new process with the virtual machine in it, since to the host, each virtual machine looks like a single process, even if it has hundreds of its own processes inside. (This new VM process in turn obtains a write lock on the machine, as described above, to prevent conflicting changes from other processes; this is why opening another session will fail while the VM is running.)

Starting a machine looks something like this:

```
IWebsessionManager mgr ...;
IVirtualBox vbox = mgr.logon(user, pass);
...
IMachine machine = ...;  // read-only machine
ISession session = mgr.getSessionObject();
IProgress prog = machine.launchVMProcess(session,
                                         "gui",  // session type
                                         "");    // possibly environment setting
prog.waitForCompletion(10000);  // give the process 10 secs
if (prog.getResultCode() != 0)  // check success
    System.out.println("Cannot launch VM!")
```

The caller's session object can then be used as a sort of remote control to the VM process that was launched. It contains a "console" object (see ISession::console) with which the VM can be paused, stopped, snapshotted or other things.

3.4 VirtualBox events

In VirtualBox, "events" provide a uniform mechanism to register for and consume specific events. A VirtualBox client can register an "event listener" (represented by the IEventListener interface),

which will then get notified by the server when an event (represented by the IEvent interface) happens.

The IEvent interface is an abstract parent interface for all events that can occur in VirtualBox. The actual events that the server sends out are then of one of the specific subclasses, for example IMachineStateChangedEvent or IMediumChangedEvent.

As an example, the VirtualBox GUI waits for machine events and can thus update its display when the machine state changes or machine settings are modified, even if this happens in another client. This is how the GUI can automatically refresh its display even if you manipulate a machine from another client, for example, from VBoxManage.

To register an event listener to listen to events, use code like this:

```
EventSource es = console.getEventSource();
IEventListener listener = es.createListener();
VBoxEventType aTypes[] = (VBoxEventType.OnMachineStateChanged);
        // list of event types to listen for
es.registerListener(listener, aTypes, false /* active */);
        // register passive listener
IEvent ev = es.getEvent(listener, 1000);
        // wait up to one second for event to happen
if (ev != null)
{
    // downcast to specific event interface (in this case we have only registered
    // for one type, otherwise IEvent::type would tell us)
    IMachineStateChangedEvent mcse = IMachineStateChangedEvent.queryInterface(ev);
    ... // inspect and do something
    es.eventProcessed(listener, ev);
}
...
es.unregisterListener(listener);
```

A graphical user interface would probably best start its own thread to wait for events and then process these in a loop.

The events mechanism was introduced with VirtualBox 3.3 and replaces various callback interfaces which were called for each event in the interface. The callback mechanism was not compatible with scripting languages, local Java bindings and remote web services as they do not support callbacks. The new mechanism with events and event listeners works with all of these.

To simplify developement of application using events, concept of event aggregator was introduced. Essentially it's mechanism to aggregate multiple event sources into single one, and then work with this single aggregated event source instead of original sources. As an example, one can evaluate demo recorder in VirtualBox Python shell, shipped with SDK - it records mouse and keyboard events, represented as separate event sources. Code is essentially like this:

```
listener = console.eventSource.createListener()
agg = console.eventSource.createAggregator([console.keyboard.eventSource, console.mouse.eventSource])
agg.registerListener(listener, [ctx['global'].constants.VBoxEventType_Any], False)
registered = True
end = time.time() + dur
while  time.time() < end:
    ev = agg.getEvent(listener, 1000)
    processEent(ev)
agg.unregisterListener(listener)
```

Without using aggregators consumer have to poll on both sources, or start multiple threads to block on those sources.

4 The VirtualBox shell

VirtualBox comes with an extensible shell, which allows you to control your virtual machines from the command line. It is also a nontrivial example of how to use the VirtualBox APIs from Python, for all three COM/XPCOM/WS styles of the API.

You can easily extend this shell with your own commands. Create a subdirectory named `.VirtualBox/shexts` below your home directory and put a Python file implementing your shell extension commands in this directory. This file must contain an array named `commands` containing your command definitions:

```
commands = {
'cmd1': ['Command cmd1 help', cmd1],
'cmd2': ['Command cmd2 help', cmd2]
}
```

For example, to create a command for creating hard drive images, the following code can be used:

```
def createHdd(ctx,args):
  # Show some meaningful error message on wrong input
  if (len(args) < 3):
    print "usage: createHdd sizeM location type"
    return 0

  # Get arguments
  size = int(args[1])
  loc = args[2]
  if len(args) > 3:
    format = args[3]
  else:
    # And provide some meaningful defaults
    format = "vdi"

# Call VirtualBox API, using context's fields
hdd = ctx['vb'].createHardDisk(format, loc)
# Access constants using ctx['global'].constants
progress = hdd.createBaseStorage(size, ctx['global'].constants.HardDiskVariant_Standard)
# use standard progress bar mechanism
ctx['progressBar'](progress)


# Report errors
if not hdd.id:
   print "cannot create disk (file %s exist?)" %(loc)
   return 0

# Give user some feedback on success too
print "created HDD with id: %s" %(hdd.id)

# 0 means continue execution, other values mean exit from the interpreter
return 0

commands = {
  'myCreateHDD': ['Create virtual HDD, createHdd size location type', createHdd]
  }
```

Just store the above text in the file `createHdd` (or any other meaningful name) in `.VirtualBox/shexts/`. Start the VirtualBox shell, or just issue the `reloadExts` command, if the shell is already running. Your new command will now be available.

5 Classes (interfaces)

5.1 IAdditionsStateChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a Guest Additions property changes. Interested callees should query IGuest attributes to find out what has changed.

5.2 IAppliance

Represents a platform-independent appliance in OVF format. An instance of this is returned by IVirtualBox::createAppliance(), which can then be used to import and export virtual machines within an appliance with VirtualBox.

The OVF standard suggests two different physical file formats:

1. If the appliance is distributed as a set of files, there must be at least one XML descriptor file that conforms to the OVF standard and carries an `.ovf` file extension. If this descriptor file references other files such as disk images, as OVF appliances typically do, those additional files must be in the same directory as the descriptor file.

2. If the appliance is distributed as a single file, it must be in TAR format and have the `.ova` file extension. This TAR file must then contain at least the OVF descriptor files and optionally other files.

   At this time, VirtualBox does not not yet support the packed (TAR) variant; support will be added with a later version.

Importing an OVF appliance into VirtualBox as instances of IMachine involves the following sequence of API calls:

1. Call IVirtualBox::createAppliance(). This will create an empty IAppliance object.

2. On the new object, call read() with the full path of the OVF file you would like to import. So long as this file is syntactically valid, this will succeed and fill the appliance object with the parsed data from the OVF file.

3. Next, call interpret(), which analyzes the OVF data and sets up the contents of the IAppliance attributes accordingly. These can be inspected by a VirtualBox front-end such as the GUI, and the suggestions can be displayed to the user. In particular, the virtualSystemDescriptions[] array contains instances of IVirtualSystemDescription which represent the virtual systems (machines) in the OVF, which in turn describe the virtual hardware prescribed by the OVF (network and hardware adapters, virtual disk images, memory size and so on). The GUI can then give the user the option to confirm and/or change these suggestions.

4. If desired, call IVirtualSystemDescription::setFinalValues() for each virtual system (machine) to override the suggestions made by the interpret() routine.

5. Finally, call importMachines() to create virtual machines in VirtualBox as instances of IMachine that match the information in the virtual system descriptions. After this call succeeded, the UUIDs of the machines created can be found in the machines[] array attribute.

Exporting VirtualBox machines into an OVF appliance involves the following steps:

1. As with importing, first call IVirtualBox::createAppliance() to create an empty IAppliance object.
2. For each machine you would like to export, call IMachine::export() with the IAppliance object you just created. Each such call creates one instance of IVirtualSystemDescription inside the appliance.
3. If desired, call IVirtualSystemDescription::setFinalValues() for each virtual system (machine) to override the suggestions made by the export() routine.
4. Finally, call write() with a path specification to have the OVF file written.

5.2.1 Attributes

5.2.1.1 path (read-only)

`wstring IAppliance::path`

Path to the main file of the OVF appliance, which is either the `.ovf` or the `.ova` file passed to read() (for import) or write() (for export). This attribute is empty until one of these methods has been called.

5.2.1.2 disks (read-only)

`wstring IAppliance::disks[]`

Array of virtual disk definitions. One such description exists for each disk definition in the OVF; each string array item represents one such piece of disk information, with the information fields separated by tab (\t) characters.

The caller should be prepared for additional fields being appended to this string in future versions of VirtualBox and therefore check for the number of tabs in the strings returned.

In the current version, the following eight fields are returned per string in the array:

1. Disk ID (unique string identifier given to disk)
2. Capacity (unsigned integer indicating the maximum capacity of the disk)
3. Populated size (optional unsigned integer indicating the current size of the disk; can be approximate; -1 if unspecified)
4. Format (string identifying the disk format, typically "http://www.vmware.com/specifications/vmdk.html#spars
5. Reference (where to find the disk image, typically a file name; if empty, then the disk should be created on import)
6. Image size (optional unsigned integer indicating the size of the image, which need not necessarily be the same as the values specified above, since the image may be compressed or sparse; -1 if not specified)
7. Chunk size (optional unsigned integer if the image is split into chunks; presently unsupported and always -1)
8. Compression (optional string equalling "gzip" if the image is gzip-compressed)

5.2.1.3 virtualSystemDescriptions (read-only)

```
IVirtualSystemDescription IAppliance::virtualSystemDescriptions[]
```

Array of virtual system descriptions. One such description is created for each virtual system (machine) found in the OVF. This array is empty until either interpret() (for import) or IMachine::export() (for export) has been called.

5.2.1.4 machines (read-only)

```
wstring IAppliance::machines[]
```

Contains the UUIDs of the machines created from the information in this appliances. This is only relevant for the import case, and will only contain data after a call to importMachines() succeeded.

5.2.2 createVFSExplorer

```
IVFSExplorer IAppliance::createVFSExplorer(
           [in] wstring aUri)
```

aUri The URI describing the file system to use.

Returns a IVFSExplorer object for the given URI.

5.2.3 getWarnings

```
wstring[] IAppliance::getWarnings()
```

Returns textual warnings which occurred during execution of interpret().

5.2.4 importMachines

```
IProgress IAppliance::importMachines()
```

Imports the appliance into VirtualBox by creating instances of IMachine and other interfaces that match the information contained in the appliance as closely as possible, as represented by the import instructions in the virtualSystemDescriptions[] array.

Calling this method is the final step of importing an appliance into VirtualBox; see IAppliance for an overview.

Since importing the appliance will most probably involve copying and converting disk images, which can take a long time, this method operates asynchronously and returns an IProgress object to allow the caller to monitor the progress.

After the import succeeded, the UUIDs of the IMachine instances created can be retrieved from the machines[] array attribute.

5.2.5 interpret

```
void IAppliance::interpret()
```

Interprets the OVF data that was read when the appliance was constructed. After calling this method, one can inspect the virtualSystemDescriptions[] array attribute, which will then contain one IVirtualSystemDescription for each virtual machine found in the appliance.

Calling this method is the second step of importing an appliance into VirtualBox; see IAppliance for an overview.

After calling this method, one should call getWarnings() to find out if problems were encountered during the processing which might later lead to errors.

5.2.6 read

```
IProgress IAppliance::read(
            [in] wstring file)
```

file Name of appliance file to open (either with an `.ovf` or `.ova` extension, depending on whether the appliance is distributed as a set of files or as a single file, respectively).

Reads an OVF file into the appliance object.

This method succeeds if the OVF is syntactically valid and, by itself, without errors. The mere fact that this method returns successfully does not mean that VirtualBox supports all features requested by the appliance; this can only be examined after a call to interpret().

5.2.7 write

```
IProgress IAppliance::write(
            [in] wstring format,
            [in] boolean manifest,
            [in] wstring path)
```

format Output format, as a string. Currently supported formats are “ovf-0.9” and “ovf-1.0”; future versions of VirtualBox may support additional formats.

manifest Indicate if the optional manifest file (.mf) should be written. The manifest file is used for integrity checks prior import.

path Name of appliance file to open (either with an `.ovf` or `.ova` extension, depending on whether the appliance is distributed as a set of files or as a single file, respectively).

Writes the contents of the appliance exports into a new OVF file.

Calling this method is the final step of exporting an appliance from VirtualBox; see IAppliance for an overview.

Since exporting the appliance will most probably involve copying and converting disk images, which can take a long time, this method operates asynchronously and returns an IProgress object to allow the caller to monitor the progress.

5.3 IAudioAdapter

The IAudioAdapter interface represents the virtual audio adapter of the virtual machine. Used in IMachine::audioAdapter.

5.3.1 Attributes

5.3.1.1 enabled (read/write)

```
boolean IAudioAdapter::enabled
```

Flag whether the audio adapter is present in the guest system. If disabled, the virtual guest hardware will not contain any audio adapter. Can only be changed when the VM is not running.

5.3.1.2 audioController (read/write)

```
AudioControllerType IAudioAdapter::audioController
```

The audio hardware we emulate.

5.3.1.3 audioDriver (read/write)

`AudioDriverType IAudioAdapter::audioDriver`

Audio driver the adapter is connected to. This setting can only be changed when the VM is not running.

5.4 IBIOSSettings

The IBIOSSettings interface represents BIOS settings of the virtual machine. This is used only in the IMachine::BIOSSettings attribute.

5.4.1 Attributes

5.4.1.1 logoFadeIn (read/write)

`boolean IBIOSSettings::logoFadeIn`

Fade in flag for BIOS logo animation.

5.4.1.2 logoFadeOut (read/write)

`boolean IBIOSSettings::logoFadeOut`

Fade out flag for BIOS logo animation.

5.4.1.3 logoDisplayTime (read/write)

`unsigned long IBIOSSettings::logoDisplayTime`

BIOS logo display time in milliseconds (0 = default).

5.4.1.4 logoImagePath (read/write)

`wstring IBIOSSettings::logoImagePath`

Local file system path for external BIOS splash image. Empty string means the default image is shown on boot.

5.4.1.5 bootMenuMode (read/write)

`BIOSBootMenuMode IBIOSSettings::bootMenuMode`

Mode of the BIOS boot device menu.

5.4.1.6 ACPIEnabled (read/write)

`boolean IBIOSSettings::ACPIEnabled`

ACPI support flag.

5.4.1.7 IOAPICEnabled (read/write)

`boolean IBIOSSettings::IOAPICEnabled`

IO APIC support flag. If set, VirtualBox will provide an IO APIC and support IRQs above 15.

5.4.1.8 timeOffset (read/write)

```
long long IBIOSSettings::timeOffset
```

Offset in milliseconds from the host system time. This allows for guests running with a different system date/time than the host. It is equivalent to setting the system date/time in the BIOS except it is not an absolute value but a relative one. Guest Additions time synchronization honors this offset.

5.4.1.9 PXEDebugEnabled (read/write)

```
boolean IBIOSSettings::PXEDebugEnabled
```

PXE debug logging flag. If set, VirtualBox will write extensive PXE trace information to the release log.

5.5 IBandwidthControl

Controls the bandwidth groups of one machine used to cap I/O done by a VM. This includes network and disk I/O.

5.5.1 Attributes

5.5.1.1 numGroups (read-only)

```
unsigned long IBandwidthControl::numGroups
```

The current number of existing bandwidth groups managed.

5.5.2 CreateBandwidthGroup

```
void IBandwidthControl::CreateBandwidthGroup(
            [in] wstring name,
            [in] BandwidthGroupType type,
            [in] unsigned long maxMbPerSec)
```

name Name of the bandwidth group.

type The type of the bandwidth group (network or disk).

maxMbPerSec The maximum number of MBytes which can be transfered by all entities attached to this group during one second.

Creates a new bandwidth group.

5.5.3 DeleteBandwidthGroup

```
void IBandwidthControl::DeleteBandwidthGroup(
            [in] wstring name)
```

name Name of the bandwidth group to delete.

Deletes a new bandwidth group.

5.5.4 GetAllBandwidthGroups

```
IBandwidthGroup[] IBandwidthControl::GetAllBandwidthGroups()
```

Get all managed bandwidth groups.

5.5.5 GetBandwidthGroup

```
IBandwidthGroup IBandwidthControl::GetBandwidthGroup(
            [in] wstring name)
```

name Name of the bandwidth group to get.

Get a bandwidth group by name.

5.6 IBandwidthGroup

Represents one bandwidth group.

5.6.1 Attributes

5.6.1.1 name (read-only)

```
wstring IBandwidthGroup::name
```

Name of the group.

5.6.1.2 type (read-only)

```
BandwidthGroupType IBandwidthGroup::type
```

Type of the group.

5.6.1.3 reference (read-only)

```
unsigned long IBandwidthGroup::reference
```

How many devices/medium attachements use this group.

5.6.1.4 maxMbPerSec (read/write)

```
unsigned long IBandwidthGroup::maxMbPerSec
```

The maximum number of MBytes which can be transfered by all entities attached to this group during one second.

5.7 IBandwidthGroupChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when one of the bandwidth groups changed

5.7.1 Attributes

5.7.1.1 bandwidthGroup (read-only)

`IBandwidthGroup IBandwidthGroupChangedEvent::bandwidthGroup`

The changed bandwidth group.

5.8 ICPUChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a CPU changes.

5.8.1 Attributes

5.8.1.1 cpu (read-only)

`unsigned long ICPUChangedEvent::cpu`

The CPU which changed.

5.8.1.2 add (read-only)

`boolean ICPUChangedEvent::add`

Flag whether the CPU was added or removed.

5.9 ICPUExecutionCapChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when the CPU execution cap changes.

5.9.1 Attributes

5.9.1.1 executionCap (read-only)

`unsigned long ICPUExecutionCapChangedEvent::executionCap`

The new CPU execution cap value. (1-100)

5.10 ICanShowWindowEvent (IVetoEvent)

> **Note:** This interface extends IVetoEvent and therefore supports all its methods and attributes as well.

Notification when a call to IMachine::canShowConsoleWindow() is made by a front-end to check if a subsequent call to IMachine::showConsoleWindow() can succeed.

The callee should give an answer appropriate to the current machine state using event veto. This answer must remain valid at least until the next machine state change.

5.11 IConsole

The IConsole interface represents an interface to control virtual machine execution.

A console object gets created when a machine has been locked for a particular session (client process) using IMachine::lockMachine() or IMachine::launchVMProcess(). The console object can then be found in the session's ISession::console attribute.

Methods of the IConsole interface allow the caller to query the current virtual machine execution state, pause the machine or power it down, save the machine state or take a snapshot, attach and detach removable media and so on.

See also: ISession

5.11.1 Attributes

5.11.1.1 machine (read-only)

`IMachine IConsole::machine`

Machine object for this console session.

> **Note:** This is a convenience property, it has the same value as ISession::machine of the corresponding session object.

5.11.1.2 state (read-only)

`MachineState IConsole::state`

Current execution state of the machine.

> **Note:** This property always returns the same value as the corresponding property of the IMachine object for this console session. For the process that owns (executes) the VM, this is the preferable way of querying the VM state, because no IPC calls are made.

5.11.1.3 guest (read-only)

`IGuest IConsole::guest`

Guest object.

5.11.1.4 keyboard (read-only)

`IKeyboard IConsole::keyboard`

Virtual keyboard object.

> **Note:** If the machine is not running, any attempt to use the returned object will result in an error.

5.11.1.5 mouse (read-only)

`IMouse IConsole::mouse`

Virtual mouse object.

> **Note:** If the machine is not running, any attempt to use the returned object will result in an error.

5.11.1.6 display (read-only)

`IDisplay IConsole::display`

Virtual display object.

> **Note:** If the machine is not running, any attempt to use the returned object will result in an error.

5.11.1.7 debugger (read-only)

`IMachineDebugger IConsole::debugger`

> **Note:** This attribute is not supported in the web service.

Debugging interface.

5.11.1.8 USBDevices (read-only)

`IUSBDevice IConsole::USBDevices[]`

Collection of USB devices currently attached to the virtual USB controller.

> **Note:** The collection is empty if the machine is not running.

5.11.1.9 remoteUSBDevices (read-only)

`IHostUSBDevice IConsole::remoteUSBDevices[]`

List of USB devices currently attached to the remote VRDE client. Once a new device is physically attached to the remote host computer, it appears in this list and remains there until detached.

5.11.1.10 sharedFolders (read-only)

`ISharedFolder IConsole::sharedFolders[]`

Collection of shared folders for the current session. These folders are called transient shared folders because they are available to the guest OS running inside the associated virtual machine only for the duration of the session (as opposed to IMachine::sharedFolders[] which represent permanent shared folders). When the session is closed (e.g. the machine is powered down), these folders are automatically discarded.

New shared folders are added to the collection using createSharedFolder(). Existing shared folders can be removed using removeSharedFolder().

5.11.1.11 VRDEServerInfo (read-only)

`IVRDEServerInfo IConsole::VRDEServerInfo`

Interface that provides information on Remote Desktop Extension (VRDE) connection.

5.11.1.12 eventSource (read-only)

`IEventSource IConsole::eventSource`

Event source for console events.

5.11.1.13 attachedPciDevices (read-only)

`IPciDeviceAttachment IConsole::attachedPciDevices[]`

Array of PCI devices attached to this machine.

5.11.2 adoptSavedState

```
void IConsole::adoptSavedState(
           [in] wstring savedStateFile)
```

savedStateFile Path to the saved state file to adopt.

Associates the given saved state file to the virtual machine.

On success, the machine will go to the Saved state. Next time it is powered up, it will be restored from the adopted saved state and continue execution from the place where the saved state file was created.

The specified saved state file path may be absolute or relative to the folder the VM normally saves the state to (usually, IMachine::snapshotFolder).

> **Note:** It's a caller's responsibility to make sure the given saved state file is compatible with the settings of this virtual machine that represent its virtual hardware (memory size, storage disk configuration etc.). If there is a mismatch, the behavior of the virtual machine is undefined.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Virtual machine state neither PoweredOff nor Aborted.

5.11.3 attachUSBDevice

```
void IConsole::attachUSBDevice(
           [in] uuid id)
```

id UUID of the host USB device to attach.

Attaches a host USB device with the given UUID to the USB controller of the virtual machine.

The device needs to be in one of the following states: Busy, Available or Held, otherwise an error is immediately returned.

When the device state is Busy, an error may also be returned if the host computer refuses to release it for some reason.

See also: IUSBController::deviceFilters, USBDeviceState

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Virtual machine state neither Running nor Paused.
- `VBOX_E_PDM_ERROR`: Virtual machine does not have a USB controller.

5.11.4 createSharedFolder

```
void IConsole::createSharedFolder(
           [in] wstring name,
           [in] wstring hostPath,
           [in] boolean writable,
           [in] boolean automount)
```

name Unique logical name of the shared folder.

hostPath Full path to the shared folder in the host file system.

writable Whether the share is writable or readonly

automount Whether the share gets automatically mounted by the guest or not.

Creates a transient new shared folder by associating the given logical name with the given host path, adds it to the collection of shared folders and starts sharing it. Refer to the description of ISharedFolder to read more about logical names.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine in Saved state or currently changing state.
- VBOX_E_FILE_ERROR: Shared folder already exists or not accessible.

5.11.5 deleteSnapshot

```
IProgress IConsole::deleteSnapshot(
           [in] uuid id)
```

id UUID of the snapshot to delete.

Starts deleting the specified snapshot asynchronously. See ISnapshot for an introduction to snapshots.

The execution state and settings of the associated machine stored in the snapshot will be deleted. The contents of all differencing media of this snapshot will be merged with the contents of their dependent child media to keep the medium chain valid (in other words, all changes represented by media being deleted will be propagated to their child medium). After that, this snapshot's differencing medium will be deleted. The parent of this snapshot will become a new parent for all its child snapshots.

If the deleted snapshot is the current one, its parent snapshot will become a new current snapshot. The current machine state is not directly affected in this case, except that currently attached differencing media based on media of the deleted snapshot will be also merged as described above.

If the deleted snapshot is the first or current snapshot, then the respective IMachine attributes will be adjusted. Deleting the current snapshot will also implicitly call IMachine::saveSettings() to make all current machine settings permanent.

Deleting a snapshot has the following preconditions:

- Child media of all normal media of the deleted snapshot must be accessible (see IMedium::state) for this operation to succeed. In particular, this means that all virtual machines whose media are directly or indirectly based on the media of deleted snapshot must be powered off.
- You cannot delete the snapshot if a medium attached to it has more than once child medium (differencing images) because otherwise merging would be impossible. This might be the case if there is more than one child snapshot or differencing images were created for other reason (e.g. implicitly because of multiple machine attachments).

The virtual machine's state is changed to "DeletingSnapshot", "DeletingSnapshotOnline" or "DeletingSnapshotPaused" while this operation is in progress.

> **Note:** Merging medium contents can be very time and disk space consuming, if these media are big in size and have many children. However, if the snapshot being deleted is the last (head) snapshot on the branch, the operation will be rather quick.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` The running virtual machine prevents deleting this snapshot. This happens only in very specific situations, usually snapshots can be deleted without trouble while a VM is running. The error message text explains the reason for the failure.

5.11.6 detachUSBDevice

```
IUSBDevice IConsole::detachUSBDevice(
           [in] uuid id)
```

id UUID of the USB device to detach.

Detaches an USB device with the given UUID from the USB controller of the virtual machine.

After this method succeeds, the VirtualBox server re-initiates all USB filters as if the device were just physically attached to the host, but filters of this machine are ignored to avoid a possible automatic re-attachment.

See also: IUSBController::deviceFilters, USBDeviceState

If this method fails, the following error codes may be reported:

- `VBOX_E_PDM_ERROR:` Virtual machine does not have a USB controller.
- `E_INVALIDARG:` USB device not attached to this virtual machine.

5.11.7 discardSavedState

```
void IConsole::discardSavedState(
           [in] boolean fRemoveFile)
```

fRemoveFile Whether to also remove the saved state file.

Forcibly resets the machine to "Powered Off" state if it is currently in the "Saved" state (previously created by saveState()). Next time the machine is powered up, a clean boot will occur.

> **Note:** This operation is equivalent to resetting or powering off the machine without doing a proper shutdown of the guest operating system; as with resetting a running phyiscal computer, it can can lead to data loss.

If `fRemoveFile` is `true`, the file in the machine directory into which the machine state was saved is also deleted. If this is `false`, then the state can be recovered and later re-inserted into a machine using adoptSavedState(). The location of the file can be found in the IMachine::stateFilePath attribute.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Virtual machine not in state Saved.

5.11.8 findUSBDeviceByAddress

```
IUSBDevice IConsole::findUSBDeviceByAddress(
           [in] wstring name)
```

name Address of the USB device (as assigned by the host) to search for.

Searches for a USB device with the given host address.
See also: IUSBDevice::address
If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: Given name does not correspond to any USB device.

5.11.9 findUSBDeviceById

```
IUSBDevice IConsole::findUSBDeviceById(
           [in] uuid id)
```

id UUID of the USB device to search for.

Searches for a USB device with the given UUID.
See also: IUSBDevice::id
If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: Given id does not correspond to any USB device.

5.11.10 getDeviceActivity

```
DeviceActivity IConsole::getDeviceActivity(
           [in] DeviceType type)
```

type

Gets the current activity type of a given device or device group.
If this method fails, the following error codes may be reported:

- E_INVALIDARG: Invalid device type.

5.11.11 getGuestEnteredACPIMode

```
boolean IConsole::getGuestEnteredACPIMode()
```

Checks if the guest entered the ACPI mode G0 (working) or G1 (sleeping). If this method returns false, the guest will most likely not respond to external ACPI events.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine not in Running state.

5.11.12 getPowerButtonHandled

```
boolean IConsole::getPowerButtonHandled()
```

Checks if the last power button event was handled by guest.
If this method fails, the following error codes may be reported:

- VBOX_E_PDM_ERROR: Checking if the event was handled by the guest OS failed.

5.11.13 pause

```
void IConsole::pause()
```

Pauses the virtual machine execution.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine not in Running state.
- VBOX_E_VM_ERROR: Virtual machine error in suspend operation.

5.11.14 powerButton

```
void IConsole::powerButton()
```

Sends the ACPI power button event to the guest.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine not in Running state.
- VBOX_E_PDM_ERROR: Controlled power off failed.

5.11.15 powerDown

```
IProgress IConsole::powerDown()
```

Initiates the power down procedure to stop the virtual machine execution.
The completion of the power down procedure is tracked using the returned IProgress object. After the operation is complete, the machine will go to the PoweredOff state.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine must be Running, Paused or Stuck to be powered down.

5.11.16 powerUp

```
IProgress IConsole::powerUp()
```

Starts the virtual machine execution using the current machine state (that is, its current execution state, current settings and current storage devices).

> **Note:** This method is only useful for front-ends that want to actually execute virtual machines in their own process (like the VirtualBox or VBoxSDL front-ends). Unless you are intending to write such a front-end, do not call this method. If you simply want to start virtual machine execution using one of the existing front-ends (for example the VirtualBox GUI or headless server), use IMachine::launchVMProcess() instead; these front-ends will power up the machine automatically for you.

If the machine is powered off or aborted, the execution will start from the beginning (as if the real hardware were just powered on).
If the machine is in the Saved state, it will continue its execution the point where the state has been saved.
If the machine IMachine::teleporterEnabled property is enabled on the machine being powered up, the machine will wait for an incoming teleportation in the TeleportingIn state. The returned progress object will have at least three operations where the last three are defined as:

(1) powering up and starting TCP server, (2) waiting for incoming teleportations, and (3) performing teleportation. These operations will be reflected as the last three operations of the progress objected returned by IMachine::launchVMProcess() as well.

See also: #saveState

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine already running.
- VBOX_E_HOST_ERROR: Host interface does not exist or name not set.
- VBOX_E_FILE_ERROR: Invalid saved state file.

5.11.17 powerUpPaused

```
IProgress IConsole::powerUpPaused()
```

Identical to powerUp except that the VM will enter the Paused state, instead of Running.

See also: #powerUp

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine already running.
- VBOX_E_HOST_ERROR: Host interface does not exist or name not set.
- VBOX_E_FILE_ERROR: Invalid saved state file.

5.11.18 removeSharedFolder

```
void IConsole::removeSharedFolder(
           [in] wstring name)
```

name Logical name of the shared folder to remove.

Removes a transient shared folder with the given name previously created by createSharedFolder() from the collection of shared folders and stops sharing it.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine in Saved state or currently changing state.
- VBOX_E_FILE_ERROR: Shared folder does not exists.

5.11.19 reset

```
void IConsole::reset()
```

Resets the virtual machine.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine not in Running state.
- VBOX_E_VM_ERROR: Virtual machine error in reset operation.

5.11.20 restoreSnapshot

```
IProgress IConsole::restoreSnapshot(
           [in] ISnapshot snapshot)
```

snapshot The snapshot to restore the VM state from.

Starts resetting the machine's current state to the state contained in the given snapshot, asynchronously. All current settings of the machine will be reset and changes stored in differencing media will be lost. See ISnapshot for an introduction to snapshots.

After this operation is successfully completed, new empty differencing media are created for all normal media of the machine.

If the given snapshot is an online snapshot, the machine will go to the Saved, so that the next time it is powered on, the execution state will be restored from the state of the snapshot.

> **Note:** The machine must not be running, otherwise the operation will fail.

> **Note:** If the machine state is Saved prior to this operation, the saved state file will be implicitly deleted (as if discardSavedState() were called).

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Virtual machine is running.

5.11.21 resume

```
void IConsole::resume()
```

Resumes the virtual machine execution.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Virtual machine not in Paused state.
- `VBOX_E_VM_ERROR:` Virtual machine error in resume operation.

5.11.22 saveState

```
IProgress IConsole::saveState()
```

Saves the current execution state of a running virtual machine and stops its execution.

After this operation completes, the machine will go to the Saved state. Next time it is powered up, this state will be restored and the machine will continue its execution from the place where it was saved.

This operation differs from taking a snapshot to the effect that it doesn't create new differencing media. Also, once the machine is powered up from the state saved using this method, the saved state is deleted, so it will be impossible to return to this state later.

> **Note:** On success, this method implicitly calls IMachine::saveSettings() to save all current machine settings (including runtime changes to the DVD medium, etc.). Together with the impossibility to change any VM settings when it is in the Saved state, this guarantees adequate hardware configuration of the machine when it is restored from the saved state file.

> **Note:** The machine must be in the Running or Paused state, otherwise the operation will fail.

See also: takeSnapshot()
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine state neither Running nor Paused.
- VBOX_E_FILE_ERROR: Failed to create directory for saved state file.

5.11.23 sleepButton

```
void IConsole::sleepButton()
```

Sends the ACPI sleep button event to the guest.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine not in Running state.
- VBOX_E_PDM_ERROR: Sending sleep button event failed.

5.11.24 takeSnapshot

```
IProgress IConsole::takeSnapshot(
           [in] wstring name,
           [in] wstring description)
```

name Short name for the snapshot.

description Optional description of the snapshot.

Saves the current execution state and all settings of the machine and creates differencing images for all normal (non-independent) media. See ISnapshot for an introduction to snapshots.
This method can be called for a PoweredOff, Saved (see saveState()), Running or Paused virtual machine. When the machine is PoweredOff, an offline snapshot is created. When the machine is Running a live snapshot is created, and an online snapshot is is created when Paused.
The taken snapshot is always based on the current snapshot of the associated virtual machine and becomes a new current snapshot.

> **Note:** This method implicitly calls IMachine::saveSettings() to save all current machine settings before taking an offline snapshot.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Virtual machine currently changing state.

5.11.25 teleport

```
IProgress IConsole::teleport(
           [in] wstring hostname,
           [in] unsigned long tcpport,
           [in] wstring password,
           [in] unsigned long maxDowntime)
```

hostname The name or IP of the host to teleport to.

tcpport The TCP port to connect to (1..65535).

password The password.

maxDowntime The maximum allowed downtime given as milliseconds. 0 is not a valid value. Recommended value: 250 ms.

The higher the value is, the greater the chance for a successful teleportation. A small value may easily result in the teleportation process taking hours and eventually fail.

Note: The current implementation treats this a guideline, not as an absolute rule.

Teleport the VM to a different host machine or process.
TODO explain the details.
If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Virtual machine not running or paused.

5.12 IDHCPServer

The IDHCPServer interface represents the vbox dhcp server configuration.
To enumerate all the dhcp servers on the host, use the IVirtualBox::DHCPServers[] attribute.

5.12.1 Attributes

5.12.1.1 enabled (read/write)

```
boolean IDHCPServer::enabled
```

specifies if the dhcp server is enabled

5.12.1.2 IPAddress (read-only)

```
wstring IDHCPServer::IPAddress
```

specifies server IP

5.12.1.3 networkMask (read-only)

```
wstring IDHCPServer::networkMask
```

specifies server network mask

5.12.1.4 networkName (read-only)

```
wstring IDHCPServer::networkName
```

specifies internal network name the server is used for

5.12.1.5 lowerIP (read-only)

```
wstring IDHCPServer::lowerIP
```

specifies from IP address in server address range

5.12.1.6 upperIP (read-only)

```
wstring IDHCPServer::upperIP
```

specifies to IP address in server address range

5.12.2 setConfiguration

```
void IDHCPServer::setConfiguration(
            [in] wstring IPAddress,
            [in] wstring networkMask,
            [in] wstring FromIPAddress,
            [in] wstring ToIPAddress)
```

IPAddress server IP address

networkMask server network mask

FromIPAddress server From IP address for address range

ToIPAddress server To IP address for address range

configures the server
If this method fails, the following error codes may be reported:

- E_INVALIDARG: invalid configuration supplied

5.12.3 start

```
void IDHCPServer::start(
            [in] wstring networkName,
            [in] wstring trunkName,
            [in] wstring trunkType)
```

networkName Name of internal network DHCP server should attach to.

trunkName Name of internal network trunk.

trunkType Type of internal network trunk.

Starts DHCP server process.
If this method fails, the following error codes may be reported:

- E_FAIL: Failed to start the process.

5.12.4 stop

```
void IDHCPServer::stop()
```

Stops DHCP server process.
If this method fails, the following error codes may be reported:

- E_FAIL: Failed to stop the process.

5.13 IDisplay

The IDisplay interface represents the virtual machine's display.

The object implementing this interface is contained in each IConsole::display attribute and represents the visual output of the virtual machine.

The virtual display supports pluggable output targets represented by the IFramebuffer interface. Examples of the output target are a window on the host computer or an RDP session's display on a remote computer.

5.13.1 completeVHWACommand

> **Note:** This method is not supported in the web service.

```
void IDisplay::completeVHWACommand(
           [in] [ptr] octet command)
```

command Pointer to VBOXVHWACMD containing the completed command.

Signals that the Video HW Acceleration command has completed.

5.13.2 drawToScreen

> **Note:** This method is not supported in the web service.

```
void IDisplay::drawToScreen(
           [in] unsigned long screenId,
           [in] [ptr] octet address,
           [in] unsigned long x,
           [in] unsigned long y,
           [in] unsigned long width,
           [in] unsigned long height)
```

screenId Monitor to take the screenshot from.

address Address to store the screenshot to

x Relative to the screen top left corner.

y Relative to the screen top left corner.

width Desired image width.

height Desired image height.

Draws a 32-bpp image of the specified size from the given buffer to the given point on the VM display.

If this method fails, the following error codes may be reported:

- `E_NOTIMPL:` Feature not implemented.
- `VBOX_E_IPRT_ERROR:` Could not draw to screen.

5.13.3 getFramebuffer

Note: This method is not supported in the web service.

```
void IDisplay::getFramebuffer(
            [in] unsigned long screenId,
            [out] IFramebuffer framebuffer,
            [out] long xOrigin,
            [out] long yOrigin)
```

screenId

framebuffer

xOrigin

yOrigin

Queries the framebuffer for given screen.

5.13.4 getScreenResolution

```
void IDisplay::getScreenResolution(
            [in] unsigned long screenId,
            [out] unsigned long width,
            [out] unsigned long height,
            [out] unsigned long bitsPerPixel)
```

screenId

width

height

bitsPerPixel

Queries display width, height and color depth for given screen.

5.13.5 invalidateAndUpdate

```
void IDisplay::invalidateAndUpdate()
```

Does a full invalidation of the VM display and instructs the VM to update it.
If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Could not invalidate and update screen.

5.13.6 resizeCompleted

```
void IDisplay::resizeCompleted(
            [in] unsigned long screenId)
```

screenId

Signals that a framebuffer has completed the resize operation.
If this method fails, the following error codes may be reported:

- VBOX_E_NOT_SUPPORTED: Operation only valid for external frame buffers.

5.13.7 setFramebuffer

Note: This method is not supported in the web service.

```
void IDisplay::setFramebuffer(
           [in] unsigned long screenId,
           [in] IFramebuffer framebuffer)
```

screenId

framebuffer

Sets the framebuffer for given screen.

5.13.8 setSeamlessMode

```
void IDisplay::setSeamlessMode(
           [in] boolean enabled)
```

enabled

Enables or disables seamless guest display rendering (seamless desktop integration) mode.

Note: Calling this method has no effect if IGuest::supportsSeamless returns `false`.

5.13.9 setVideoModeHint

```
void IDisplay::setVideoModeHint(
           [in] unsigned long width,
           [in] unsigned long height,
           [in] unsigned long bitsPerPixel,
           [in] unsigned long display)
```

width

height

bitsPerPixel

display

Asks VirtualBox to request the given video mode from the guest. This is just a hint and it cannot be guaranteed that the requested resolution will be used. Guest Additions are required for the request to be seen by guests. The caller should issue the request and wait for a resolution change and after a timeout retry.

Specifying `0` for either `width`, `height` or `bitsPerPixel` parameters means that the corresponding values should be taken from the current video mode (i.e. left unchanged).

If the guest OS supports multi-monitor configuration then the `display` parameter specifies the number of the guest display to send the hint to: `0` is the primary display, `1` is the first secondary and so on. If the multi-monitor configuration is not supported, `display` must be `0`.

If this method fails, the following error codes may be reported:

- `E_INVALIDARG`: The `display` is not associated with any monitor.

5.13.10 takeScreenShot

> **Note:** This method is not supported in the web service.

```
void IDisplay::takeScreenShot(
            [in] unsigned long screenId,
            [in] [ptr] octet address,
            [in] unsigned long width,
            [in] unsigned long height)
```

screenId

address

width

height

Takes a screen shot of the requested size and copies it to the 32-bpp buffer allocated by the caller and pointed to by `address`. A pixel consists of 4 bytes in order: B, G, R, 0.

> **Note:** This API can be used only by the COM/XPCOM C++ API as it requires pointer support. Use takeScreenShotToArray() with other language bindings.

If this method fails, the following error codes may be reported:

- `E_NOTIMPL:` Feature not implemented.
- `VBOX_E_IPRT_ERROR:` Could not take a screenshot.

5.13.11 takeScreenShotPNGToArray

```
octet[] IDisplay::takeScreenShotPNGToArray(
            [in] unsigned long screenId,
            [in] unsigned long width,
            [in] unsigned long height)
```

screenId Monitor to take the screenshot from.

width Desired image width.

height Desired image height.

Takes a guest screen shot of the requested size and returns it as PNG image in array.
If this method fails, the following error codes may be reported:

- `E_NOTIMPL:` Feature not implemented.
- `VBOX_E_IPRT_ERROR:` Could not take a screenshot.

5.13.12 takeScreenShotToArray

```
octet[] IDisplay::takeScreenShotToArray(
            [in] unsigned long screenId,
            [in] unsigned long width,
            [in] unsigned long height)
```

screenId Monitor to take screenshot from.

width Desired image width.

height Desired image height.

Takes a guest screen shot of the requested size and returns it as an array of bytes in uncompressed 32-bit RGBA format. A pixel consists of 4 bytes in order: R, G, B, 0xFF.

This API is slow, but could be the only option to get guest screenshot for scriptable languages not allowed to manipulate with addresses directly.

If this method fails, the following error codes may be reported:

- `E_NOTIMPL:` Feature not implemented.
- `VBOX_E_IPRT_ERROR:` Could not take a screenshot.

5.14 IEvent

Abstract parent interface for VirtualBox events. Actual events will typically implement a more specific interface which derives from this (see below).

Introduction to VirtualBox events

Generally speaking, an event (represented by this interface) signals that something happened, while an event listener (see IEventListener) represents an entity that is interested in certain events. In order for this to work with unidirectional protocols (i.e. web services), the concepts of passive and active listener are used.

Event consumers can register themselves as listeners, providing an array of events they are interested in (see IEventSource::registerListener()). When an event triggers, the listener is notified about the event. The exact mechanism of the notification depends on whether the listener was registered as an active or passive listener:

- An active listener is very similar to a callback: it is a function invoked by the API. As opposed to the callbacks that were used in the API before VirtualBox 4.0 however, events are now objects with an interface hierarchy.
- Passive listeners are somewhat trickier to implement, but do not require a client function to be callable, which is not an option with scripting languages or web service clients. Internally the IEventSource implementation maintains an event queue for each passive listener, and newly arrived events are put in this queue. When the listener calls IEventSource::getEvent(), first element from its internal event queue is returned. When the client completes processing of an event, the IEventSource::eventProcessed() function must be called, acknowledging that the event was processed. It supports implementing waitable events. On passive listener unregistration, all events from its queue are auto-acknowledged.

Waitable events are useful in situations where the event generator wants to track delivery or a party wants to wait until all listeners have completed the event. A typical example would be a vetoable event (see IVetoEvent) where a listeners might veto a certain action, and thus the event producer has to make sure that all listeners have processed the event and not vetoed before taking the action.

A given event may have both passive and active listeners at the same time.

Using events

Any VirtualBox object capable of producing externally visible events provides an `eventSource` read-only attribute, which is of the type IEventSource. This event source object is notified by VirtualBox once something has happened, so consumers may register event listeners with this event source. To register a listener, an object implementing the IEventListener interface must be provided. For active listeners, such an object is typically created by the consumer, while for passive listeners IEventSource::createListener() should be used. Please note that a listener created with `createListener`() must not be used as an active listener.

Once created, the listener must be registered to listen for the desired events (see IEventSource::registerListener()), providing an array of VBoxEventType enums. Those elements can either be the individual event IDs or wildcards matching multiple event IDs.

After registration, the callback's IEventListener::handleEvent() method is called automatically when the event is triggered, while passive listeners have to call IEventSource::getEvent() and IEventSource::eventProcessed() in an event processing loop.

The IEvent interface is an abstract parent interface for all such VirtualBox events coming in. As a result, the standard use pattern inside IEventListener::handleEvent() or the event processing loop is to check the type attribute of the event and then cast to the appropriate specific interface using `QueryInterface`().

5.14.1 Attributes

5.14.1.1 type (read-only)

`VBoxEventType IEvent::type`

Event type.

5.14.1.2 source (read-only)

`IEventSource IEvent::source`

Source of this event.

5.14.1.3 waitable (read-only)

`boolean IEvent::waitable`

If we can wait for this event being processed. If false, waitProcessed() returns immediately, and setProcessed() doesn't make sense. Non-waitable events are generally better performing, as no additional overhead associated with waitability imposed. Waitable events are needed when one need to be able to wait for particular event processed, for example for vetoable changes, or if event refers to some resource which need to be kept immutable until all consumers confirmed events.

5.14.2 setProcessed

`void IEvent::setProcessed()`

Internal method called by the system when all listeners of a particular event have called IEventSource::eventProcessed(). This should not be called by client code.

5.14.3 waitProcessed

```
boolean IEvent::waitProcessed(
            [in] long timeout)
```

timeout Maximum time to wait for event processeing, in ms; 0 = no wait, -1 = indefinite wait.

Wait until time outs, or this event is processed. Event must be waitable for this operation to have described semantics, for non-waitable returns true immediately.

5.15 IEventContext

Placeholder class for event contexts.

5.16 IEventListener

Event listener. An event listener can work in either active or passive mode, depending on the way it was registered. See IEvent for an introduction to VirtualBox event handling.

5.16.1 handleEvent

```
void IEventListener::handleEvent(
            [in] IEvent event)
```

event Event available.

Handle event callback for active listeners. It is not called for passive listeners. After calling handleEvent() on all active listeners and having received acknowledgement from all passive listeners via IEventSource::eventProcessed(), the event is marked as processed and IEvent::waitProcessed() will return immediately.

5.17 IEventSource

Event source. Generally, any object which could generate events can be an event source, or aggregate one. To simplify using one-way protocols such as webservices running on top of HTTP(S), an event source can work with listeners in either active or passive mode. In active mode it is up to the IEventSource implementation to call IEventListener::handleEvent(), in passive mode the event source keeps track of pending events for each listener and returns available events on demand.

See IEvent for an introduction to VirtualBox event handling.

5.17.1 createAggregator

```
IEventSource IEventSource::createAggregator(
            [in] IEventSource subordinates[])
```

subordinates Subordinate event source this one aggregatres.

Creates an aggregator event source, collecting events from multiple sources. This way a single listener can listen for events coming from multiple sources, using a single blocking getEvent() on the returned aggregator.

5.17.2 createListener

```
IEventListener IEventSource::createListener()
```

Creates a new listener object, useful for passive mode.

5.17.3 eventProcessed

```
void IEventSource::eventProcessed(
            [in] IEventListener listener,
            [in] IEvent event)
```

listener Which listener processed event.

event Which event.

Must be called for waitable events after a particular listener finished its event processing. When all listeners of a particular event have called this method, the system will then call IEvent::setProcessed().

5.17.4 fireEvent

```
boolean IEventSource::fireEvent(
            [in] IEvent event,
            [in] long timeout)
```

event Event to deliver.

timeout Maximum time to wait for event processing (if event is waitable), in ms; 0 = no wait, -1 = indefinite wait.

Fire an event for this source.

5.17.5 getEvent

```
IEvent IEventSource::getEvent(
            [in] IEventListener listener,
            [in] long timeout)
```

listener Which listener to get data for.

timeout Maximum time to wait for events, in ms; 0 = no wait, -1 = indefinite wait.

Get events from this peer's event queue (for passive mode). Calling this method regularly is required for passive event listeners to avoid system overload; see registerListener() for details.

If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: Listener is not registered, or autounregistered.

5.17.6 registerListener

```
void IEventSource::registerListener(
            [in] IEventListener listener,
            [in] VBoxEventType interesting[],
            [in] boolean active)
```

listener Listener to register.

interesting Event types listener is interested in. One can use wildcards like - Any to specify wildcards, matching more than one event.

active Which mode this listener is operating in. In active mode, IEventListener::handleEvent() is called directly. In passive mode, an internal event queue is created for this this IEventListener. For each event coming in, it is added to queues for all interested registered passive listeners. It is then up to the external code to call the listener's IEventListener::handleEvent() method. When done with an event, the external code must call eventProcessed().

Register an event listener.

> **Note:** To avoid system overload, the VirtualBox server process checks if passive event listeners call getEvent() frequently enough. In the current implementation, if more than 500 pending events are detected for a passive event listener, it is forcefully unregistered by the system, and further getEvent() calls will return VBOX_E_OBJECT_NOT_FOUND.

5.17.7 unregisterListener

```
void IEventSource::unregisterListener(
          [in] IEventListener listener)
```

listener Listener to unregister.

Unregister an event listener. If listener is passive, and some waitable events are still in queue they are marked as processed automatically.

5.18 IEventSourceChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when an event source state changes (listener added or removed).

5.18.1 Attributes

5.18.1.1 listener (read-only)

```
IEventListener IEventSourceChangedEvent::listener
```

Event listener which has changed.

5.18.1.2 add (read-only)

```
boolean IEventSourceChangedEvent::add
```

Flag whether listener was added or removed.

5.19 IExtPack (IExtPackBase)

Note: This interface is not supported in the web service.

Note: This interface extends IExtPackBase and therefore supports all its methods and attributes as well.

Interface for querying information about an extension pack as well as accessing COM objects within it.

5.19.1 queryObject

```
$unknown IExtPack::queryObject(
           [in] wstring objUuid)
```

objUuid The object ID. What exactly this is

Queries the IUnknown interface to an object in the extension pack main module. This allows plug-ins and others to talk directly to an extension pack.

5.20 IExtPackBase

Note: This interface is not supported in the web service.

Interface for querying information about an extension pack as well as accessing COM objects within it.

5.20.1 Attributes

5.20.1.1 name (read-only)

```
wstring IExtPackBase::name
```

The extension pack name. This is unique.

5.20.1.2 description (read-only)

```
wstring IExtPackBase::description
```

The extension pack description.

5.20.1.3 version (read-only)

```
wstring IExtPackBase::version
```

The extension pack version string. This is on the same form as other VirtualBox version strings, i.e.: “1.2.3”, “1.2.3_BETA1”, “1.2.3-OSE”, “1.2.3r45678”, “1.2.3r45678-OSE”, “1.2.3_BETA1-r45678” or “1.2.3_BETA1-r45678-OSE”

5.20.1.4 revision (read-only)

```
unsigned long IExtPackBase::revision
```

The extension pack internal revision number.

5.20.1.5 VRDEModule (read-only)

```
wstring IExtPackBase::VRDEModule
```

The name of the VRDE module if the extension pack sports one.

5.20.1.6 plugIns (read-only)

```
IExtPackPlugIn IExtPackBase::plugIns[]
```

Note: This attribute is not supported in the web service.

Plug-ins provided by this extension pack.

5.20.1.7 usable (read-only)

```
boolean IExtPackBase::usable
```

Indicates whether the extension pack is usable or not.
There are a number of reasons why an extension pack might be unusable, typical examples would be broken installation/file or that it is incompatible with the current VirtualBox version.

5.20.1.8 whyUnusable (read-only)

```
wstring IExtPackBase::whyUnusable
```

String indicating why the extension pack is not usable. This is an empty string if usable and always a non-empty string if not usable.

5.20.1.9 showLicense (read-only)

```
boolean IExtPackBase::showLicense
```

Whether to show the license before installation

5.20.1.10 license (read-only)

```
wstring IExtPackBase::license
```

The default HTML license text for the extension pack. Same as calling queryLicense with preferredLocale and preferredLanguage as empty strings and format set to html.

5.20.2 queryLicense

```
wstring IExtPackBase::queryLicense(
            [in] wstring preferredLocale,
            [in] wstring preferredLanguage,
            [in] wstring format)
```

preferredLocale The preferred license locale. Pass an empty string to get the default license.

preferredLanguage The preferred license language. Pass an empty string to get the default language for the locale.

format The license format: html, rtf or txt. If a license is present there will always be an HTML of it, the rich text format (RTF) and plain text (txt) versions are optional. If

Full feature version of the license attribute.

5.21 IExtPackFile (IExtPackBase)

Note: This interface is not supported in the web service.

Note: This interface extends IExtPackBase and therefore supports all its methods and attributes as well.

Extension pack file (aka tarball, .vbox-extpack) representation returned by IExtPackManager::openExtPackFile. This provides the base extension pack information with the addition of the file name. It also provides an alternative to IExtPackManager::install.

5.21.1 Attributes

5.21.1.1 filePath (read-only)

```
wstring IExtPackFile::filePath
```

The path to the extension pack file.

5.21.2 install

```
IProgress IExtPackFile::install(
            [in] boolean replace,
            [in] wstring displayInfo)
```

replace Set this to automatically uninstall any existing extension pack with the same name as the one being installed.

displayInfo Platform specific display information. Reserved for future hacks.

Install the extension pack.

5.22 IExtPackManager

Note: This interface is not supported in the web service.

Interface for managing VirtualBox Extension Packs.
TODO: Describe extension packs, how they are managed and how to create one.

5.22.1 Attributes

5.22.1.1 installedExtPacks (read-only)

```
IExtPack IExtPackManager::installedExtPacks[]
```

> **Note:** This attribute is not supported in the web service.

List of the installed extension packs.

5.22.2 IsExtPackUsable

```
boolean IExtPackManager::IsExtPackUsable(
            [in] wstring name)
```

name The name of the extension pack to check for.

Check if the given extension pack is loaded and usable.

5.22.3 QueryAllPlugInsForFrontend

```
wstring[] IExtPackManager::QueryAllPlugInsForFrontend(
            [in] wstring frontendName)
```

frontendName The name of the frontend or component.

Gets the path to all the plug-in modules for a given frontend.
This is a convenience method that is intended to simplify the plug-in loading process for a frontend.

5.22.4 cleanup

```
void IExtPackManager::cleanup()
```

Cleans up failed installs and uninstalls

5.22.5 find

> **Note:** This method is not supported in the web service.

```
IExtPack IExtPackManager::find(
            [in] wstring name)
```

name The name of the extension pack to locate.

Returns the extension pack with the specified name if found.
If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: No extension pack matching `name` was found.

5.22.6 openExtPackFile

> **Note:** This method is not supported in the web service.

```
IExtPackFile IExtPackManager::openExtPackFile(
           [in] wstring path)
```

path The path of the extension pack tarball.

Attempts to open an extension pack file in preparation for installation.

5.22.7 uninstall

```
IProgress IExtPackManager::uninstall(
           [in] wstring name,
           [in] boolean forcedRemoval,
           [in] wstring displayInfo)
```

name The name of the extension pack to uninstall.

forcedRemoval Forced removal of the extension pack. This means that the uninstall hook will not be called.

displayInfo Platform specific display information. Reserved for future hacks.

Uninstalls an extension pack, removing all related files.

5.23 IExtPackPlugIn

> **Note:** This interface is not supported in the web service.

Interface for keeping information about a plug-in that ships with an extension pack.

5.23.1 Attributes

5.23.1.1 name (read-only)

```
wstring IExtPackPlugIn::name
```

The plug-in name.

5.23.1.2 description (read-only)

```
wstring IExtPackPlugIn::description
```

The plug-in description.

5.23.1.3 frontend (read-only)

```
wstring IExtPackPlugIn::frontend
```

The name of the frontend or component name this plug-in plugs into.

5.23.1.4 modulePath (read-only)

`wstring IExtPackPlugIn::modulePath`

The module path.

5.24 IExtraDataCanChangeEvent (IVetoEvent)

Note: This interface extends IVetoEvent and therefore supports all its methods and attributes as well.

Notification when someone tries to change extra data for either the given machine or (if `null`) global extra data. This gives the chance to veto against changes.

5.24.1 Attributes

5.24.1.1 machineId (read-only)

`uuid IExtraDataCanChangeEvent::machineId`

ID of the machine this event relates to. Null for global extra data changes.

5.24.1.2 key (read-only)

`wstring IExtraDataCanChangeEvent::key`

Extra data key that has changed.

5.24.1.3 value (read-only)

`wstring IExtraDataCanChangeEvent::value`

Extra data value for the given key.

5.25 IExtraDataChangedEvent (IEvent)

Note: This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when machine specific or global extra data has changed.

5.25.1 Attributes

5.25.1.1 machineId (read-only)

`uuid IExtraDataChangedEvent::machineId`

ID of the machine this event relates to. Null for global extra data changes.

5.25.1.2 key (read-only)

`wstring IExtraDataChangedEvent::key`

Extra data key that has changed.

5.25.1.3 value (read-only)

```
wstring IExtraDataChangedEvent::value
```

Extra data value for the given key.

5.26 IFramebuffer

> **Note:** This interface is not supported in the web service.

5.26.1 Attributes

5.26.1.1 address (read-only)

```
octet IFramebuffer::address
```

Address of the start byte of the frame buffer.

5.26.1.2 width (read-only)

```
unsigned long IFramebuffer::width
```

Frame buffer width, in pixels.

5.26.1.3 height (read-only)

```
unsigned long IFramebuffer::height
```

Frame buffer height, in pixels.

5.26.1.4 bitsPerPixel (read-only)

```
unsigned long IFramebuffer::bitsPerPixel
```

Color depth, in bits per pixel. When pixelFormat is FOURCC_RGB, valid values are: 8, 15, 16, 24 and 32.

5.26.1.5 bytesPerLine (read-only)

```
unsigned long IFramebuffer::bytesPerLine
```

Scan line size, in bytes. When pixelFormat is FOURCC_RGB, the size of the scan line must be aligned to 32 bits.

5.26.1.6 pixelFormat (read-only)

```
unsigned long IFramebuffer::pixelFormat
```

Frame buffer pixel format. It's either one of the values defined by FramebufferPixelFormat or a raw FOURCC code.

> **Note:** This attribute must never return Opaque – the format of the buffer address points to must be always known.

5.26.1.7 usesGuestVRAM (read-only)

```
boolean IFramebuffer::usesGuestVRAM
```

Defines whether this frame buffer uses the virtual video card's memory buffer (guest VRAM) directly or not. See requestResize() for more information.

5.26.1.8 heightReduction (read-only)

```
unsigned long IFramebuffer::heightReduction
```

Hint from the frame buffer about how much of the standard screen height it wants to use for itself. This information is exposed to the guest through the VESA BIOS and VMMDev interface so that it can use it for determining its video mode table. It is not guaranteed that the guest respects the value.

5.26.1.9 overlay (read-only)

```
IFramebufferOverlay IFramebuffer::overlay
```

> **Note:** This attribute is not supported in the web service.

An alpha-blended overlay which is superposed over the frame buffer. The initial purpose is to allow the display of icons providing information about the VM state, including disk activity, in front ends which do not have other means of doing that. The overlay is designed to controlled exclusively by IDisplay. It has no locking of its own, and any changes made to it are not guaranteed to be visible until the affected portion of IFramebuffer is updated. The overlay can be created lazily the first time it is requested. This attribute can also return `null` to signal that the overlay is not implemented.

5.26.1.10 winId (read-only)

```
long long IFramebuffer::winId
```

Platform-dependent identifier of the window where context of this frame buffer is drawn, or zero if there's no such window.

5.26.2 getVisibleRegion

> **Note:** This method is not supported in the web service.

```
unsigned long IFramebuffer::getVisibleRegion(
            [in] [ptr] octet rectangles,
            [in] unsigned long count)
```

rectangles Pointer to the `RTRECT` array to receive region data.

count Number of `RTRECT` elements in the `rectangles` array.

Returns the visible region of this frame buffer.

If the `rectangles` parameter is `null` then the value of the `count` parameter is ignored and the number of elements necessary to describe the current visible region is returned in `countCopied`.

If `rectangles` is not `null` but `count` is less than the required number of elements to store region data, the method will report a failure. If `count` is equal or greater than the required number of elements, then the actual number of elements copied to the provided array will be returned in `countCopied`.

> **Note:** The address of the provided array must be in the process space of this IFramebuffer object.

> **Note:** Method not yet implemented.

5.26.3 lock

```
void IFramebuffer::lock()
```

Locks the frame buffer. Gets called by the IDisplay object where this frame buffer is bound to.

5.26.4 notifyUpdate

```
void IFramebuffer::notifyUpdate(
            [in] unsigned long x,
            [in] unsigned long y,
            [in] unsigned long width,
            [in] unsigned long height)
```

x

y

width

height

Informs about an update. Gets called by the display object where this buffer is registered.

5.26.5 processVHWACommand

> **Note:** This method is not supported in the web service.

```
void IFramebuffer::processVHWACommand(
            [in] [ptr] octet command)
```

command Pointer to VBOXVHWACMD containing the command to execute.

Posts a Video HW Acceleration Command to the frame buffer for processing. The commands used for 2D video acceleration (DDraw surface creation/destroying, blitting, scaling, color conversion, overlaying, etc.) are posted from quest to the host to be processed by the host hardware.

> **Note:** The address of the provided command must be in the process space of this IFramebuffer object.

5.26.6 requestResize

Note: This method is not supported in the web service.

```
boolean IFramebuffer::requestResize(
            [in] unsigned long screenId,
            [in] unsigned long pixelFormat,
            [in] [ptr] octet VRAM,
            [in] unsigned long bitsPerPixel,
            [in] unsigned long bytesPerLine,
            [in] unsigned long width,
            [in] unsigned long height)
```

screenId Logical screen number. Must be used in the corresponding call to IDisplay::resizeCompleted() if this call is made.

pixelFormat Pixel format of the memory buffer pointed to by `VRAM`. See also FramebufferPixelFormat.

VRAM Pointer to the virtual video card's VRAM (may be `null`).

bitsPerPixel Color depth, bits per pixel.

bytesPerLine Size of one scan line, in bytes.

width Width of the guest display, in pixels.

height Height of the guest display, in pixels.

Requests a size and pixel format change.

There are two modes of working with the video buffer of the virtual machine. The *indirect* mode implies that the IFramebuffer implementation allocates a memory buffer for the requested display mode and provides it to the virtual machine. In *direct* mode, the IFramebuffer implementation uses the memory buffer allocated and owned by the virtual machine. This buffer represents the video memory of the emulated video adapter (so called *guest VRAM*). The direct mode is usually faster because the implementation gets a raw pointer to the guest VRAM buffer which it can directly use for visualizing the contents of the virtual display, as opposed to the indirect mode where the contents of guest VRAM are copied to the memory buffer provided by the implementation every time a display update occurs.

It is important to note that the direct mode is really fast only when the implementation uses the given guest VRAM buffer directly, for example, by blitting it to the window representing the virtual machine's display, which saves at least one copy operation comparing to the indirect mode. However, using the guest VRAM buffer directly is not always possible: the format and the color depth of this buffer may be not supported by the target window, or it may be unknown (opaque) as in case of text or non-linear multi-plane VGA video modes. In this case, the indirect mode (that is always available) should be used as a fallback: when the guest VRAM contents are copied to the implementation-provided memory buffer, color and format conversion is done automatically by the underlying code.

The `pixelFormat` parameter defines whether the direct mode is available or not. If `pixelFormat` is Opaque then direct access to the guest VRAM buffer is not available – the `VRAM`, `bitsPerPixel` and `bytesPerLine` parameters must be ignored and the implementation must use the indirect mode (where it provides its own buffer in one of the supported formats). In all other cases, `pixelFormat` together with `bitsPerPixel` and `bytesPerLine` define the format of the video memory buffer pointed to by the `VRAM` parameter and the implementation is free to choose which mode to use. To indicate that this frame buffer uses the direct mode, the implementation of the usesGuestVRAM attribute must return `true` and address must return exactly

the same address that is passed in the VRAM parameter of this method; otherwise it is assumed that the indirect strategy is chosen.

The width and height parameters represent the size of the requested display mode in both modes. In case of indirect mode, the provided memory buffer should be big enough to store data of the given display mode. In case of direct mode, it is guaranteed that the given VRAM buffer contains enough space to represent the display mode of the given size. Note that this frame buffer's width and height attributes must return exactly the same values as passed to this method after the resize is completed (see below).

The finished output parameter determines if the implementation has finished resizing the frame buffer or not. If, for some reason, the resize cannot be finished immediately during this call, finished must be set to false, and the implementation must call IDisplay::resizeCompleted() after it has returned from this method as soon as possible. If finished is false, the machine will not call any frame buffer methods until IDisplay::resizeCompleted() is called.

Note that if the direct mode is chosen, the bitsPerPixel, bytesPerLine and pixelFormat attributes of this frame buffer must return exactly the same values as specified in the parameters of this method, after the resize is completed. If the indirect mode is chosen, these attributes must return values describing the format of the implementation's own memory buffer address points to. Note also that the bitsPerPixel value must always correlate with pixelFormat. Note that the pixelFormat attribute must never return Opaque regardless of the selected mode.

> **Note:** This method is called by the IDisplay object under the lock() provided by this IFramebuffer implementation. If this method returns false in finished, then this lock is not released until IDisplay::resizeCompleted() is called.

5.26.7 setVisibleRegion

> **Note:** This method is not supported in the web service.

```
void IFramebuffer::setVisibleRegion(
           [in] [ptr] octet rectangles,
           [in] unsigned long count)
```

rectangles Pointer to the RTRECT array.

count Number of RTRECT elements in the rectangles array.

Suggests a new visible region to this frame buffer. This region represents the area of the VM display which is a union of regions of all top-level windows of the guest operating system running inside the VM (if the Guest Additions for this system support this functionality). This information may be used by the frontends to implement the seamless desktop integration feature.

> **Note:** The address of the provided array must be in the process space of this IFramebuffer object.

> **Note:** The IFramebuffer implementation must make a copy of the provided array of rectangles.

> **Note:** Method not yet implemented.

5.26.8 unlock

```
void IFramebuffer::unlock()
```

Unlocks the frame buffer. Gets called by the IDisplay object where this frame buffer is bound to.

5.26.9 videoModeSupported

```
boolean IFramebuffer::videoModeSupported(
            [in] unsigned long width,
            [in] unsigned long height,
            [in] unsigned long bpp)
```

width

height

bpp

Returns whether the frame buffer implementation is willing to support a given video mode. In case it is not able to render the video mode (or for some reason not willing), it should return `false`. Usually this method is called when the guest asks the VMM device whether a given video mode is supported so the information returned is directly exposed to the guest. It is important that this method returns very quickly.

5.27 IFramebufferOverlay (IFramebuffer)

Note: This interface is not supported in the web service.

Note: This interface extends IFramebuffer and therefore supports all its methods and attributes as well.

The IFramebufferOverlay interface represents an alpha blended overlay for displaying status icons above an IFramebuffer. It is always created not visible, so that it must be explicitly shown. It only covers a portion of the IFramebuffer, determined by its width, height and co-ordinates. It is always in packed pixel little-endian 32bit ARGB (in that order) format, and may be written to directly. Do re-read the width though, after setting it, as it may be adjusted (increased) to make it more suitable for the front end.

5.27.1 Attributes

5.27.1.1 x (read-only)

```
unsigned long IFramebufferOverlay::x
```

X position of the overlay, relative to the frame buffer.

5.27.1.2 y (read-only)

```
unsigned long IFramebufferOverlay::y
```

Y position of the overlay, relative to the frame buffer.

5.27.1.3 visible (read/write)

```
boolean IFramebufferOverlay::visible
```

Whether the overlay is currently visible.

5.27.1.4 alpha (read/write)

```
unsigned long IFramebufferOverlay::alpha
```

The global alpha value for the overlay. This may or may not be supported by a given front end.

5.27.2 move

```
void IFramebufferOverlay::move(
           [in] unsigned long x,
           [in] unsigned long y)
```

x

y

Changes the overlay's position relative to the IFramebuffer.

5.28 IGuest

The IGuest interface represents information about the operating system running inside the virtual machine. Used in IConsole::guest.

IGuest provides information about the guest operating system, whether Guest Additions are installed and other OS-specific virtual machine properties.

5.28.1 Attributes

5.28.1.1 OSTypeId (read-only)

```
wstring IGuest::OSTypeId
```

Identifier of the Guest OS type as reported by the Guest Additions. You may use IVirtualBox::getGuestOSType() to obtain an IGuestOSType object representing details about the given Guest OS type.

> **Note:** If Guest Additions are not installed, this value will be the same as IMachine::OSTypeId.

5.28.1.2 additionsRunLevel (read-only)

```
AdditionsRunLevelType IGuest::additionsRunLevel
```

Current run level of the Guest Additions.

5.28.1.3 additionsVersion (read-only)

```
wstring IGuest::additionsVersion
```

Version of the Guest Additions including the revision (3 decimal numbers separated by dots + revision number) installed on the guest or empty when the Additions are not installed.

5.28.1.4 supportsSeamless (read-only)

```
boolean IGuest::supportsSeamless
```

Flag whether seamless guest display rendering (seamless desktop integration) is supported.

5.28.1.5 supportsGraphics (read-only)

```
boolean IGuest::supportsGraphics
```

Flag whether the guest is in graphics mode. If it is not, then seamless rendering will not work, resize hints are not immediately acted on and guest display resizes are probably not initiated by the guest additions.

5.28.1.6 memoryBalloonSize (read/write)

```
unsigned long IGuest::memoryBalloonSize
```

Guest system memory balloon size in megabytes (transient property).

5.28.1.7 statisticsUpdateInterval (read/write)

```
unsigned long IGuest::statisticsUpdateInterval
```

Interval to update guest statistics in seconds.

5.28.2 copyToGuest

```
IProgress IGuest::copyToGuest(
            [in] wstring source,
            [in] wstring dest,
            [in] wstring userName,
            [in] wstring password,
            [in] unsigned long flags)
```

source Source file on the host to copy.

dest Destination path on the guest.

userName User name under which the copy command will be executed; the user has to exist and have the appropriate rights to write to the destination path.

password Password of the user account specified.

flags CopyFileFlag flags. Not used at the moment and should be set to 0.

Copies files/directories from host to the guest.
If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Error while copying.

5.28.3 createDirectory

```
IProgress IGuest::createDirectory(
            [in] wstring directory,
            [in] wstring userName,
            [in] wstring password,
            [in] unsigned long mode,
            [in] unsigned long flags)
```

directory Directory to create.

userName User name under which the directory creation will be executed; the user has to exist and have the appropriate rights to create the desired directory.

password Password of the user account specified.

mode File mode.

flags CreateDirectoryFlag flags.

Creates a directory on the guest.
If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Error while creating directory.

5.28.4 executeProcess

```
IProgress IGuest::executeProcess(
            [in] wstring execName,
            [in] unsigned long flags,
            [in] wstring arguments[],
            [in] wstring environment[],
            [in] wstring userName,
            [in] wstring password,
            [in] unsigned long timeoutMS,
            [out] unsigned long pid)
```

execName Full path name of the command to execute on the guest; the commands has to exists in the guest VM in order to be executed.

flags ExecuteProcessFlag flags.

arguments Array of arguments passed to the execution command.

environment Environment variables that can be set while the command is being executed, in form of "NAME=VALUE"; one pair per entry. To unset a variable just set its name ("NAME") without a value.

userName User name under which the command will be executed; has to exist and have the appropriate rights to execute programs in the VM.

password Password of the user account specified.

timeoutMS The maximum timeout value (in msec) to wait for finished program execution. Pass 0 for an infinite timeout.

pid The PID (process ID) of the started command for later reference.

Executes an existing program inside the guest VM.
If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Could not execute process.

5.28.5 getAdditionsStatus

```
boolean IGuest::getAdditionsStatus(
            [in] AdditionsRunLevelType level)
```

level Status level to check

Retrieve the current status of a certain Guest Additions run level.
If this method fails, the following error codes may be reported:

- VBOX_E_NOT_SUPPORTED: Wrong status level specified.

5.28.6 getProcessOutput

```
octet[] IGuest::getProcessOutput(
            [in] unsigned long pid,
            [in] unsigned long flags,
            [in] unsigned long timeoutMS,
            [in] long long size)
```

pid Process id returned by earlier executeProcess() call.

flags Flags describing which output to retrieve.

timeoutMS The maximum timeout value (in msec) to wait for output data. Pass 0 for an infinite timeout.

size Size in bytes to read in the buffer.

Retrieves output of a formerly started process.
If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Could not retrieve output.

5.28.7 getProcessStatus

```
unsigned long IGuest::getProcessStatus(
            [in] unsigned long pid,
            [out] unsigned long exitcode,
            [out] unsigned long flags)
```

pid Process id returned by earlier executeProcess() call.

exitcode The exit code (if available).

flags Additional flags of process status. Not used at the moment and must be set to 0.

Retrieves status, exit code and the exit reason of a formerly started process.
If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Process with specified PID was not found.

5.28.8 internalGetStatistics

```
void IGuest::internalGetStatistics(
            [out] unsigned long cpuUser,
            [out] unsigned long cpuKernel,
            [out] unsigned long cpuIdle,
            [out] unsigned long memTotal,
            [out] unsigned long memFree,
            [out] unsigned long memBalloon,
            [out] unsigned long memShared,
            [out] unsigned long memCache,
            [out] unsigned long pagedTotal,
            [out] unsigned long memAllocTotal,
            [out] unsigned long memFreeTotal,
            [out] unsigned long memBalloonTotal,
            [out] unsigned long memSharedTotal)
```

cpuUser Percentage of processor time spent in user mode as seen by the guest

cpuKernel Percentage of processor time spent in kernel mode as seen by the guest

cpuIdle Percentage of processor time spent idling as seen by the guest

memTotal Total amount of physical guest RAM

memFree Free amount of physical guest RAM

memBalloon Amount of ballooned physical guest RAM

memShared Amount of shared physical guest RAM

memCache Total amount of guest (disk) cache memory

pagedTotal Total amount of space in the page file

memAllocTotal Total amount of memory allocated by the hypervisor

memFreeTotal Total amount of free memory available in the hypervisor

memBalloonTotal Total amount of memory ballooned by the hypervisor

memSharedTotal Total amount of shared memory in the hypervisor

Internal method; do not use as it might change at any time

5.28.9 setCredentials

```
void IGuest::setCredentials(
            [in] wstring userName,
            [in] wstring password,
            [in] wstring domain,
            [in] boolean allowInteractiveLogon)
```

userName User name string, can be empty

password Password string, can be empty

domain Domain name (guest logon scheme specific), can be empty

allowInteractiveLogon Flag whether the guest should alternatively allow the user to interactively specify different credentials. This flag might not be supported by all versions of the Additions.

Store login credentials that can be queried by guest operating systems with Additions installed. The credentials are transient to the session and the guest may also choose to erase them. Note that the caller cannot determine whether the guest operating system has queried or made use of the credentials.

If this method fails, the following error codes may be reported:

- VBOX_E_VM_ERROR: VMM device is not available.

5.28.10 setProcessInput

```
unsigned long IGuest::setProcessInput(
            [in] unsigned long pid,
            [in] unsigned long flags,
            [in] unsigned long timeoutMS,
            [in] octet data[])
```

pid Process id returned by earlier executeProcess() call.

flags ProcessInputFlag flags.

timeoutMS The maximum timeout value (in msec) to wait for getting the data transfered to the guest. Pass 0 for an infinite timeout.

data Buffer of input data to send to the started process to.

Sends input into a formerly started process.

If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Could not send input.

5.28.11 updateGuestAdditions

```
IProgress IGuest::updateGuestAdditions(
            [in] wstring source,
            [in] unsigned long flags)
```

source Path to the Guest Additions .ISO file to use for the upate.

flags AdditionsUpdateFlag flags.

Updates already installed Guest Additions in a VM (Windows guests only).

If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Error while updating.

5.29 IGuestKeyboardEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when guest keyboard event happens.

5.29.1 Attributes

5.29.1.1 scancodes (read-only)

```
long IGuestKeyboardEvent::scancodes[]
```

Array of scancodes.

5.30 IGuestMonitorChangedEvent (IEvent)

Note: This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when the guest enables one of its monitors.

5.30.1 Attributes

5.30.1.1 changeType (read-only)

`GuestMonitorChangedEventType IGuestMonitorChangedEvent::changeType`

What was changed for this guest monitor.

5.30.1.2 screenId (read-only)

`unsigned long IGuestMonitorChangedEvent::screenId`

The monitor which was changed.

5.30.1.3 originX (read-only)

`unsigned long IGuestMonitorChangedEvent::originX`

Physical X origin relative to the primary screen. Valid for Enabled and NewOrigin.

5.30.1.4 originY (read-only)

`unsigned long IGuestMonitorChangedEvent::originY`

Physical Y origin relative to the primary screen. Valid for Enabled and NewOrigin.

5.30.1.5 width (read-only)

`unsigned long IGuestMonitorChangedEvent::width`

Width of the screen. Valid for Enabled.

5.30.1.6 height (read-only)

`unsigned long IGuestMonitorChangedEvent::height`

Height of the screen. Valid for Enabled.

5.31 IGuestMouseEvent (IReusableEvent)

Note: This interface extends IReusableEvent and therefore supports all its methods and attributes as well.

Notification when guest mouse event happens.

5.31.1 Attributes

5.31.1.1 absolute (read-only)

```
boolean IGuestMouseEvent::absolute
```

If this event is relative or absolute.

5.31.1.2 x (read-only)

```
long IGuestMouseEvent::x
```

New X position, or X delta.

5.31.1.3 y (read-only)

```
long IGuestMouseEvent::y
```

New Y position, or Y delta.

5.31.1.4 z (read-only)

```
long IGuestMouseEvent::z
```

Z delta.

5.31.1.5 w (read-only)

```
long IGuestMouseEvent::w
```

W delta.

5.31.1.6 buttons (read-only)

```
long IGuestMouseEvent::buttons
```

Button state bitmask.

5.32 IGuestOSType

Note: With the web service, this interface is mapped to a structure. Attributes that return this interface will not return an object, but a complete structure containing the attributes listed below as structure members.

5.32.1 Attributes

5.32.1.1 familyId (read-only)

```
wstring IGuestOSType::familyId
```

Guest OS family identifier string.

5.32.1.2 familyDescription (read-only)

`wstring IGuestOSType::familyDescription`

Human readable description of the guest OS family.

5.32.1.3 id (read-only)

`wstring IGuestOSType::id`

Guest OS identifier string.

5.32.1.4 description (read-only)

`wstring IGuestOSType::description`

Human readable description of the guest OS.

5.32.1.5 is64Bit (read-only)

`boolean IGuestOSType::is64Bit`

Returns `true` if the given OS is 64-bit

5.32.1.6 recommendedIOAPIC (read-only)

`boolean IGuestOSType::recommendedIOAPIC`

Returns `true` if IO APIC recommended for this OS type.

5.32.1.7 recommendedVirtEx (read-only)

`boolean IGuestOSType::recommendedVirtEx`

Returns `true` if VT-x or AMD-V recommended for this OS type.

5.32.1.8 recommendedRAM (read-only)

`unsigned long IGuestOSType::recommendedRAM`

Recommended RAM size in Megabytes.

5.32.1.9 recommendedVRAM (read-only)

`unsigned long IGuestOSType::recommendedVRAM`

Recommended video RAM size in Megabytes.

5.32.1.10 recommendedHDD (read-only)

`long long IGuestOSType::recommendedHDD`

Recommended hard disk size in bytes.

5.32.1.11 adapterType (read-only)

`NetworkAdapterType IGuestOSType::adapterType`

Returns recommended network adapter for this OS type.

5.32.1.12 recommendedPae (read-only)

`boolean IGuestOSType::recommendedPae`

Returns `true` if using PAE is recommended for this OS type.

5.32.1.13 recommendedDvdStorageController (read-only)

`StorageControllerType IGuestOSType::recommendedDvdStorageController`

Recommended storage controller type for DVD/CD drives.

5.32.1.14 recommendedDvdStorageBus (read-only)

`StorageBus IGuestOSType::recommendedDvdStorageBus`

Recommended storage bus type for DVD/CD drives.

5.32.1.15 recommendedHdStorageController (read-only)

`StorageControllerType IGuestOSType::recommendedHdStorageController`

Recommended storage controller type for HD drives.

5.32.1.16 recommendedHdStorageBus (read-only)

`StorageBus IGuestOSType::recommendedHdStorageBus`

Recommended storage bus type for HD drives.

5.32.1.17 recommendedFirmware (read-only)

`FirmwareType IGuestOSType::recommendedFirmware`

Recommended firmware type.

5.32.1.18 recommendedUsbHid (read-only)

`boolean IGuestOSType::recommendedUsbHid`

Returns `true` if using USB Human Interface Devices, such as keyboard and mouse recommended.

5.32.1.19 recommendedHpet (read-only)

`boolean IGuestOSType::recommendedHpet`

Returns `true` if using HPET is recommended for this OS type.

5.32.1.20 recommendedUsbTablet (read-only)

`boolean IGuestOSType::recommendedUsbTablet`

Returns `true` if using a USB Tablet is recommended.

5.32.1.21 recommendedRtcUseUtc (read-only)

`boolean IGuestOSType::recommendedRtcUseUtc`

Returns `true` if the RTC of this VM should be set to UTC

5.32.1.22 recommendedChipset (read-only)

`ChipsetType IGuestOSType::recommendedChipset`

Recommended chipset type.

5.32.1.23 recommendedAudioController (read-only)

`AudioControllerType IGuestOSType::recommendedAudioController`

Recommended audio type.

5.33 IGuestPropertyChangedEvent (IMachineEvent)

> **Note:** This interface extends IMachineEvent and therefore supports all its methods and attributes as well.

Notification when a guest property has changed.

5.33.1 Attributes

5.33.1.1 name (read-only)

`wstring IGuestPropertyChangedEvent::name`

The name of the property that has changed.

5.33.1.2 value (read-only)

`wstring IGuestPropertyChangedEvent::value`

The new property value.

5.33.1.3 flags (read-only)

`wstring IGuestPropertyChangedEvent::flags`

The new property flags.

5.34 IHost

The IHost interface represents the physical machine that this VirtualBox installation runs on.

An object implementing this interface is returned by the IVirtualBox::host attribute. This interface contains read-only information about the host's physical hardware (such as what processors and disks are available, what the host operating system is, and so on) and also allows for manipulating some of the host's hardware, such as global USB device filters and host interface networking.

5.34.1 Attributes

5.34.1.1 DVDDrives (read-only)

`IMedium IHost::DVDDrives[]`

List of DVD drives available on the host.

5.34.1.2 floppyDrives (read-only)

`IMedium IHost::floppyDrives[]`

List of floppy drives available on the host.

5.34.1.3 USBDevices (read-only)

`IHostUSBDevice IHost::USBDevices[]`

List of USB devices currently attached to the host. Once a new device is physically attached to the host computer, it appears in this list and remains there until detached.

> **Note:** If USB functionality is not available in the given edition of VirtualBox, this method will set the result code to `E_NOTIMPL`.

5.34.1.4 USBDeviceFilters (read-only)

`IHostUSBDeviceFilter IHost::USBDeviceFilters[]`

List of USB device filters in action. When a new device is physically attached to the host computer, filters from this list are applied to it (in order they are stored in the list). The first matched filter will determine the action performed on the device.

Unless the device is ignored by these filters, filters of all currently running virtual machines (IUSBController::deviceFilters[]) are applied to it.

> **Note:** If USB functionality is not available in the given edition of VirtualBox, this method will set the result code to `E_NOTIMPL`.

See also: IHostUSBDeviceFilter, USBDeviceState

5.34.1.5 networkInterfaces (read-only)

`IHostNetworkInterface IHost::networkInterfaces[]`

List of host network interfaces currently defined on the host.

5.34.1.6 processorCount (read-only)

`unsigned long IHost::processorCount`

Number of (logical) CPUs installed in the host system.

5.34.1.7 processorOnlineCount (read-only)

`unsigned long IHost::processorOnlineCount`

Number of (logical) CPUs online in the host system.

5.34.1.8 processorCoreCount (read-only)

```
unsigned long IHost::processorCoreCount
```

Number of physical processor cores installed in the host system.

5.34.1.9 memorySize (read-only)

```
unsigned long IHost::memorySize
```

Amount of system memory in megabytes installed in the host system.

5.34.1.10 memoryAvailable (read-only)

```
unsigned long IHost::memoryAvailable
```

Available system memory in the host system.

5.34.1.11 operatingSystem (read-only)

```
wstring IHost::operatingSystem
```

Name of the host system's operating system.

5.34.1.12 OSVersion (read-only)

```
wstring IHost::OSVersion
```

Host operating system's version string.

5.34.1.13 UTCTime (read-only)

```
long long IHost::UTCTime
```

Returns the current host time in milliseconds since 1970-01-01 UTC.

5.34.1.14 Acceleration3DAvailable (read-only)

```
boolean IHost::Acceleration3DAvailable
```

Returns `true` when the host supports 3D hardware acceleration.

5.34.2 createHostOnlyNetworkInterface

```
IProgress IHost::createHostOnlyNetworkInterface(
           [out] IHostNetworkInterface hostInterface)
```

hostInterface Created host interface object.

Creates a new adapter for Host Only Networking.
If this method fails, the following error codes may be reported:

- `E_INVALIDARG`: Host network interface `name` already exists.

5.34.3 createUSBDeviceFilter

```
IHostUSBDeviceFilter IHost::createUSBDeviceFilter(
            [in] wstring name)
```

name Filter name. See IUSBDeviceFilter::name for more information.

Creates a new USB device filter. All attributes except the filter name are set to empty (any match), *active* is `false` (the filter is not active).
The created filter can be added to the list of filters using insertUSBDeviceFilter().
See also: #USBDeviceFilters

5.34.4 findHostDVDDrive

```
IMedium IHost::findHostDVDDrive(
            [in] wstring name)
```

name Name of the host drive to search for

Searches for a host DVD drive with the given `name`.
If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: Given `name` does not correspond to any host drive.

5.34.5 findHostFloppyDrive

```
IMedium IHost::findHostFloppyDrive(
            [in] wstring name)
```

name Name of the host floppy drive to search for

Searches for a host floppy drive with the given `name`.
If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: Given `name` does not correspond to any host floppy drive.

5.34.6 findHostNetworkInterfaceById

```
IHostNetworkInterface IHost::findHostNetworkInterfaceById(
            [in] uuid id)
```

id GUID of the host network interface to search for.

Searches through all host network interfaces for an interface with the given GUID.

> **Note:** The method returns an error if the given GUID does not correspond to any host network interface.

5.34.7 findHostNetworkInterfaceByName

```
IHostNetworkInterface IHost::findHostNetworkInterfaceByName(
            [in] wstring name)
```

name Name of the host network interface to search for.

Searches through all host network interfaces for an interface with the given `name`.

> **Note:** The method returns an error if the given `name` does not correspond to any host network interface.

5.34.8 findHostNetworkInterfacesOfType

```
IHostNetworkInterface[] IHost::findHostNetworkInterfacesOfType(
            [in] HostNetworkInterfaceType type)
```

type type of the host network interfaces to search for.

Searches through all host network interfaces and returns a list of interfaces of the specified type

5.34.9 findUSBDeviceByAddress

```
IHostUSBDevice IHost::findUSBDeviceByAddress(
            [in] wstring name)
```

name Address of the USB device (as assigned by the host) to search for.

Searches for a USB device with the given host address.
See also: IHostUSBDevice::address
If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: Given `name` does not correspond to any USB device.

5.34.10 findUSBDeviceById

```
IHostUSBDevice IHost::findUSBDeviceById(
            [in] uuid id)
```

id UUID of the USB device to search for.

Searches for a USB device with the given UUID.
See also: IHostUSBDevice::id
If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: Given `id` does not correspond to any USB device.

5.34.11 getProcessorCPUIDLeaf

```
void IHost::getProcessorCPUIDLeaf(
            [in] unsigned long cpuId,
            [in] unsigned long leaf,
            [in] unsigned long subLeaf,
            [out] unsigned long valEax,
            [out] unsigned long valEbx,
            [out] unsigned long valEcx,
            [out] unsigned long valEdx)
```

cpuId Identifier of the CPU. The CPU most be online.

> **Note:** The current implementation might not necessarily return the description for this exact CPU.

leaf CPUID leaf index (eax).

subLeaf CPUID leaf sub index (ecx). This currently only applies to cache information on Intel CPUs. Use 0 if retrieving values for IMachine::setCPUIDLeaf().

valEax CPUID leaf value for register eax.

valEbx CPUID leaf value for register ebx.

valEcx CPUID leaf value for register ecx.

valEdx CPUID leaf value for register edx.

Returns the CPU cpuid information for the specified leaf.

5.34.12 getProcessorDescription

```
wstring IHost::getProcessorDescription(
            [in] unsigned long cpuId)
```

cpuId Identifier of the CPU.

> **Note:** The current implementation might not necessarily return the description for this exact CPU.

Query the model string of a specified host CPU.

5.34.13 getProcessorFeature

```
boolean IHost::getProcessorFeature(
            [in] ProcessorFeature feature)
```

feature CPU Feature identifier.

Query whether a CPU feature is supported or not.

5.34.14 getProcessorSpeed

```
unsigned long IHost::getProcessorSpeed(
            [in] unsigned long cpuId)
```

cpuId Identifier of the CPU.

Query the (approximate) maximum speed of a specified host CPU in Megahertz.

5.34.15 insertUSBDeviceFilter

```
void IHost::insertUSBDeviceFilter(
            [in] unsigned long position,
            [in] IHostUSBDeviceFilter filter)
```

position Position to insert the filter to.

filter USB device filter to insert.

Inserts the given USB device to the specified position in the list of filters.
Positions are numbered starting from `0`. If the specified position is equal to or greater than the number of elements in the list, the filter is added at the end of the collection.

> **Note:** Duplicates are not allowed, so an attempt to insert a filter already in the list is an error.

> **Note:** If USB functionality is not available in the given edition of VirtualBox, this method will set the result code to E_NOTIMPL.

See also: #USBDeviceFilters
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_OBJECT_STATE: USB device filter is not created within this VirtualBox instance.
- E_INVALIDARG: USB device filter already in list.

5.34.16 removeHostOnlyNetworkInterface

```
IProgress IHost::removeHostOnlyNetworkInterface(
            [in] uuid id)
```

id Adapter GUID.

Removes the given Host Only Networking interface.
If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: No host network interface matching id found.

5.34.17 removeUSBDeviceFilter

```
void IHost::removeUSBDeviceFilter(
            [in] unsigned long position)
```

position Position to remove the filter from.

Removes a USB device filter from the specified position in the list of filters.
Positions are numbered starting from 0. Specifying a position equal to or greater than the number of elements in the list will produce an error.

> **Note:** If USB functionality is not available in the given edition of VirtualBox, this method will set the result code to E_NOTIMPL.

See also: #USBDeviceFilters
If this method fails, the following error codes may be reported:

- E_INVALIDARG: USB device filter list empty or invalid position.

5.35 IHostNetworkInterface

Represents one of host's network interfaces. IP V6 address and network mask are strings of 32 hexdecimal digits grouped by four. Groups are separated by colons. For example, fe80:0000:0000:0000:021e:c2ff:fed2:b030.

5.35.1 Attributes

5.35.1.1 name (read-only)

```
wstring IHostNetworkInterface::name
```

Returns the host network interface name.

5.35.1.2 id (read-only)

```
uuid IHostNetworkInterface::id
```

Returns the interface UUID.

5.35.1.3 networkName (read-only)

```
wstring IHostNetworkInterface::networkName
```

Returns the name of a virtual network the interface gets attached to.

5.35.1.4 dhcpEnabled (read-only)

```
boolean IHostNetworkInterface::dhcpEnabled
```

Specifies whether the DHCP is enabled for the interface.

5.35.1.5 IPAddress (read-only)

```
wstring IHostNetworkInterface::IPAddress
```

Returns the IP V4 address of the interface.

5.35.1.6 networkMask (read-only)

```
wstring IHostNetworkInterface::networkMask
```

Returns the network mask of the interface.

5.35.1.7 IPV6Supported (read-only)

```
boolean IHostNetworkInterface::IPV6Supported
```

Specifies whether the IP V6 is supported/enabled for the interface.

5.35.1.8 IPV6Address (read-only)

```
wstring IHostNetworkInterface::IPV6Address
```

Returns the IP V6 address of the interface.

5.35.1.9 IPV6NetworkMaskPrefixLength (read-only)

```
unsigned long IHostNetworkInterface::IPV6NetworkMaskPrefixLength
```

Returns the length IP V6 network mask prefix of the interface.

5.35.1.10 hardwareAddress (read-only)

```
wstring IHostNetworkInterface::hardwareAddress
```

Returns the hardware address. For Ethernet it is MAC address.

5.35.1.11 mediumType (read-only)

```
HostNetworkInterfaceMediumType IHostNetworkInterface::mediumType
```

Type of protocol encapsulation used.

5.35.1.12 status (read-only)

```
HostNetworkInterfaceStatus IHostNetworkInterface::status
```

Status of the interface.

5.35.1.13 interfaceType (read-only)

```
HostNetworkInterfaceType IHostNetworkInterface::interfaceType
```

specifies the host interface type.

5.35.2 dhcpRediscover

```
void IHostNetworkInterface::dhcpRediscover()
```

refreshes the IP configuration for dhcp-enabled interface.

5.35.3 enableDynamicIpConfig

```
void IHostNetworkInterface::enableDynamicIpConfig()
```

enables the dynamic IP configuration.

5.35.4 enableStaticIpConfig

```
void IHostNetworkInterface::enableStaticIpConfig(
            [in] wstring IPAddress,
            [in] wstring networkMask)
```

IPAddress IP address.

networkMask network mask.

sets and enables the static IP V4 configuration for the given interface.

5.35.5 enableStaticIpConfigV6

```
void IHostNetworkInterface::enableStaticIpConfigV6(
            [in] wstring IPV6Address,
            [in] unsigned long IPV6NetworkMaskPrefixLength)
```

IPV6Address IP address.

IPV6NetworkMaskPrefixLength network mask.

sets and enables the static IP V6 configuration for the given interface.

5.36 IHostPciDevicePlugEvent (IMachineEvent)

> **Note:** This interface extends IMachineEvent and therefore supports all its methods and attributes as well.

Notification when host PCI device is plugged/unplugged.

5.36.1 Attributes

5.36.1.1 plugged (read-only)

`boolean IHostPciDevicePlugEvent::plugged`

If device successfully plugged or unplugged.

5.36.1.2 success (read-only)

`boolean IHostPciDevicePlugEvent::success`

If operation was successful, if false - 'message' attribute may be of interest.

5.36.1.3 attachment (read-only)

`IPciDeviceAttachment IHostPciDevicePlugEvent::attachment`

Attachment info for this device.

5.36.1.4 eventContext (read-only)

`IEventContext IHostPciDevicePlugEvent::eventContext`

Context object, passed into attachHostPciDevice() and attachHostPciDevice().

5.36.1.5 message (read-only)

`wstring IHostPciDevicePlugEvent::message`

Optional error message.

5.37 IHostUSBDevice (IUSBDevice)

> **Note:** This interface extends IUSBDevice and therefore supports all its methods and attributes as well.

The IHostUSBDevice interface represents a physical USB device attached to the host computer.
Besides properties inherited from IUSBDevice, this interface adds the state property that holds the current state of the USB device.
See also: IHost::USBDevices, IHost::USBDeviceFilters

5.37.1 Attributes

5.37.1.1 state (read-only)

`USBDeviceState IHostUSBDevice::state`

Current state of the device.

5.38 IHostUSBDeviceFilter (IUSBDeviceFilter)

> **Note:** This interface extends IUSBDeviceFilter and therefore supports all its methods and attributes as well.

The IHostUSBDeviceFilter interface represents a global filter for a physical USB device used by the host computer. Used indirectly in IHost::USBDeviceFilters[].

Using filters of this type, the host computer determines the initial state of the USB device after it is physically attached to the host's USB controller.

> **Note:** The IUSBDeviceFilter::remote attribute is ignored by this type of filters, because it makes sense only for machine USB filters.

See also: IHost::USBDeviceFilters

5.38.1 Attributes

5.38.1.1 action (read/write)

```
USBDeviceFilterAction IHostUSBDeviceFilter::action
```

Action performed by the host when an attached USB device matches this filter.

5.39 IInternalMachineControl

> **Note:** This interface is not supported in the web service.

5.39.1 adoptSavedState

```
void IInternalMachineControl::adoptSavedState(
           [in] wstring savedStateFile)
```

savedStateFile Path to the saved state file to adopt.

Gets called by IConsole::adoptSavedState.

If this method fails, the following error codes may be reported:

- `VBOX_E_FILE_ERROR:` Invalid saved state file path.

5.39.2 autoCaptureUSBDevices

```
void IInternalMachineControl::autoCaptureUSBDevices()
```

Requests a capture all matching USB devices attached to the host. When the request is completed, the VM process will get a IInternalSessionControl::onUSBDeviceAttach() notification per every captured device.

5.39.3 beginPowerUp

```
void IInternalMachineControl::beginPowerUp(
           [in] IProgress aProgress)
```

aProgress

Tells VBoxSVC that IConsole::powerUp() is under ways and gives it the progress object that should be part of any pending IMachine::launchVMProcess() operations. The progress object may be called back to reflect an early cancelation, so some care have to be taken with respect to any cancelation callbacks. The console object will call endPowerUp() to signal the completion of the progress object.

5.39.4 beginPoweringDown

```
void IInternalMachineControl::beginPoweringDown(
           [out] IProgress progress)
```

progress Progress object created by VBoxSVC to wait until the VM is powered down.

Called by the VM process to inform the server it wants to stop the VM execution and power down.

5.39.5 beginSavingState

```
void IInternalMachineControl::beginSavingState(
           [out] IProgress progress,
           [out] wstring stateFilePath)
```

progress Progress object created by VBoxSVC to wait until the state is saved.

stateFilePath File path the VM process must save the execution state to.

Called by the VM process to inform the server it wants to save the current state and stop the VM execution.

5.39.6 beginTakingSnapshot

```
void IInternalMachineControl::beginTakingSnapshot(
           [in] IConsole initiator,
           [in] wstring name,
           [in] wstring description,
           [in] IProgress consoleProgress,
           [in] boolean fTakingSnapshotOnline,
           [out] wstring stateFilePath)
```

initiator The console object that initiated this call.

name Snapshot name.

description Snapshot description.

consoleProgress Progress object created by the VM process tracking the snapshot's progress. This has the following sub-operations:

- setting up (weight 1);
- one for each medium attachment that needs a differencing image (weight 1 each);

- another one to copy the VM state (if offline with saved state, weight is VM memory size in MB);
- another one to save the VM state (if online, weight is VM memory size in MB);
- finishing up (weight 1)

fTakingSnapshotOnline Whether this is an online snapshot (i.e. the machine is running).

stateFilePath File path the VM process must save the execution state to.

Called from the VM process to request from the server to perform the server-side actions of creating a snapshot (creating differencing images and the snapshot object).

If this method fails, the following error codes may be reported:

- `VBOX_E_FILE_ERROR:` Settings file not accessible.
- `VBOX_E_XML_ERROR:` Could not parse the settings file.

5.39.7 captureUSBDevice

```
void IInternalMachineControl::captureUSBDevice(
            [in] uuid id)
```

id

Requests a capture of the given host USB device. When the request is completed, the VM process will get a IInternalSessionControl::onUSBDeviceAttach() notification.

5.39.8 deleteSnapshot

```
IProgress IInternalMachineControl::deleteSnapshot(
            [in] IConsole initiator,
            [in] uuid id,
            [out] MachineState machineState)
```

initiator The console object that initiated this call.

id UUID of the snapshot to delete.

machineState New machine state after this operation is started.

Gets called by IConsole::deleteSnapshot.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_OBJECT_STATE:` Snapshot has more than one child snapshot.

5.39.9 detachAllUSBDevices

```
void IInternalMachineControl::detachAllUSBDevices(
            [in] boolean done)
```

done

Notification that a VM that is being powered down. The done parameter indicates whether which stage of the power down we're at. When `done = false` the VM is announcing its intentions, while when `done = true` the VM is reporting what it has done.

> **Note:** In the `done = true` case, the server must run its own filters and filters of all VMs but this one on all detach devices as if they were just attached to the host computer.

5.39.10 detachUSBDevice

```
void IInternalMachineControl::detachUSBDevice(
            [in] uuid id,
            [in] boolean done)
```

id

done

Notification that a VM is going to detach (`done = false`) or has already detached (`done = true`) the given USB device. When the `done = true` request is completed, the VM process will get a IInternalSessionControl::onUSBDeviceDetach() notification.

> **Note:** In the `done = true` case, the server must run its own filters and filters of all VMs but this one on the detached device as if it were just attached to the host computer.

5.39.11 endPowerUp

```
void IInternalMachineControl::endPowerUp(
            [in] long result)
```

result

Tells VBoxSVC that IConsole::powerUp() has completed. This method may query status information from the progress object it received in beginPowerUp() and copy it over to any in-progress IMachine::launchVMProcess() call in order to complete that progress object.

5.39.12 endPoweringDown

```
void IInternalMachineControl::endPoweringDown(
            [in] long result,
            [in] wstring errMsg)
```

result `S_OK` to indicate success.

errMsg `human` readable error message in case of failure.

Called by the VM process to inform the server that powering down previously requested by #beginPoweringDown is either successfully finished or there was a failure.

If this method fails, the following error codes may be reported:

- `VBOX_E_FILE_ERROR`: Settings file not accessible.
- `VBOX_E_XML_ERROR`: Could not parse the settings file.

5.39.13 endSavingState

```
void IInternalMachineControl::endSavingState(
            [in] long result,
            [in] wstring errMsg)
```

result `S_OK` to indicate success.

errMsg `human` readable error message in case of failure.

Called by the VM process to inform the server that saving the state previously requested by #beginSavingState is either successfully finished or there was a failure.
If this method fails, the following error codes may be reported:

- VBOX_E_FILE_ERROR: Settings file not accessible.
- VBOX_E_XML_ERROR: Could not parse the settings file.

5.39.14 endTakingSnapshot

```
void IInternalMachineControl::endTakingSnapshot(
           [in] boolean success)
```

success true to indicate success and false otherwise

Called by the VM process to inform the server that the snapshot previously requested by #beginTakingSnapshot is either successfully taken or there was a failure.

5.39.15 finishOnlineMergeMedium

```
void IInternalMachineControl::finishOnlineMergeMedium(
           [in] IMediumAttachment mediumAttachment,
           [in] IMedium source,
           [in] IMedium target,
           [in] boolean mergeForward,
           [in] IMedium parentForTarget,
           [in] IMedium childrenToReparent[])
```

mediumAttachment The medium attachment which needs to be cleaned up.

source Merge source medium.

target Merge target medium.

mergeForward Merge direction.

parentForTarget For forward merges: new parent for target medium.

childrenToReparent For backward merges: list of media which need their parent UUID updated.

Gets called by IConsole::onlineMergeMedium.

5.39.16 getIPCId

```
wstring IInternalMachineControl::getIPCId()
```

5.39.17 lockMedia

```
void IInternalMachineControl::lockMedia()
```

Locks all media attached to the machine for writing and parents of attached differencing media (if any) for reading. This operation is atomic so that if it fails no media is actually locked.
This method is intended to be called when the machine is in Starting or Restoring state. The locked media will be automatically unlocked when the machine is powered off or crashed.

5.39.18 onSessionEnd

```
IProgress IInternalMachineControl::onSessionEnd(
            [in] ISession session)
```

session Session that is being closed

Triggered by the given session object when the session is about to close normally.

5.39.19 pullGuestProperties

```
void IInternalMachineControl::pullGuestProperties(
            [out] wstring name[],
            [out] wstring value[],
            [out] long long timestamp[],
            [out] wstring flags[])
```

name The names of the properties returned.

value The values of the properties returned. The array entries match the corresponding entries in the `name` array.

timestamp The time stamps of the properties returned. The array entries match the corresponding entries in the `name` array.

flags The flags of the properties returned. The array entries match the corresponding entries in the `name` array.

Get the list of the guest properties matching a set of patterns along with their values, time stamps and flags and give responsibility for managing properties to the console.

5.39.20 pushGuestProperty

```
void IInternalMachineControl::pushGuestProperty(
            [in] wstring name,
            [in] wstring value,
            [in] long long timestamp,
            [in] wstring flags)
```

name The name of the property to be updated.

value The value of the property.

timestamp The timestamp of the property.

flags The flags of the property.

Update a single guest property in IMachine.

5.39.21 restoreSnapshot

```
IProgress IInternalMachineControl::restoreSnapshot(
            [in] IConsole initiator,
            [in] ISnapshot snapshot,
            [out] MachineState machineState)
```

initiator The console object that initiated this call.

snapshot The snapshot to restore the VM state from.

machineState New machine state after this operation is started.

Gets called by IConsole::RestoreSnapshot.

5.39.22 runUSBDeviceFilters

```
void IInternalMachineControl::runUSBDeviceFilters(
            [in] IUSBDevice device,
            [out] boolean matched,
            [out] unsigned long maskedInterfaces)
```

device

matched

maskedInterfaces

Asks the server to run USB devices filters of the associated machine against the given USB device and tell if there is a match.

> **Note:** Intended to be used only for remote USB devices. Local ones don't require to call this method (this is done implicitly by the Host and USBProxyService).

5.39.23 setRemoveSavedStateFile

```
void IInternalMachineControl::setRemoveSavedStateFile(
            [in] boolean aRemove)
```

aRemove

Updates the flag whether the saved state file is removed on a machine state change from Saved to PoweredOff.

5.39.24 unlockMedia

```
void IInternalMachineControl::unlockMedia()
```

Unlocks all media previously locked using lockMedia().

This method is intended to be used with teleportation so that it is possible to teleport between processes on the same machine.

5.39.25 updateState

```
void IInternalMachineControl::updateState(
            [in] MachineState state)
```

state

Updates the VM state.

> **Note:** This operation will also update the settings file with the correct information about the saved state file and delete this file from disk when appropriate.

5.40 IInternalSessionControl

> **Note:** This interface is not supported in the web service.

5.40.1 accessGuestProperty

```
void IInternalSessionControl::accessGuestProperty(
            [in] wstring name,
            [in] wstring value,
            [in] wstring flags,
            [in] boolean isSetter,
            [out] wstring retValue,
            [out] long long retTimestamp,
            [out] wstring retFlags)
```

name

value

flags

isSetter

retValue

retTimestamp

retFlags

Called by IMachine::getGuestProperty() and by IMachine::setGuestProperty() in order to read and modify guest properties.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Machine session is not open.
- VBOX_E_INVALID_OBJECT_STATE: Session type is not direct.

5.40.2 assignMachine

```
void IInternalSessionControl::assignMachine(
            [in] IMachine machine)
```

machine

Assigns the machine object associated with this direct-type session or informs the session that it will be a remote one (if `machine == null`).

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.3 assignRemoteMachine

```
void IInternalSessionControl::assignRemoteMachine(
            [in] IMachine machine,
            [in] IConsole console)
```

machine

console

Assigns the machine and the (remote) console object associated with this remote-type session.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.

5.40.4 enumerateGuestProperties

```
void IInternalSessionControl::enumerateGuestProperties(
            [in] wstring patterns,
            [out] wstring key[],
            [out] wstring value[],
            [out] long long timestamp[],
            [out] wstring flags[])
```

patterns The patterns to match the properties against as a comma-separated string. If this is empty, all properties currently set will be returned.

key The key names of the properties returned.

value The values of the properties returned. The array entries match the corresponding entries in the key array.

timestamp The time stamps of the properties returned. The array entries match the corresponding entries in the key array.

flags The flags of the properties returned. The array entries match the corresponding entries in the key array.

Return a list of the guest properties matching a set of patterns along with their values, time stamps and flags.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Machine session is not open.
- VBOX_E_INVALID_OBJECT_STATE: Session type is not direct.

5.40.5 getPID

```
unsigned long IInternalSessionControl::getPID()
```

PID of the process that has created this Session object.

5.40.6 getRemoteConsole

```
IConsole IInternalSessionControl::getRemoteConsole()
```

Returns the console object suitable for remote control.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.7 onBandwidthGroupChange

```
void IInternalSessionControl::onBandwidthGroupChange(
            [in] IBandwidthGroup bandwidthGroup)
```

bandwidthGroup The bandwidth group which changed.

Notification when one of the bandwidth groups change.

5.40.8 onCPUChange

```
void IInternalSessionControl::onCPUChange(
            [in] unsigned long cpu,
            [in] boolean add)
```

cpu The CPU which changed

add Flag whether the CPU was added or removed

Notification when a CPU changes.

5.40.9 onCPUExecutionCapChange

```
void IInternalSessionControl::onCPUExecutionCapChange(
            [in] unsigned long executionCap)
```

executionCap The new CPU execution cap value. (1-100)

Notification when the CPU execution cap changes.

5.40.10 onMediumChange

```
void IInternalSessionControl::onMediumChange(
            [in] IMediumAttachment mediumAttachment,
            [in] boolean force)
```

mediumAttachment

force

Triggered when attached media of the associated virtual machine have changed.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.11 onNetworkAdapterChange

```
void IInternalSessionControl::onNetworkAdapterChange(
            [in] INetworkAdapter networkAdapter,
            [in] boolean changeAdapter)
```

networkAdapter

changeAdapter

Triggered when settings of a network adapter of the associated virtual machine have changed.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.12 onParallelPortChange

```
void IInternalSessionControl::onParallelPortChange(
            [in] IParallelPort parallelPort)
```

parallelPort

Triggered when settings of a parallel port of the associated virtual machine have changed.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.13 onSerialPortChange

```
void IInternalSessionControl::onSerialPortChange(
            [in] ISerialPort serialPort)
```

serialPort

Triggered when settings of a serial port of the associated virtual machine have changed.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.14 onSharedFolderChange

```
void IInternalSessionControl::onSharedFolderChange(
            [in] boolean global)
```

global

Triggered when a permanent (global or machine) shared folder has been created or removed.

> **Note:** We don't pass shared folder parameters in this notification because the order in which parallel notifications are delivered is not defined, therefore it could happen that these parameters were outdated by the time of processing this notification.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.15 onShowWindow

```
void IInternalSessionControl::onShowWindow(
            [in] boolean check,
            [out] boolean canShow,
            [out] long long winId)
```

check

canShow

winId

Called by IMachine::canShowConsoleWindow() and by IMachine::showConsoleWindow() in order to notify console listeners ICanShowWindowEvent and IShowWindowEvent.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.16 onStorageControllerChange

```
void IInternalSessionControl::onStorageControllerChange()
```

Triggered when settings of a storage controller of the associated virtual machine have changed.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.17 onUSBControllerChange

```
void IInternalSessionControl::onUSBControllerChange()
```

Triggered when settings of the USB controller object of the associated virtual machine have changed.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.18 onUSBDeviceAttach

```
void IInternalSessionControl::onUSBDeviceAttach(
            [in] IUSBDevice device,
            [in] IVirtualBoxErrorInfo error,
            [in] unsigned long maskedInterfaces)
```

device

error

maskedInterfaces

Triggered when a request to capture a USB device (as a result of matched USB filters or direct call to IConsole::attachUSBDevice()) has completed. A nullerror object means success, otherwise it describes a failure.

If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Session state prevents operation.
- VBOX_E_INVALID_OBJECT_STATE: Session type prevents operation.

5.40.19 onUSBDeviceDetach

```
void IInternalSessionControl::onUSBDeviceDetach(
            [in] uuid id,
            [in] IVirtualBoxErrorInfo error)
```

id

error

Triggered when a request to release the USB device (as a result of machine termination or direct call to IConsole::detachUSBDevice()) has completed. A `nullerror` object means success, otherwise it describes a failure.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Session state prevents operation.
- `VBOX_E_INVALID_OBJECT_STATE:` Session type prevents operation.

5.40.20 onVRDEServerChange

```
void IInternalSessionControl::onVRDEServerChange(
            [in] boolean restart)
```

restart Flag whether the server must be restarted

Triggered when settings of the VRDE server object of the associated virtual machine have changed.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Session state prevents operation.
- `VBOX_E_INVALID_OBJECT_STATE:` Session type prevents operation.

5.40.21 onlineMergeMedium

```
void IInternalSessionControl::onlineMergeMedium(
            [in] IMediumAttachment mediumAttachment,
            [in] unsigned long sourceIdx,
            [in] unsigned long targetIdx,
            [in] IMedium source,
            [in] IMedium target,
            [in] boolean mergeForward,
            [in] IMedium parentForTarget,
            [in] IMedium childrenToReparent[],
            [in] IProgress progress)
```

mediumAttachment The medium attachment to identify the medium chain.

sourceIdx The index of the source image in the chain. Redundant, but drastically reduces IPC.

targetIdx The index of the target image in the chain. Redundant, but drastically reduces IPC.

source Merge source medium.

target Merge target medium.

mergeForward Merge direction.

parentForTarget For forward merges: new parent for target medium.

childrenToReparent For backward merges: list of media which need their parent UUID updated.

progress Progress object for this operation.

Triggers online merging of a hard disk. Used internally when deleting a snapshot while a VM referring to the same hard disk chain is running.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Machine session is not open.
- `VBOX_E_INVALID_OBJECT_STATE`: Session type is not direct.

5.40.22 uninitialize

```
void IInternalSessionControl::uninitialize()
```

Uninitializes (closes) this session. Used by VirtualBox to close the corresponding remote session when the direct session dies or gets closed.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Session state prevents operation.

5.40.23 updateMachineState

```
void IInternalSessionControl::updateMachineState(
          [in] MachineState aMachineState)
```

aMachineState

Updates the machine state in the VM process. Must be called only in certain cases (see the method implementation).

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Session state prevents operation.
- `VBOX_E_INVALID_OBJECT_STATE`: Session type prevents operation.

5.41 IKeyboard

The IKeyboard interface represents the virtual machine's keyboard. Used in IConsole::keyboard.

Use this interface to send keystrokes or the Ctrl-Alt-Del sequence to the virtual machine.

5.41.1 Attributes

5.41.1.1 eventSource (read-only)

```
IEventSource IKeyboard::eventSource
```

Event source for keyboard events.

5.41.2 putCAD

```
void IKeyboard::putCAD()
```

Sends the Ctrl-Alt-Del sequence to the keyboard. This function is nothing special, it is just a convenience function calling putScancodes() with the proper scancodes.

If this method fails, the following error codes may be reported:

- `VBOX_E_IPRT_ERROR`: Could not send all scan codes to virtual keyboard.

5.41.3 putScancode

```
void IKeyboard::putScancode(
            [in] long scancode)
```

scancode

Sends a scancode to the keyboard.
If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Could not send scan code to virtual keyboard.

5.41.4 putScancodes

```
unsigned long IKeyboard::putScancodes(
            [in] long scancodes[])
```

scancodes

Sends an array of scancodes to the keyboard.
If this method fails, the following error codes may be reported:

- VBOX_E_IPRT_ERROR: Could not send all scan codes to virtual keyboard.

5.42 IKeyboardLedsChangedEvent (IEvent)

Note: This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when the guest OS executes the KBD_CMD_SET_LEDS command to alter the state of the keyboard LEDs.

5.42.1 Attributes

5.42.1.1 numLock (read-only)

```
boolean IKeyboardLedsChangedEvent::numLock
```

NumLock status.

5.42.1.2 capsLock (read-only)

```
boolean IKeyboardLedsChangedEvent::capsLock
```

CapsLock status.

5.42.1.3 scrollLock (read-only)

```
boolean IKeyboardLedsChangedEvent::scrollLock
```

ScrollLock status.

5.43 IMachine

The IMachine interface represents a virtual machine, or guest, created in VirtualBox.

This interface is used in two contexts. First of all, a collection of objects implementing this interface is stored in the IVirtualBox::machines[] attribute which lists all the virtual machines that are currently registered with this VirtualBox installation. Also, once a session has been opened for the given virtual machine (e.g. the virtual machine is running), the machine object associated with the open session can be queried from the session object; see ISession for details.

The main role of this interface is to expose the settings of the virtual machine and provide methods to change various aspects of the virtual machine's configuration. For machine objects stored in the IVirtualBox::machines[] collection, all attributes are read-only unless explicitly stated otherwise in individual attribute and method descriptions.

In order to change a machine setting, a session for this machine must be opened using one of the lockMachine() or launchVMProcess() methods. After the machine has been successfully locked for a session, a mutable machine object needs to be queried from the session object and then the desired settings changes can be applied to the returned object using IMachine attributes and methods. See the ISession interface description for more information about sessions.

Note that IMachine does not provide methods to control virtual machine execution (such as start the machine, or power it down) – these methods are grouped in a separate interface called IConsole.

See also: ISession, IConsole

5.43.1 Attributes

5.43.1.1 parent (read-only)

`IVirtualBox IMachine::parent`

Associated parent object.

5.43.1.2 accessible (read-only)

`boolean IMachine::accessible`

Whether this virtual machine is currently accessible or not.

A machine is always deemed accessible unless it is registered *and* its settings file cannot be read or parsed (either because the file itself is unavailable or has invalid XML contents).

Every time this property is read, the accessibility state of this machine is re-evaluated. If the returned value is `false`, the accessError property may be used to get the detailed error information describing the reason of inaccessibility, including XML error messages.

When the machine is inaccessible, only the following properties can be used on it:

- parent
- id
- settingsFilePath
- accessible
- accessError

An attempt to access any other property or method will return an error.

The only possible action you can perform on an inaccessible machine is to unregister it using the unregister() call (or, to check for the accessibility state once more by querying this property).

Note: In the current implementation, once this property returns `true`, the machine will never become inaccessible later, even if its settings file cannot be successfully read/written any more (at least, until the VirtualBox server is restarted). This limitation may be removed in future releases.

5.43.1.3 accessError (read-only)

`IVirtualBoxErrorInfo IMachine::accessError`

Error information describing the reason of machine inaccessibility.

Reading this property is only valid after the last call to accessible returned `false` (i.e. the machine is currently inaccessible). Otherwise, a `null` IVirtualBoxErrorInfo object will be returned.

5.43.1.4 name (read/write)

`wstring IMachine::name`

Name of the virtual machine.

Besides being used for human-readable identification purposes everywhere in VirtualBox, the virtual machine name is also used as a name of the machine's settings file and as a name of the subdirectory this settings file resides in. Thus, every time you change the value of this property, the settings file will be renamed once you call saveSettings() to confirm the change. The containing subdirectory will be also renamed, but only if it has exactly the same name as the settings file itself prior to changing this property (for backward compatibility with previous API releases). The above implies the following limitations:

- The machine name cannot be empty.
- The machine name can contain only characters that are valid file name characters according to the rules of the file system used to store VirtualBox configuration.
- You cannot have two or more machines with the same name if they use the same subdirectory for storing the machine settings files.
- You cannot change the name of the machine if it is running, or if any file in the directory containing the settings file is being used by another running machine or by any other process in the host operating system at a time when saveSettings() is called.

If any of the above limitations are hit, saveSettings() will return an appropriate error message explaining the exact reason and the changes you made to this machine will not be saved.

Starting with VirtualBox 4.0, a ".vbox" extension of the settings file is recommended, but not enforced. (Previous versions always used a generic ".xml" extension.)

5.43.1.5 description (read/write)

`wstring IMachine::description`

Description of the virtual machine.

The description attribute can contain any text and is typically used to describe the hardware and software configuration of the virtual machine in detail (i.e. network settings, versions of the installed software and so on).

5.43.1.6 id (read-only)

`uuid IMachine::id`

UUID of the virtual machine.

5.43.1.7 OSTypeId (read/write)

```
wstring IMachine::OSTypeId
```

User-defined identifier of the Guest OS type. You may use IVirtualBox::getGuestOSType() to obtain an IGuestOSType object representing details about the given Guest OS type.

> **Note:** This value may differ from the value returned by IGuest::OSTypeId if Guest Additions are installed to the guest OS.

5.43.1.8 HardwareVersion (read/write)

```
wstring IMachine::HardwareVersion
```

Hardware version identifier. Internal use only for now.

5.43.1.9 hardwareUUID (read/write)

```
uuid IMachine::hardwareUUID
```

The UUID presented to the guest via memory tables, hardware and guest properties. For most VMs this is the same as the `id`, but for VMs which have been cloned or teleported it may be the same as the source VM. This latter is because the guest shouldn't notice that it was cloned or teleported.

5.43.1.10 CPUCount (read/write)

```
unsigned long IMachine::CPUCount
```

Number of virtual CPUs in the VM.

5.43.1.11 CPUHotPlugEnabled (read/write)

```
boolean IMachine::CPUHotPlugEnabled
```

This setting determines whether VirtualBox allows CPU hotplugging for this machine.

5.43.1.12 CPUExecutionCap (read/write)

```
unsigned long IMachine::CPUExecutionCap
```

Means to limit the number of CPU cycles a guest can use. The unit is percentage of host CPU cycles per second. The valid range is 1 - 100. 100 (the default) implies no limit.

5.43.1.13 memorySize (read/write)

```
unsigned long IMachine::memorySize
```

System memory size in megabytes.

5.43.1.14 memoryBalloonSize (read/write)

```
unsigned long IMachine::memoryBalloonSize
```

Memory balloon size in megabytes.

5.43.1.15 PageFusionEnabled (read/write)

`boolean IMachine::PageFusionEnabled`

This setting determines whether VirtualBox allows page fusion for this machine (64 bits host only).

5.43.1.16 VRAMSize (read/write)

`unsigned long IMachine::VRAMSize`

Video memory size in megabytes.

5.43.1.17 accelerate3DEnabled (read/write)

`boolean IMachine::accelerate3DEnabled`

This setting determines whether VirtualBox allows this machine to make use of the 3D graphics support available on the host.

5.43.1.18 accelerate2DVideoEnabled (read/write)

`boolean IMachine::accelerate2DVideoEnabled`

This setting determines whether VirtualBox allows this machine to make use of the 2D video acceleration support available on the host.

5.43.1.19 monitorCount (read/write)

`unsigned long IMachine::monitorCount`

Number of virtual monitors.

> **Note:** Only effective on Windows XP and later guests with Guest Additions installed.

5.43.1.20 BIOSSettings (read-only)

`IBIOSSettings IMachine::BIOSSettings`

Object containing all BIOS settings.

5.43.1.21 firmwareType (read/write)

`FirmwareType IMachine::firmwareType`

Type of firmware (such as legacy BIOS or EFI), used for initial bootstrap in this VM.

5.43.1.22 pointingHidType (read/write)

`PointingHidType IMachine::pointingHidType`

Type of pointing HID (such as mouse or tablet) used in this VM. The default is typically "PS2Mouse" but can vary depending on the requirements of the guest operating system.

5.43.1.23 keyboardHidType (read/write)

`KeyboardHidType IMachine::keyboardHidType`

Type of keyboard HID used in this VM. The default is typically "PS2Keyboard" but can vary depending on the requirements of the guest operating system.

5.43.1.24 hpetEnabled (read/write)

`boolean IMachine::hpetEnabled`

This attribute controls if High Precision Event Timer (HPET) is enabled in this VM. Use this property if you want to provide guests with additional time source, or if guest requires HPET to function correctly. Default is false.

5.43.1.25 chipsetType (read/write)

`ChipsetType IMachine::chipsetType`

Chipset type used in this VM.

5.43.1.26 snapshotFolder (read/write)

`wstring IMachine::snapshotFolder`

Full path to the directory used to store snapshot data (differencing media and saved state files) of this machine.

The initial value of this property is <path_to_settings_file>/<machine_uuid>.

Currently, it is an error to try to change this property on a machine that has snapshots (because this would require to move possibly large files to a different location). A separate method will be available for this purpose later.

> **Note:** Setting this property to `null` or to an empty string will restore the initial value.

> **Note:** When setting this property, the specified path can be absolute (full path) or relative to the directory where the machine settings file is located. When reading this property, a full path is always returned.

> **Note:** The specified path may not exist, it will be created when necessary.

5.43.1.27 VRDEServer (read-only)

`IVRDEServer IMachine::VRDEServer`

VirtualBox Remote Desktop Extension (VRDE) server object.

5.43.1.28 mediumAttachments (read-only)

`IMediumAttachment IMachine::mediumAttachments[]`

Array of media attached to this machine.

5.43.1.29 USBController (read-only)

`IUSBController IMachine::USBController`

Associated USB controller object.

> **Note:** If USB functionality is not available in the given edition of VirtualBox, this method will set the result code to `E_NOTIMPL`.

5.43.1.30 audioAdapter (read-only)

`IAudioAdapter IMachine::audioAdapter`

Associated audio adapter, always present.

5.43.1.31 storageControllers (read-only)

`IStorageController IMachine::storageControllers[]`

Array of storage controllers attached to this machine.

5.43.1.32 settingsFilePath (read-only)

`wstring IMachine::settingsFilePath`

Full name of the file containing machine settings data.

5.43.1.33 settingsModified (read-only)

`boolean IMachine::settingsModified`

Whether the settings of this machine have been modified (but neither yet saved nor discarded).

> **Note:** Reading this property is only valid on instances returned by ISession::machine and on new machines created by IVirtualBox::createMachine() or opened by IVirtualBox::openMachine() but not yet registered, or on unregistered machines after calling unregister(). For all other cases, the settings can never be modified.

> **Note:** For newly created unregistered machines, the value of this property is always `true` until saveSettings() is called (no matter if any machine settings have been changed after the creation or not). For opened machines the value is set to `false` (and then follows to normal rules).

5.43.1.34 sessionState (read-only)

`SessionState IMachine::sessionState`

Current session state for this machine.

5.43.1.35 sessionType (read-only)

```
wstring IMachine::sessionType
```

Type of the session. If sessionState is Spawning or Locked, this attribute contains the same value as passed to the launchVMProcess() method in the `type` parameter. If the session was used with lockMachine(), or if sessionState is SessionClosed, the value of this attribute is an empty string.

5.43.1.36 sessionPid (read-only)

```
unsigned long IMachine::sessionPid
```

Identifier of the session process. This attribute contains the platform-dependent identifier of the process whose session was used with lockMachine() call. The returned value is only valid if sessionState is Locked or Unlocking by the time this property is read.

5.43.1.37 state (read-only)

```
MachineState IMachine::state
```

Current execution state of this machine.

5.43.1.38 lastStateChange (read-only)

```
long long IMachine::lastStateChange
```

Time stamp of the last execution state change, in milliseconds since 1970-01-01 UTC.

5.43.1.39 stateFilePath (read-only)

```
wstring IMachine::stateFilePath
```

Full path to the file that stores the execution state of the machine when it is in the Saved state.

> **Note:** When the machine is not in the Saved state, this attribute is an empty string.

5.43.1.40 logFolder (read-only)

```
wstring IMachine::logFolder
```

Full path to the folder that stores a set of rotated log files recorded during machine execution. The most recent log file is named `VBox.log`, the previous log file is named `VBox.log.1` and so on (up to `VBox.log.3` in the current version).

5.43.1.41 currentSnapshot (read-only)

```
ISnapshot IMachine::currentSnapshot
```

Current snapshot of this machine. This is `null` if the machine currently has no snapshots. If it is not `null`, then it was set by one of IConsole::takeSnapshot(), IConsole::deleteSnapshot() or IConsole::restoreSnapshot(), depending on which was called last. See ISnapshot for details.

5.43.1.42 snapshotCount (read-only)

`unsigned long IMachine::snapshotCount`

Number of snapshots taken on this machine. Zero means the machine doesn't have any snapshots.

5.43.1.43 currentStateModified (read-only)

`boolean IMachine::currentStateModified`

Returns `true` if the current state of the machine is not identical to the state stored in the current snapshot.

The current state is identical to the current snapshot only directly after one of the following calls are made:

- IConsole::restoreSnapshot()
- IConsole::takeSnapshot() (issued on a "powered off" or "saved" machine, for which settingsModified returns `false`)

The current state remains identical until one of the following happens:

- settings of the machine are changed
- the saved state is deleted
- the current snapshot is deleted
- an attempt to execute the machine is made

Note: For machines that don't have snapshots, this property is always `false`.

5.43.1.44 sharedFolders (read-only)

`ISharedFolder IMachine::sharedFolders[]`

Collection of shared folders for this machine (permanent shared folders). These folders are shared automatically at machine startup and available only to the guest OS installed within this machine.

New shared folders are added to the collection using createSharedFolder(). Existing shared folders can be removed using removeSharedFolder().

5.43.1.45 clipboardMode (read/write)

`ClipboardMode IMachine::clipboardMode`

Synchronization mode between the host OS clipboard and the guest OS clipboard.

5.43.1.46 guestPropertyNotificationPatterns (read/write)

`wstring IMachine::guestPropertyNotificationPatterns`

A comma-separated list of simple glob patterns. Changes to guest properties whose name matches one of the patterns will generate an IGuestPropertyChangedEvent signal.

5.43.1.47 teleporterEnabled (read/write)

```
boolean IMachine::teleporterEnabled
```

When set to `true`, the virtual machine becomes a target teleporter the next time it is powered on. This can only set to `true` when the VM is in the `PoweredOff` or `Aborted` state.

5.43.1.48 teleporterPort (read/write)

```
unsigned long IMachine::teleporterPort
```

The TCP port the target teleporter will listen for incoming teleportations on.
0 means the port is automatically selected upon power on. The actual value can be read from this property while the machine is waiting for incoming teleportations.

5.43.1.49 teleporterAddress (read/write)

```
wstring IMachine::teleporterAddress
```

The address the target teleporter will listen on. If set to an empty string, it will listen on all addresses.

5.43.1.50 teleporterPassword (read/write)

```
wstring IMachine::teleporterPassword
```

The password to check for on the target teleporter. This is just a very basic measure to prevent simple hacks and operators accidentally beaming a virtual machine to the wrong place.

5.43.1.51 faultToleranceState (read/write)

```
FaultToleranceState IMachine::faultToleranceState
```

Fault tolerance state; disabled, source or target. This property can be changed at any time. If you change it for a running VM, then the fault tolerance address and port must be set beforehand.

5.43.1.52 faultTolerancePort (read/write)

```
unsigned long IMachine::faultTolerancePort
```

The TCP port the fault tolerance source or target will use for communication.

5.43.1.53 faultToleranceAddress (read/write)

```
wstring IMachine::faultToleranceAddress
```

The address the fault tolerance source or target.

5.43.1.54 faultTolerancePassword (read/write)

```
wstring IMachine::faultTolerancePassword
```

The password to check for on the standby VM. This is just a very basic measure to prevent simple hacks and operators accidentally choosing the wrong standby VM.

5.43.1.55 faultToleranceSyncInterval (read/write)

`unsigned long IMachine::faultToleranceSyncInterval`

The interval in ms used for syncing the state between source and target.

5.43.1.56 RTCUseUTC (read/write)

`boolean IMachine::RTCUseUTC`

When set to `true`, the RTC device of the virtual machine will run in UTC time, otherwise in local time. Especially Unix guests prefer the time in UTC.

5.43.1.57 ioCacheEnabled (read/write)

`boolean IMachine::ioCacheEnabled`

When set to `true`, the builtin I/O cache of the virtual machine will be enabled.

5.43.1.58 ioCacheSize (read/write)

`unsigned long IMachine::ioCacheSize`

Maximum size of the I/O cache in MB.

5.43.1.59 bandwidthControl (read-only)

`IBandwidthControl IMachine::bandwidthControl`

Bandwidth control manager.

5.43.1.60 pciDeviceAssignments (read-only)

`IPciDeviceAttachment IMachine::pciDeviceAssignments[]`

Array of PCI devices assigned to this machine, to get list of all PCI devices attached to the machine use IConsole::attachedPciDevices attribute, as this attribute is intended to list only devices additional to what described in virtual hardware config. Usually, this list keeps host's physical devices assigned to the particular machine.

5.43.2 addStorageController

```
IStorageController IMachine::addStorageController(
            [in] wstring name,
            [in] StorageBus connectionType)
```

name

connectionType

Adds a new storage controller (SCSI, SAS or SATA controller) to the machine and returns it as an instance of IStorageController.

`name` identifies the controller for subsequent calls such as getStorageControllerByName(), getStorageControllerByInstance(), removeStorageController(), attachDevice() or mountMedium().

After the controller has been added, you can set its exact type by setting the IStorageController::controllerType.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_IN_USE`: A storage controller with given name exists already.
- `E_INVALIDARG`: Invalid `controllerType`.

5.43.3 attachDevice

```
void IMachine::attachDevice(
            [in] wstring name,
            [in] long controllerPort,
            [in] long device,
            [in] DeviceType type,
            [in] IMedium medium)
```

name Name of the storage controller to attach the device to.

controllerPort Port to attach the device to. For an IDE controller, 0 specifies the primary controller and 1 specifies the secondary controller. For a SCSI controller, this must range from 0 to 15; for a SATA controller, from 0 to 29; for an SAS controller, from 0 to 7.

device Device slot in the given port to attach the device to. This is only relevant for IDE controllers, for which 0 specifies the master device and 1 specifies the slave device. For all other controller types, this must be 0.

type Device type of the attached device. For media opened by IVirtualBox::openMedium(), this must match the device type specified there.

medium Medium to mount or NULL for an empty drive.

Attaches a device and optionally mounts a medium to the given storage controller (IStorageController, identified by `name`), at the indicated port and device.

This method is intended for managing storage devices in general while a machine is powered off. It can be used to attach and detach fixed and removable media. The following kind of media can be attached to a machine:

- For fixed and removable media, you can pass in a medium that was previously opened using IVirtualBox::openMedium().
- Only for storage devices supporting removable media (such as DVDs and floppies), you can also specify a null pointer to indicate an empty drive or one of the medium objects listed in the IHost::DVDDrives[] and IHost::floppyDrives[] arrays to indicate a host drive. For removable devices, you can also use mountMedium() to change the media while the machine is running.

In a VM's default configuration of virtual machines, the secondary master of the IDE controller is used for a CD/DVD drive.

After calling this returns successfully, a new instance of IMediumAttachment will appear in the machine's list of medium attachments (see mediumAttachments[]).

See IMedium and IMediumAttachment for more information about attaching media.

The specified device slot must not have a device attached to it, or this method will fail.

> **Note:** You cannot attach a device to a newly created machine until this machine's settings are saved to disk using saveSettings().

> **Note:** If the medium is being attached indirectly, a new differencing medium will implicitly be created for it and attached instead. If the changes made to the machine settings (including this indirect attachment) are later cancelled using discardSettings(), this implicitly created differencing medium will implicitly be deleted.

If this method fails, the following error codes may be reported:

- `E_INVALIDARG:` SATA device, SATA port, IDE port or IDE slot out of range, or file or UUID not found.
- `VBOX_E_INVALID_OBJECT_STATE:` Machine must be registered before media can be attached.
- `VBOX_E_INVALID_VM_STATE:` Invalid machine state.
- `VBOX_E_OBJECT_IN_USE:` A medium is already attached to this or another virtual machine.

5.43.4 attachHostPciDevice

```
void IMachine::attachHostPciDevice(
           [in] long hostAddress,
           [in] long desiredGuestAddress,
           [in] IEventContext eventContext,
           [in] boolean tryToUnbind)
```

hostAddress Address of the host PCI device.

desiredGuestAddress Desired position of this device on guest PCI bus.

eventContext Context passed into IHostPciDevicePlugEvent event.

tryToUnbind If VMM shall try to unbind existing drivers from the device before attaching it to the guest.

Attaches host PCI device with the given (host) PCI address to the PCI bus of the virtual machine. Please note, that this operation is two phase, as real attachment will happen when VM will start, and most information will be delivered as IHostPciDevicePlugEvent on IVirtualBox event source.

Note: Not yet implemented.

See also: IHostPciDevicePlugEvent
If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Virtual machine state is not stopped (PCI hotplug not yet implemented).
- `VBOX_E_PDM_ERROR:` Virtual machine does not have a PCI controller allowing attachment of physical devices.
- `VBOX_E_NOT_SUPPORTED:` Hardware or host OS doesn't allow PCI device passthrought.

5.43.5 canShowConsoleWindow

```
boolean IMachine::canShowConsoleWindow()
```

Returns `true` if the VM console process can activate the console window and bring it to foreground on the desktop of the host PC.

Note: This method will fail if a session for this machine is not currently open.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Machine session is not open.

5.43.6 createSharedFolder

```
void IMachine::createSharedFolder(
           [in] wstring name,
           [in] wstring hostPath,
           [in] boolean writable,
           [in] boolean automount)
```

name Unique logical name of the shared folder.

hostPath Full path to the shared folder in the host file system.

writable Whether the share is writable or readonly.

automount Whether the share gets automatically mounted by the guest or not.

Creates a new permanent shared folder by associating the given logical name with the given host path, adds it to the collection of shared folders and starts sharing it. Refer to the description of ISharedFolder to read more about logical names.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_IN_USE`: Shared folder already exists.
- `VBOX_E_FILE_ERROR`: Shared folder `hostPath` not accessible.

5.43.7 delete

```
IProgress IMachine::delete(
           [in] IMedium aMedia[])
```

aMedia List of media to be closed and whose storage files will be deleted.

Deletes the files associated with this machine from disk. If medium objects are passed in with the `aMedia` argument, they are closed and, if closing was successful, their storage files are deleted as well. For convenience, this array of media files can be the same as the one returned from a previous unregister() call.

This method must only be called on machines which are either write-locked (i.e. on instances returned by ISession::machine) or on unregistered machines (i.e. not yet registered machines created by IVirtualBox::createMachine() or opened by IVirtualBox::openMachine(), or after having called unregister()).

The following files will be deleted by this method:

- If unregister() had been previously called with a `cleanupMode` argument other than "UnregisterOnly", this will delete all saved state files that the machine had in use; possibly one if the machine was in "Saved" state and one for each online snapshot that the machine had.
- On each medium object passed in the `aMedia` array, this will call IMedium::close(). If that succeeds, this will attempt to delete the medium's storage on disk. Since the close() call will fail if the medium is still in use, e.g. because it is still attached to a second machine; in that case the storage will not be deleted.
- Finally, the machine's own XML file will be deleted.

Since deleting large disk image files can be a time-consuming I/O operation, this method operates asynchronously and returns an IProgress object to allow the caller to monitor the progress. There will be one sub-operation for each file that is being deleted (saved state or medium storage file).

Note: settingsModified will return `true` after this method successfully returns.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Machine is registered but not write-locked.
- `VBOX_E_IPRT_ERROR`: Could not delete the settings file.

5.43.8 detachDevice

```
void IMachine::detachDevice(
           [in] wstring name,
           [in] long controllerPort,
           [in] long device)
```

name Name of the storage controller to detach the medium from.

controllerPort Port number to detach the medium from.

device Device slot number to detach the medium from.

Detaches the device attached to a device slot of the specified bus.
Detaching the device from the virtual machine is deferred. This means that the medium remains associated with the machine when this method returns and gets actually de-associated only after a successful saveSettings() call. See IMedium for more detailed information about attaching media.

Note: You cannot detach a device from a running machine.

Note: Detaching differencing media implicitly created by attachDevice() for the indirect attachment using this method will **not** implicitly delete them. The IMedium::deleteStorage() operation should be explicitly performed by the caller after the medium is successfully detached and the settings are saved with saveSettings(), if it is the desired action.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Attempt to detach medium from a running virtual machine.
- `VBOX_E_OBJECT_NOT_FOUND`: No medium attached to given slot/bus.
- `VBOX_E_NOT_SUPPORTED`: Medium format does not support storage deletion.

5.43.9 detachHostPciDevice

```
void IMachine::detachHostPciDevice(
           [in] long hostAddress)
```

hostAddress Address of the host PCI device.

Detach host PCI device from the virtual machine. Also HostPciDevicePlugEvent on IVirtualBox event source will be delivered.

Note: Not yet implemented.

See also: IHostPciDevicePlugEvent
If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Virtual machine state is not stopped (PCI hotplug not yet implemented).
- `VBOX_E_OBJECT_NOT_FOUND`: This host device is not attached to this machine.
- `VBOX_E_PDM_ERROR`: Virtual machine does not have a PCI controller allowing attachment of physical devices.
- `VBOX_E_NOT_SUPPORTED`: Hardware or host OS doesn't allow PCI device passthrought.

5.43.10 discardSettings

```
void IMachine::discardSettings()
```

Discards any changes to the machine settings made since the session has been opened or since the last call to saveSettings() or discardSettings().

> **Note:** Calling this method is only valid on instances returned by ISession::machine and on new machines created by IVirtualBox::createMachine() or opened by IVirtualBox::openMachine() but not yet registered, or on unregistered machines after calling unregister().

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Virtual machine is not mutable.

5.43.11 enumerateGuestProperties

```
void IMachine::enumerateGuestProperties(
            [in] wstring patterns,
            [out] wstring name[],
            [out] wstring value[],
            [out] long long timestamp[],
            [out] wstring flags[])
```

patterns The patterns to match the properties against, separated by '|' characters. If this is empty or `null`, all properties will match.

name The names of the properties returned.

value The values of the properties returned. The array entries match the corresponding entries in the `name` array.

timestamp The time stamps of the properties returned. The array entries match the corresponding entries in the `name` array.

flags The flags of the properties returned. The array entries match the corresponding entries in the `name` array.

Return a list of the guest properties matching a set of patterns along with their values, time stamps and flags.

5.43.12 export

```
IVirtualSystemDescription IMachine::export(
            [in] IAppliance aAppliance,
            [in] wstring location)
```

aAppliance Appliance to export this machine to.

location The target location.

Exports the machine to an OVF appliance. See IAppliance for the steps required to export VirtualBox machines to OVF.

5.43.13 findSnapshot

```
ISnapshot IMachine::findSnapshot(
            [in] wstring nameOrId)
```

nameOrId What to search for. Name or UUID of the snapshot to find

Returns a snapshot of this machine with the given name or UUID.

Returns a snapshot of this machine with the given UUID. A `null` argument can be used to obtain the first snapshot taken on this machine. To traverse the whole tree of snapshots starting from the root, inspect the root snapshot's ISnapshot::children[] attribute and recurse over those children.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: Virtual machine has no snapshots or snapshot not found.

5.43.14 getBootOrder

```
DeviceType IMachine::getBootOrder(
            [in] unsigned long position)
```

position Position in the boot order (1 to the total number of devices the machine can boot from, as returned by ISystemProperties::maxBootPosition).

Returns the device type that occupies the specified position in the boot order.

@todo [remove?] If the machine can have more than one device of the returned type (such as hard disks), then a separate method should be used to retrieve the individual device that occupies the given position.

If here are no devices at the given position, then Null is returned.

@todo getHardDiskBootOrder(), getNetworkBootOrder()

If this method fails, the following error codes may be reported:

- `E_INVALIDARG`: Boot `position` out of range.

5.43.15 getCPUIDLeaf

```
void IMachine::getCPUIDLeaf(
            [in] unsigned long id,
            [out] unsigned long valEax,
            [out] unsigned long valEbx,
            [out] unsigned long valEcx,
            [out] unsigned long valEdx)
```

id CPUID leaf index.

valEax CPUID leaf value for register eax.

valEbx CPUID leaf value for register ebx.

valEcx CPUID leaf value for register ecx.

valEdx CPUID leaf value for register edx.

Returns the virtual CPU cpuid information for the specified leaf.
Currently supported index values for cpuid: Standard CPUID leafs: 0 - 0xA Extended CPUID leafs: 0x80000000 - 0x8000000A
See the Intel and AMD programmer's manuals for detailed information about the cpuid instruction and its leafs.
If this method fails, the following error codes may be reported:

- `E_INVALIDARG:` Invalid id.

5.43.16 getCPUProperty

```
boolean IMachine::getCPUProperty(
            [in] CPUPropertyType property)
```

property Property type to query.

Returns the virtual CPU boolean value of the specified property.
If this method fails, the following error codes may be reported:

- `E_INVALIDARG:` Invalid property.

5.43.17 getCPUStatus

```
boolean IMachine::getCPUStatus(
            [in] unsigned long cpu)
```

cpu The CPU id to check for.

Returns the current status of the given CPU.

5.43.18 getExtraData

```
wstring IMachine::getExtraData(
            [in] wstring key)
```

key Name of the data key to get.

Returns associated machine-specific extra data.
If the requested data `key` does not exist, this function will succeed and return an empty string in the `value` argument.
If this method fails, the following error codes may be reported:

- `VBOX_E_FILE_ERROR:` Settings file not accessible.
- `VBOX_E_XML_ERROR:` Could not parse the settings file.

5.43.19 getExtraDataKeys

```
wstring[] IMachine::getExtraDataKeys()
```

Returns an array representing the machine-specific extra data keys which currently have values defined.

5.43.20 getGuestProperty

```
void IMachine::getGuestProperty(
            [in] wstring name,
            [out] wstring value,
            [out] long long timestamp,
            [out] wstring flags)
```

name The name of the property to read.

value The value of the property. If the property does not exist then this will be empty.

timestamp The time at which the property was last modified, as seen by the server process.

flags Additional property parameters, passed as a comma-separated list of "name=value" type entries.

Reads an entry from the machine's guest property store.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Machine session is not open.

5.43.21 getGuestPropertyTimestamp

```
long long IMachine::getGuestPropertyTimestamp(
            [in] wstring property)
```

property The name of the property to read.

Reads a property timestamp from the machine's guest property store.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Machine session is not open.

5.43.22 getGuestPropertyValue

```
wstring IMachine::getGuestPropertyValue(
            [in] wstring property)
```

property The name of the property to read.

Reads a value from the machine's guest property store.
If this method fails, the following error codes may be reported:

- VBOX_E_INVALID_VM_STATE: Machine session is not open.

5.43.23 getHWVirtExProperty

```
boolean IMachine::getHWVirtExProperty(
            [in] HWVirtExPropertyType property)
```

property Property type to query.

Returns the value of the specified hardware virtualization boolean property.
If this method fails, the following error codes may be reported:

- E_INVALIDARG: Invalid property.

5.43.24 getMedium

```
IMedium IMachine::getMedium(
            [in] wstring name,
            [in] long controllerPort,
            [in] long device)
```

name Name of the storage controller the medium is attached to.

controllerPort Port to query.

device Device slot in the given port to query.

Returns the virtual medium attached to a device slot of the specified bus.

Note that if the medium was indirectly attached by mountMedium() to the given device slot then this method will return not the same object as passed to the mountMedium() call. See IMedium for more detailed information about mounting a medium.

If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: No medium attached to given slot/bus.

5.43.25 getMediumAttachment

```
IMediumAttachment IMachine::getMediumAttachment(
            [in] wstring name,
            [in] long controllerPort,
            [in] long device)
```

name

controllerPort

device

Returns a medium attachment which corresponds to the controller with the given name, on the given port and device slot.

If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: No attachment exists for the given controller/port/device combination.

5.43.26 getMediumAttachmentsOfController

```
IMediumAttachment[] IMachine::getMediumAttachmentsOfController(
            [in] wstring name)
```

name

Returns an array of medium attachments which are attached to the the controller with the given name.

If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: A storage controller with given name doesn't exist.

5.43.27 getNetworkAdapter

```
INetworkAdapter IMachine::getNetworkAdapter(
            [in] unsigned long slot)
```

slot

Returns the network adapter associated with the given slot. Slots are numbered sequentially, starting with zero. The total number of adapters per machine is defined by the ISystemProperties::networkAdapterCount property, so the maximum slot number is one less than that property's value.

If this method fails, the following error codes may be reported:

- E_INVALIDARG: Invalid slot number.

5.43.28 getParallelPort

```
IParallelPort IMachine::getParallelPort(
            [in] unsigned long slot)
```

slot

Returns the parallel port associated with the given slot. Slots are numbered sequentially, starting with zero. The total number of parallel ports per machine is defined by the ISystemProperties::parallelPortCount property, so the maximum slot number is one less than that property's value.

If this method fails, the following error codes may be reported:

- E_INVALIDARG: Invalid slot number.

5.43.29 getSerialPort

```
ISerialPort IMachine::getSerialPort(
            [in] unsigned long slot)
```

slot

Returns the serial port associated with the given slot. Slots are numbered sequentially, starting with zero. The total number of serial ports per machine is defined by the ISystemProperties::serialPortCount property, so the maximum slot number is one less than that property's value.

If this method fails, the following error codes may be reported:

- E_INVALIDARG: Invalid slot number.

5.43.30 getStorageControllerByInstance

```
IStorageController IMachine::getStorageControllerByInstance(
            [in] unsigned long instance)
```

instance

Returns a storage controller with the given instance number.

If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: A storage controller with given instance number doesn't exist.

5.43.31 getStorageControllerByName

```
IStorageController IMachine::getStorageControllerByName(
            [in] wstring name)
```

name

Returns a storage controller with the given name.
If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: A storage controller with given name doesn't exist.

5.43.32 hotPlugCPU

```
void IMachine::hotPlugCPU(
            [in] unsigned long cpu)
```

cpu The CPU id to insert.

Plugs a CPU into the machine.

5.43.33 hotUnplugCPU

```
void IMachine::hotUnplugCPU(
            [in] unsigned long cpu)
```

cpu The CPU id to remove.

Removes a CPU from the machine.

5.43.34 launchVMProcess

```
IProgress IMachine::launchVMProcess(
            [in] ISession session,
            [in] wstring type,
            [in] wstring environment)
```

session Client session object to which the VM process will be connected (this must be in "Unlocked" state).

type Front-end to use for the new VM process. The following are currently supported:

- "gui": VirtualBox Qt GUI front-end
- "headless": VBoxHeadless (VRDE Server) front-end
- "sdl": VirtualBox SDL front-end
- "emergencystop": reserved value, used for aborting the currently running VM or session owner. In this case the session parameter may be NULL (if it is non-null it isn't used in any way), and the progress return value will be always NULL. The operation completes immediately.

environment Environment to pass to the VM process.

Spawns a new process that will execute the virtual machine and obtains a shared lock on the machine for the calling session.
If launching the VM succeeds, the new VM process will create its own session and write-lock the machine for it, preventing conflicting changes from other processes. If the machine is already locked (because it is already running or because another session has a write lock), launching the

VM process will therefore fail. Reversely, future attempts to obtain a write lock will also fail while the machine is running.

The caller's session object remains separate from the session opened by the new VM process. It receives its own IConsole object which can be used to control machine execution, but it cannot be used to change all VM settings which would be available after a lockMachine() call.

The caller must eventually release the session's shared lock by calling ISession::unlockMachine() on the local session object once this call has returned. However, the session's state (see ISession::state) will not return to "Unlocked" until the remote session has also unlocked the machine (i.e. the machine has stopped running).

Launching a VM process can take some time (a new VM is started in a new process, for which memory and other resources need to be set up). Because of this, an IProgress object is returned to allow the caller to wait for this asynchronous operation to be completed. Until then, the caller's session object remains in the "Unlocked" state, and its ISession::machine and ISession::console attributes cannot be accessed. It is recommended to use IProgress::waitForCompletion() or similar calls to wait for completion. Completion is signalled when the VM is powered on. If launching the VM fails, error messages can be queried via the progress object, if available.

The progress object will have at least 2 sub-operations. The first operation covers the period up to the new VM process calls powerUp. The subsequent operations mirror the IConsole::powerUp() progress object. Because IConsole::powerUp() may require some extra sub-operations, the IProgress::operationCount may change at the completion of operation.

For details on the teleportation progress operation, see IConsole::powerUp().

The `environment` argument is a string containing definitions of environment variables in the following format: @code NAME[=VALUE]\n NAME[=VALUE]\n ... @endcode where `\\n` is the new line character. These environment variables will be appended to the environment of the VirtualBox server process. If an environment variable exists both in the server process and in this list, the value from this list takes precedence over the server's variable. If the value of the environment variable is omitted, this variable will be removed from the resulting environment. If the environment string is `null` or empty, the server environment is inherited by the started process as is.

If this method fails, the following error codes may be reported:

- `E_UNEXPECTED:` Virtual machine not registered.
- `E_INVALIDARG:` Invalid session type `type`.
- `VBOX_E_OBJECT_NOT_FOUND:` No machine matching `machineId` found.
- `VBOX_E_INVALID_OBJECT_STATE:` Session already open or being opened.
- `VBOX_E_IPRT_ERROR:` Launching process for machine failed.
- `VBOX_E_VM_ERROR:` Failed to assign machine to session.

5.43.35 lockMachine

```
void IMachine::lockMachine(
           [in] ISession session,
           [in] LockType lockType)
```

session Session object for which the machine will be locked.

lockType If set to `Write`, then attempt to acquire an exclusive write lock or fail. If set to `Shared`, then either acquire an exclusive write lock or establish a link to an existing session.

Locks the machine for the given session to enable the caller to make changes to the machine or start the VM or control VM execution.

There are two ways to lock a machine for such uses:

- If you want to make changes to the machine settings, you must obtain an exclusive write lock on the machine by setting `lockType` to `Write`.

  This will only succeed if no other process has locked the machine to prevent conflicting changes. Only after an exclusive write lock has been obtained using this method, one can change all VM settings or execute the VM in the process space of the session object. (Note that the latter is only of interest if you actually want to write a new front-end for virtual machines; but this API gets called internally by the existing front-ends such as VBoxHeadless and the VirtualBox GUI to acquire a write lock on the machine that they are running.)

  On success, write-locking the machine for a session creates a second copy of the IMachine object. It is this second object upon which changes can be made; in VirtualBox terminology, the second copy is "mutable". It is only this second, mutable machine object upon which you can call methods that change the machine state. After having called this method, you can obtain this second, mutable machine object using the ISession::machine attribute.

- If you only want to check the machine state or control machine execution without actually changing machine settings (e.g. to get access to VM statistics or take a snapshot or save the machine state), then set the `lockType` argument to `Shared`.

  If no other session has obtained a lock, you will obtain an exclusive write lock as described above. However, if another session has already obtained such a lock, then a link to that existing session will be established which allows you to control that existing session.

  To find out which type of lock was obtained, you can inspect ISession::type, which will have been set to either `WriteLock` or `Shared`.

In either case, you can get access to the IConsole object which controls VM execution.

Also in all of the above cases, one must always call ISession::unlockMachine() to release the lock on the machine, or the machine's state will eventually be set to "Aborted".

To change settings on a machine, the following sequence is typically performed:

1. Call this method to obtain an exclusive write lock for the current session.
2. Obtain a mutable IMachine object from ISession::machine.
3. Change the settings of the machine by invoking IMachine methods.
4. Call saveSettings().
5. Release the write lock by calling ISession::unlockMachine().

If this method fails, the following error codes may be reported:

- `E_UNEXPECTED:` Virtual machine not registered.
- `E_ACCESSDENIED:` Process not started by OpenRemoteSession.
- `VBOX_E_INVALID_OBJECT_STATE:` Session already open or being opened.
- `VBOX_E_VM_ERROR:` Failed to assign machine to session.

5.43.36 mountMedium

```
void IMachine::mountMedium(
            [in] wstring name,
            [in] long controllerPort,
            [in] long device,
            [in] IMedium medium,
            [in] boolean force)
```

name Name of the storage controller to attach the medium to.

controllerPort Port to attach the medium to.

device Device slot in the given port to attach the medium to.

medium Medium to mount or NULL for an empty drive.

force Allows to force unmount/mount of a medium which is locked by the device slot in the given port to attach the medium to.

Mounts a medium (IMedium, identified by the given UUID `id`) to the given storage controller (IStorageController, identified by `name`), at the indicated port and device. The device must already exist; see attachDevice() for how to attach a new device.

This method is intended only for managing removable media, where the device is fixed but media is changeable at runtime (such as DVDs and floppies). It cannot be used for fixed media such as hard disks.

The `controllerPort` and `device` parameters specify the device slot and have have the same meaning as with attachDevice().

The specified device slot can have a medium mounted, which will be unmounted first. Specifying a zero UUID (or an empty string) for `medium` does just an unmount.

See IMedium for more detailed information about attaching media.

If this method fails, the following error codes may be reported:

- `E_INVALIDARG:` SATA device, SATA port, IDE port or IDE slot out of range.
- `VBOX_E_INVALID_OBJECT_STATE:` Attempt to attach medium to an unregistered virtual machine.
- `VBOX_E_INVALID_VM_STATE:` Invalid machine state.
- `VBOX_E_OBJECT_IN_USE:` Medium already attached to this or another virtual machine.

5.43.37 passthroughDevice

```
void IMachine::passthroughDevice(
            [in] wstring name,
            [in] long controllerPort,
            [in] long device,
            [in] boolean passthrough)
```

name Name of the storage controller.

controllerPort Storage controller port.

device Device slot in the given port.

passthrough New value for the passthrough setting.

Sets the passthrough mode of an existing DVD device. Changing the setting while the VM is running is forbidden. The setting is only used if at VM start the device is configured as a host DVD drive, in all other cases it is ignored. The device must already exist; see attachDevice() for how to attach a new device.

The `controllerPort` and `device` parameters specify the device slot and have have the same meaning as with attachDevice().

If this method fails, the following error codes may be reported:

- `E_INVALIDARG:` SATA device, SATA port, IDE port or IDE slot out of range.
- `VBOX_E_INVALID_OBJECT_STATE:` Attempt to modify an unregistered virtual machine.
- `VBOX_E_INVALID_VM_STATE:` Invalid machine state.

5.43.38 queryLogFilename

```
wstring IMachine::queryLogFilename(
            [in] unsigned long idx)
```

idx Which log file name to query. 0=current log file.

Queries for the VM log file name of an given index. Returns an empty string if a log file with that index doesn't exists.

5.43.39 querySavedGuestSize

```
void IMachine::querySavedGuestSize(
            [in] unsigned long screenId,
            [out] unsigned long width,
            [out] unsigned long height)
```

screenId Saved guest screen to query info from.

width Guest width at the time of the saved state was taken.

height Guest height at the time of the saved state was taken.

Returns the guest dimensions from the saved state.

5.43.40 querySavedScreenshotPNGSize

```
void IMachine::querySavedScreenshotPNGSize(
            [in] unsigned long screenId,
            [out] unsigned long size,
            [out] unsigned long width,
            [out] unsigned long height)
```

screenId Saved guest screen to query info from.

size Size of buffer required to store the PNG binary data.

width Image width.

height Image height.

Returns size in bytes and dimensions of a saved PNG image of screenshot from saved state.

5.43.41 querySavedThumbnailSize

```
void IMachine::querySavedThumbnailSize(
            [in] unsigned long screenId,
            [out] unsigned long size,
            [out] unsigned long width,
            [out] unsigned long height)
```

screenId Saved guest screen to query info from.

size Size of buffer required to store the bitmap.

width Bitmap width.

height Bitmap height.

Returns size in bytes and dimensions in pixels of a saved thumbnail bitmap from saved state.

5.43.42 readLog

```
octet[] IMachine::readLog(
            [in] unsigned long idx,
            [in] long long offset,
            [in] long long size)
```

idx Which log file to read. 0=current log file.

offset Offset in the log file.

size Chunk size to read in the log file.

Reads the VM log file. The chunk size is limited, so even if you ask for a big piece there might be less data returned.

5.43.43 readSavedScreenshotPNGToArray

```
octet[] IMachine::readSavedScreenshotPNGToArray(
            [in] unsigned long screenId,
            [out] unsigned long width,
            [out] unsigned long height)
```

screenId Saved guest screen to read from.

width Image width.

height Image height.

Screenshot in PNG format is retrieved to an array of bytes.

5.43.44 readSavedThumbnailPNGToArray

```
octet[] IMachine::readSavedThumbnailPNGToArray(
            [in] unsigned long screenId,
            [out] unsigned long width,
            [out] unsigned long height)
```

screenId Saved guest screen to read from.

width Image width.

height Image height.

Thumbnail in PNG format is retrieved to an array of bytes.

5.43.45 readSavedThumbnailToArray

```
octet[] IMachine::readSavedThumbnailToArray(
            [in] unsigned long screenId,
            [in] boolean BGR,
            [out] unsigned long width,
            [out] unsigned long height)
```

screenId Saved guest screen to read from.

BGR How to order bytes in the pixel. A pixel consists of 4 bytes. If this parameter is true, then bytes order is: B, G, R, 0xFF. If this parameter is false, then bytes order is: R, G, B, 0xFF.

width Bitmap width.

height Bitmap height.

Thumbnail is retrieved to an array of bytes in uncompressed 32-bit BGRA or RGBA format.

5.43.46 removeAllCPUIDLeaves

```
void IMachine::removeAllCPUIDLeaves()
```

Removes all the virtual CPU cpuid leaves

5.43.47 removeCPUIDLeaf

```
void IMachine::removeCPUIDLeaf(
            [in] unsigned long id)
```

id CPUID leaf index.

Removes the virtual CPU cpuid leaf for the specified index
If this method fails, the following error codes may be reported:

- `E_INVALIDARG`: Invalid id.

5.43.48 removeSharedFolder

```
void IMachine::removeSharedFolder(
            [in] wstring name)
```

name Logical name of the shared folder to remove.

Removes the permanent shared folder with the given name previously created by createSharedFolder() from the collection of shared folders and stops sharing it.
If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Virtual machine is not mutable.
- `VBOX_E_OBJECT_NOT_FOUND`: Shared folder `name` does not exist.

5.43.49 removeStorageController

```
void IMachine::removeStorageController(
            [in] wstring name)
```

name

Removes a storage controller from the machine.
If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: A storage controller with given name doesn't exist.

5.43.50 saveSettings

```
void IMachine::saveSettings()
```

Saves any changes to machine settings made since the session has been opened or a new machine has been created, or since the last call to saveSettings() or discardSettings(). For registered machines, new settings become visible to all other VirtualBox clients after successful invocation of this method.

> **Note:** The method sends IMachineDataChangedEvent notification event after the configuration has been successfully saved (only for registered machines).

Note: Calling this method is only valid on instances returned by ISession::machine and on new machines created by IVirtualBox::createMachine() but not yet registered, or on unregistered machines after calling unregister().

If this method fails, the following error codes may be reported:

- VBOX_E_FILE_ERROR: Settings file not accessible.
- VBOX_E_XML_ERROR: Could not parse the settings file.
- E_ACCESSDENIED: Modification request refused.

5.43.51 setBandwidthGroupForDevice

```
void IMachine::setBandwidthGroupForDevice(
            [in] wstring name,
            [in] long controllerPort,
            [in] long device,
            [in] IBandwidthGroup bandwidthGroup)
```

name Name of the storage controller.

controllerPort Storage controller port.

device Device slot in the given port.

bandwidthGroup New value for the bandwidth group or NULL for no group.

Sets the bandwidth group of an existing storage device. The device must already exist; see attachDevice() for how to attach a new device.

The controllerPort and device parameters specify the device slot and have have the same meaning as with attachDevice().

If this method fails, the following error codes may be reported:

- E_INVALIDARG: SATA device, SATA port, IDE port or IDE slot out of range.
- VBOX_E_INVALID_OBJECT_STATE: Attempt to modify an unregistered virtual machine.
- VBOX_E_INVALID_VM_STATE: Invalid machine state.

5.43.52 setBootOrder

```
void IMachine::setBootOrder(
            [in] unsigned long position,
            [in] DeviceType device)
```

position Position in the boot order (1 to the total number of devices the machine can boot from, as returned by ISystemProperties::maxBootPosition).

device The type of the device used to boot at the given position.

Puts the given device to the specified position in the boot order.

To indicate that no device is associated with the given position, Null should be used.

@todo setHardDiskBootOrder(), setNetworkBootOrder()

If this method fails, the following error codes may be reported:

- E_INVALIDARG: Boot position out of range.
- E_NOTIMPL: Booting from USB device currently not supported.

5.43.53 setCPUIDLeaf

```
void IMachine::setCPUIDLeaf(
            [in] unsigned long id,
            [in] unsigned long valEax,
            [in] unsigned long valEbx,
            [in] unsigned long valEcx,
            [in] unsigned long valEdx)
```

id CPUID leaf index.

valEax CPUID leaf value for register eax.

valEbx CPUID leaf value for register ebx.

valEcx CPUID leaf value for register ecx.

valEdx CPUID leaf value for register edx.

Sets the virtual CPU cpuid information for the specified leaf. Note that these values are not passed unmodified. VirtualBox clears features that it doesn't support.

Currently supported index values for cpuid: Standard CPUID leafs: 0 - 0xA Extended CPUID leafs: 0x80000000 - 0x8000000A

See the Intel and AMD programmer's manuals for detailed information about the cpuid instruction and its leafs.

Do not use this method unless you know exactly what you're doing. Misuse can lead to random crashes inside VMs.

If this method fails, the following error codes may be reported:

- `E_INVALIDARG:` Invalid id.

5.43.54 setCPUProperty

```
void IMachine::setCPUProperty(
            [in] CPUPropertyType property,
            [in] boolean value)
```

property Property type to query.

value Property value.

Sets the virtual CPU boolean value of the specified property.

If this method fails, the following error codes may be reported:

- `E_INVALIDARG:` Invalid property.

5.43.55 setExtraData

```
void IMachine::setExtraData(
            [in] wstring key,
            [in] wstring value)
```

key Name of the data key to set.

value Value to assign to the key.

Sets associated machine-specific extra data.

If you pass `null` or an empty string as a key `value`, the given `key` will be deleted.

Note: Before performing the actual data change, this method will ask all registered listeners using the IExtraDataCanChangeEvent notification for a permission. If one of the listeners refuses the new value, the change will not be performed.

Note: On success, the IExtraDataChangedEvent notification is called to inform all registered listeners about a successful data change.

Note: This method can be called outside the machine session and therefore it's a caller's responsibility to handle possible race conditions when several clients change the same key at the same time.

If this method fails, the following error codes may be reported:

- `VBOX_E_FILE_ERROR:` Settings file not accessible.
- `VBOX_E_XML_ERROR:` Could not parse the settings file.

5.43.56 setGuestProperty

```
void IMachine::setGuestProperty(
            [in] wstring property,
            [in] wstring value,
            [in] wstring flags)
```

property The name of the property to set, change or delete.

value The new value of the property to set, change or delete. If the property does not yet exist and value is non-empty, it will be created. If the value is `null` or empty, the property will be deleted if it exists.

flags Additional property parameters, passed as a comma-separated list of "name=value" type entries.

Sets, changes or deletes an entry in the machine's guest property store.
If this method fails, the following error codes may be reported:

- `E_ACCESSDENIED:` Property cannot be changed.
- `E_INVALIDARG:` Invalid `flags`.
- `VBOX_E_INVALID_VM_STATE:` Virtual machine is not mutable or session not open.
- `VBOX_E_INVALID_OBJECT_STATE:` Cannot set transient property when machine not running.

5.43.57 setGuestPropertyValue

```
void IMachine::setGuestPropertyValue(
            [in] wstring property,
            [in] wstring value)
```

property The name of the property to set, change or delete.

value The new value of the property to set, change or delete. If the property does not yet exist and value is non-empty, it will be created. If the value is `null` or empty, the property will be deleted if it exists.

Sets, changes or deletes a value in the machine's guest property store. The flags field will be left unchanged or created empty for a new property.
If this method fails, the following error codes may be reported:

- `E_ACCESSDENIED:` Property cannot be changed.
- `VBOX_E_INVALID_VM_STATE:` Virtual machine is not mutable or session not open.
- `VBOX_E_INVALID_OBJECT_STATE:` Cannot set transient property when machine not running.

5.43.58 setHWVirtExProperty

```
void IMachine::setHWVirtExProperty(
            [in] HWVirtExPropertyType property,
            [in] boolean value)
```

property Property type to set.

value New property value.

Sets a new value for the specified hardware virtualization boolean property.
If this method fails, the following error codes may be reported:

- `E_INVALIDARG:` Invalid property.

5.43.59 setStorageControllerBootable

```
void IMachine::setStorageControllerBootable(
            [in] wstring name,
            [in] boolean bootable)
```

name

bootable

Sets the bootable flag of the storage controller with the given name.
If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND:` A storage controller with given name doesn't exist.
- `VBOX_E_OBJECT_IN_USE:` Another storage controller is marked as bootable already.

5.43.60 showConsoleWindow

```
long long IMachine::showConsoleWindow()
```

Activates the console window and brings it to foreground on the desktop of the host PC. Many modern window managers on many platforms implement some sort of focus stealing prevention logic, so that it may be impossible to activate a window without the help of the currently active application. In this case, this method will return a non-zero identifier that represents the top-level window of the VM console process. The caller, if it represents a currently active process, is responsible to use this identifier (in a platform-dependent manner) to perform actual window activation.

Note: This method will fail if a session for this machine is not currently open.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE`: Machine session is not open.

5.43.61 unregister

```
IMedium[] IMachine::unregister(
          [in] CleanupMode cleanupMode)
```

cleanupMode How to clean up after the machine has been unregistered.

Unregisters a machine previously registered with IVirtualBox::registerMachine() and optionally do additional cleanup before the machine is unregistered.

This method does not delete any files. It only changes the machine configuration and the list of registered machines in the VirtualBox object. To delete the files which belonged to the machine, including the XML file of the machine itself, call delete(), optionally with the array of IMedium objects which was returned from this method.

How thoroughly this method cleans up the machine configuration before unregistering the machine depends on the `cleanupMode` argument.

- With "UnregisterOnly", the machine will only be unregistered, but no additional cleanup will be performed. The call will fail if the machine is in "Saved" state or has any snapshots or any media attached (see IMediumAttachment). It is the responsibility of the caller to delete all such configuration in this mode. In this mode, the API behaves like the former `IVirtualBox`::unregisterMachine() API which it replaces.
- With "DetachAllReturnNone", the call will succeed even if the machine is in "Saved" state or if it has snapshots or media attached. All media attached to the current machine state or in snapshots will be detached. No medium objects will be returned; all of the machine's media will remain open.
- With "DetachAllReturnHardDisksOnly", the call will behave like with "DetachAllReturnNone", except that all the hard disk medium objects which were detached from the machine will be returned as an array. This allows for quickly passing them to the delete() API for closing and deletion.
- With "Full", the call will behave like with "DetachAllReturnHardDisksOnly", except that all media will be returned in the array, including removable media like DVDs and floppies. This might be useful if the user wants to inspect in detail which media were attached to the machine. Be careful when passing the media array to delete() in that case because users will typically want to preserve ISO and RAW image files.

A typical implementation will use "DetachAllReturnHardDisksOnly" and then pass the resulting IMedium array to delete(). This way, the machine is completely deleted with all its saved states and hard disk images, but images for removable drives (such as ISO and RAW files) will remain on disk.

This API does not verify whether the media files returned in the array are still attached to other machines (i.e. shared between several machines). If such a shared image is passed to delete() however, closing the image will fail there and the image will be silently skipped.

This API may, however, move media from this machine's media registry to other media registries (see IMedium for details on media registries). For machines created with VirtualBox 4.0 or later, if media from this machine's media registry are also attached to another machine (shared attachments), each such medium will be moved to another machine's registry. This is because

without this machine's media registry, the other machine cannot find its media any more and would become inaccessible.

This API implicitly calls saveSettings() to save all current machine settings before unregistering it. It may also silently call saveSettings() on other machines if media are moved to other machines' media registries.

After successful method invocation, the IMachineRegisteredEvent event is fired.

The call will fail if the machine is currently locked (see ISession).

> **Note:** If the given machine is inaccessible (see accessible), it will be unregistered and fully uninitialized right afterwards. As a result, the returned machine object will be unusable and an attempt to call **any** method will return the "Object not ready" error.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_OBJECT_STATE`: Machine is currently locked for a session.

5.44 IMachineDataChangedEvent (IMachineEvent)

> **Note:** This interface extends IMachineEvent and therefore supports all its methods and attributes as well.

Any of the settings of the given machine has changed.

5.45 IMachineDebugger

> **Note:** This interface is not supported in the web service.

5.45.1 Attributes

5.45.1.1 singlestep (read/write)

`boolean IMachineDebugger::singlestep`

Switch for enabling singlestepping.

5.45.1.2 recompileUser (read/write)

`boolean IMachineDebugger::recompileUser`

Switch for forcing code recompilation for user mode code.

5.45.1.3 recompileSupervisor (read/write)

`boolean IMachineDebugger::recompileSupervisor`

Switch for forcing code recompilation for supervisor mode code.

5.45.1.4 PATMEnabled (read/write)

`boolean IMachineDebugger::PATMEnabled`

Switch for enabling and disabling the PATM component.

5.45.1.5 CSAMEnabled (read/write)

```
boolean IMachineDebugger::CSAMEnabled
```

Switch for enabling and disabling the CSAM component.

5.45.1.6 logEnabled (read/write)

```
boolean IMachineDebugger::logEnabled
```

Switch for enabling and disabling the debug logger.

5.45.1.7 logFlags (read-only)

```
wstring IMachineDebugger::logFlags
```

The debug logger flags.

5.45.1.8 logGroups (read-only)

```
wstring IMachineDebugger::logGroups
```

The debug logger's group settings.

5.45.1.9 logDestinations (read-only)

```
wstring IMachineDebugger::logDestinations
```

The debug logger's destination settings.

5.45.1.10 HWVirtExEnabled (read-only)

```
boolean IMachineDebugger::HWVirtExEnabled
```

Flag indicating whether the VM is currently making use of CPU hardware virtualization extensions.

5.45.1.11 HWVirtExNestedPagingEnabled (read-only)

```
boolean IMachineDebugger::HWVirtExNestedPagingEnabled
```

Flag indicating whether the VM is currently making use of the nested paging CPU hardware virtualization extension.

5.45.1.12 HWVirtExVPIDEnabled (read-only)

```
boolean IMachineDebugger::HWVirtExVPIDEnabled
```

Flag indicating whether the VM is currently making use of the VPID VT-x extension.

5.45.1.13 OSName (read-only)

```
wstring IMachineDebugger::OSName
```

Query the guest OS kernel name as detected by the DBGF.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.1.14 OSVersion (read-only)

```
wstring IMachineDebugger::OSVersion
```

Query the guest OS kernel version string as detected by the DBGF.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.1.15 PAEEnabled (read-only)

```
boolean IMachineDebugger::PAEEnabled
```

Flag indicating whether the VM is currently making use of the Physical Address Extension CPU feature.

5.45.1.16 virtualTimeRate (read/write)

```
unsigned long IMachineDebugger::virtualTimeRate
```

The rate at which the virtual time runs expressed as a percentage. The accepted range is 2% to 20000%.

5.45.1.17 VM (read-only)

```
long long IMachineDebugger::VM
```

Gets the VM handle. This is only for internal use while we carve the details of this interface.

5.45.2 detectOS

```
wstring IMachineDebugger::detectOS()
```

Tries to (re-)detect the guest OS kernel.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.3 dumpGuestCore

```
void IMachineDebugger::dumpGuestCore(
           [in] wstring filename,
           [in] wstring compression)
```

filename The name of the output file. The file must not exist.

compression Reserved for future compression method indicator.

Takes a core dump of the guest.
See include/VBox/dbgfcorefmt.h for details on the file format.

5.45.4 dumpGuestStack

```
wstring IMachineDebugger::dumpGuestStack(
           [in] unsigned long cpuId)
```

cpuId The identifier of the Virtual CPU.

Produce a simple stack dump using the current guest state.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.5 dumpHostProcessCore

```
void IMachineDebugger::dumpHostProcessCore(
            [in] wstring filename,
            [in] wstring compression)
```

filename The name of the output file. The file must not exist.

compression Reserved for future compression method indicator.

Takes a core dump of the VM process on the host.
This feature is not implemented in the 4.0.0 release but it may show up in a dot release.

5.45.6 dumpStats

```
void IMachineDebugger::dumpStats(
            [in] wstring pattern)
```

pattern The selection pattern. A bit similar to filename globbing.

Dumps VM statistics.

5.45.7 getRegister

```
wstring IMachineDebugger::getRegister(
            [in] unsigned long cpuId,
            [in] wstring name)
```

cpuId The identifier of the Virtual CPU.

name The register name, case is ignored.

Gets one register.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.8 getRegisters

```
void IMachineDebugger::getRegisters(
            [in] unsigned long cpuId,
            [out] wstring names[],
            [out] wstring values[])
```

cpuId The identifier of the Virtual CPU.

names Array containing the lowercase register names.

values Array paralell to the names holding the register values as if the register was returned by getRegister().

Gets all the registers for the given CPU.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.9 getStats

```
void IMachineDebugger::getStats(
            [in] wstring pattern,
            [in] boolean withDescriptions,
            [out] wstring stats)
```

pattern The selection pattern. A bit similar to filename globbing.

withDescriptions Whether to include the descriptions.

stats The XML document containing the statistics.

Get the VM statistics in a XMLish format.

5.45.10 info

```
wstring IMachineDebugger::info(
            [in] wstring name,
            [in] wstring args)
```

name The name of the info item.

args Arguments to the info dumper.

Interfaces with the info dumpers (DBGFInfo).
This feature is not implemented in the 4.0.0 release but it may show up in a dot release.

5.45.11 injectNMI

```
void IMachineDebugger::injectNMI()
```

Inject an NMI into a running VT-x/AMD-V VM.

5.45.12 modifyLogDestinations

```
void IMachineDebugger::modifyLogDestinations(
            [in] wstring settings)
```

settings The destination settings string. See iprt/log.h for details.

Modifies the debug logger destinations.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.13 modifyLogFlags

```
void IMachineDebugger::modifyLogFlags(
            [in] wstring settings)
```

settings The flags settings string. See iprt/log.h for details.

Modifies the debug logger flags.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.14 modifyLogGroups

```
void IMachineDebugger::modifyLogGroups(
            [in] wstring settings)
```

settings The group settings string. See iprt/log.h for details.

Modifies the group settings of the debug logger.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.15 readPhysicalMemory

```
octet[] IMachineDebugger::readPhysicalMemory(
            [in] long long address,
            [in] unsigned long size)
```

address The guest physical address.

size The number of bytes to read.

Reads guest physical memory, no side effects (MMIO++).
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.16 readVirtualMemory

```
octet[] IMachineDebugger::readVirtualMemory(
            [in] unsigned long cpuId,
            [in] long long address,
            [in] unsigned long size)
```

cpuId The identifier of the Virtual CPU.

address The guest virtual address.

size The number of bytes to read.

Reads guest virtual memory, no side effects (MMIO++).
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.17 resetStats

```
void IMachineDebugger::resetStats(
            [in] wstring pattern)
```

pattern The selection pattern. A bit similar to filename globbing.

Reset VM statistics.

5.45.18 setRegister

```
void IMachineDebugger::setRegister(
            [in] unsigned long cpuId,
            [in] wstring name,
            [in] wstring value)
```

cpuId The identifier of the Virtual CPU.

name The register name, case is ignored.

value The new register value. Hexadecimal, decimal and octal formattings are supported in addition to any special formattings returned by the getters.

Gets one register.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.19 setRegisters

```
void IMachineDebugger::setRegisters(
            [in] unsigned long cpuId,
            [in] wstring names[],
            [in] wstring values[])
```

cpuId The identifier of the Virtual CPU.

names Array containing the register names, case ignored.

values Array paralell to the names holding the register values. See setRegister() for formatting guidelines.

Sets zero or more registers atomically.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.20 writePhysicalMemory

```
void IMachineDebugger::writePhysicalMemory(
            [in] long long address,
            [in] unsigned long size,
            [in] octet bytes[])
```

address The guest physical address.

size The number of bytes to read.

bytes The bytes to write.

Writes guest physical memory, access handles (MMIO++) are ignored.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.45.21 writeVirtualMemory

```
void IMachineDebugger::writeVirtualMemory(
            [in] unsigned long cpuId,
            [in] long long address,
            [in] unsigned long size,
            [in] octet bytes[])
```

cpuId The identifier of the Virtual CPU.

address The guest virtual address.

size The number of bytes to read.

bytes The bytes to write.

Writes guest virtual memory, access handles (MMIO++) are ignored.
This feature is not implemented in the 4.0.0 release but may show up in a dot release.

5.46 IMachineEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Base abstract interface for all machine events.

5.46.1 Attributes

5.46.1.1 machineId (read-only)

```
uuid IMachineEvent::machineId
```

ID of the machine this event relates to.

5.47 IMachineRegisteredEvent (IMachineEvent)

> **Note:** This interface extends IMachineEvent and therefore supports all its methods and attributes as well.

The given machine was registered or unregistered within this VirtualBox installation.

5.47.1 Attributes

5.47.1.1 registered (read-only)

```
boolean IMachineRegisteredEvent::registered
```

If `true`, the machine was registered, otherwise it was unregistered.

5.48 IMachineStateChangedEvent (IMachineEvent)

> **Note:** This interface extends IMachineEvent and therefore supports all its methods and attributes as well.

Machine state change event.

5.48.1 Attributes

5.48.1.1 state (read-only)

```
MachineState IMachineStateChangedEvent::state
```

New execution state.

5.49 IManagedObjectRef

Note: This interface is supported in the web service only, not in COM/XPCOM.

Managed object reference.

Only within the webservice, a managed object reference (which is really an opaque number) allows a webservice client to address an object that lives in the address space of the webservice server.

Behind each managed object reference, there is a COM object that lives in the webservice server's address space. The COM object is not freed until the managed object reference is released, either by an explicit call to release() or by logging off from the webservice (IWebsessionManager::logoff()), which releases all objects created during the webservice session.

Whenever a method call of the VirtualBox API returns a COM object, the webservice representation of that method will instead return a managed object reference, which can then be used to invoke methods on that object.

5.49.1 getInterfaceName

```
wstring IManagedObjectRef::getInterfaceName()
```

Returns the name of the interface that this managed object represents, for example, "IMachine", as a string.

5.49.2 release

```
void IManagedObjectRef::release()
```

Releases this managed object reference and frees the resources that were allocated for it in the webservice server process. After calling this method, the identifier of the reference can no longer be used.

5.50 IMedium

The IMedium interface represents virtual storage for a machine's hard disks, CD/DVD or floppy drives. It will typically represent a disk image on the host, for example a VDI or VMDK file representing a virtual hard disk, or an ISO or RAW file representing virtual removable media, but can also point to a network location (e.g. for iSCSI targets).

Instances of IMedium are connected to virtual machines by way of medium attachments, which link the storage medium to a particular device slot of a storage controller of the virtual machine. In the VirtualBox API, virtual storage is therefore always represented by the following chain of object links:

- IMachine::storageControllers[] contains an array of storage controllers (IDE, SATA, SCSI, SAS or a floppy controller; these are instances of IStorageController).
- IMachine::mediumAttachments[] contains an array of medium attachments (instances of IMediumAttachment created by IMachine::attachDevice()), each containing a storage controller from the above array, a port/device specification, and an instance of IMedium representing the medium storage (image file).

  For removable media, the storage medium is optional; a medium attachment with no medium represents a CD/DVD or floppy drive with no medium inserted. By contrast, hard disk attachments will always have an IMedium object attached.

- Each IMedium in turn points to a storage unit (such as a file on the host computer or a network resource) that holds actual data. This location is represented by the location attribute.

Existing media are opened using IVirtualBox::openMedium(); new hard disk media can be created with the VirtualBox API using the IVirtualBox::createHardDisk() method. Differencing hard disks (see below) are usually implicitly created by VirtualBox as needed, but may also be created explicitly using createDiffStorage(). VirtualBox cannot create CD/DVD or floppy images (ISO and RAW files); these should be created with external tools and then opened from within VirtualBox.

Only for CD/DVDs and floppies, an IMedium instance can also represent a host drive. In that case the id attribute contains the UUID of one of the drives in IHost::DVDDrives[] or IHost::floppyDrives[].

Media registries

When a medium has been opened or created using one of the aforementioned APIs, it becomes "known" to VirtualBox. Known media can be attached to virtual machines and accessed through IVirtualBox::findMedium(). They also appear in the global IVirtualBox::hardDisks[], IVirtualBox::DVDImages[] and IVirtualBox::floppyImages[] arrays.

Prior to VirtualBox 4.0, opening a medium added it to a global media registry in the VirtualBox.xml file, which was shared between all machines and made transporting machines and their media from one host to another difficult.

Starting with VirtualBox 4.0, media are only added to a registry when they are *attached* to a machine using IMachine::attachDevice(). For backwards compatibility, which registry a medium is added to depends on which VirtualBox version created a machine:

- If the medium has first been attached to a machine which was created by VirtualBox 4.0 or later, it is added to that machine's media registry in the machine XML settings file. This way all information about a machine's media attachments is contained in a single file and can be transported easily.

- For older media attachments (i.e. if the medium was first attached to a machine which was created with a VirtualBox version before 4.0), media continue to be registered in the global VirtualBox settings file, for backwards compatibility.

See IVirtualBox::openMedium() for more information.

Media are removed from media registries by the close(), deleteStorage() and mergeTo() methods.

Accessibility checks

VirtualBox defers media accessibility checks until the refreshState() method is called explicitly on a medium. This is done to make the VirtualBox object ready for serving requests as fast as possible and let the end-user application decide if it needs to check media accessibility right away or not.

As a result, when VirtualBox starts up (e.g. the VirtualBox object gets created for the first time), all known media are in the "Inaccessible" state, but the value of the lastAccessError attribute is an empty string because no actual accessibility check has been made yet.

After calling refreshState(), a medium is considered *accessible* if its storage unit can be read. In that case, the state attribute has a value of "Created". If the storage unit cannot be read (for example, because it is located on a disconnected network resource, or was accidentally deleted outside VirtualBox), the medium is considered *inaccessible*, which is indicated by the "Inaccessible" state. The exact reason why the medium is inaccessible can be obtained by reading the lastAccessError attribute.

Medium types

There are five types of medium behavior which are stored in the type attribute (see MediumType) and which define the medium's behavior with attachments and snapshots.

All media can be also divided in two groups: *base* media and *differencing* media. A base medium contains all sectors of the medium data in its own storage and therefore can be used independently. In contrast, a differencing medium is a "delta" to some other medium and contains only those sectors which differ from that other medium, which is then called a *parent*. The differencing medium is said to be *linked to* that parent. The parent may be itself a differencing medium, thus forming a chain of linked media. The last element in that chain must always be a base medium. Note that several differencing media may be linked to the same parent medium.

Differencing media can be distinguished from base media by querying the parent attribute: base media do not have parents they would depend on, so the value of this attribute is always `null` for them. Using this attribute, it is possible to walk up the medium tree (from the child medium to its parent). It is also possible to walk down the tree using the children[] attribute.

Note that the type of all differencing media is "normal"; all other values are meaningless for them. Base media may be of any type.

Automatic composition of the file name part

Another extension to the location attribute is that there is a possibility to cause VirtualBox to compose a unique value for the file name part of the location using the UUID of the hard disk. This applies only to hard disks in NotCreated state, e.g. before the storage unit is created, and works as follows. You set the value of the location attribute to a location specification which only contains the path specification but not the file name part and ends with either a forward slash or a backslash character. In response, VirtualBox will generate a new UUID for the hard disk and compose the file name using the following pattern:

```
<path>/{<uuid>}.<ext>
```

where `<path>` is the supplied path specification, `<uuid>` is the newly generated UUID and `<ext>` is the default extension for the storage format of this hard disk. After that, you may call any of the methods that create a new hard disk storage unit and they will use the generated UUID and file name.

5.50.1 Attributes

5.50.1.1 id (read-only)

```
uuid IMedium::id
```

UUID of the medium. For a newly created medium, this value is a randomly generated UUID.

> **Note:** For media in one of MediumState_NotCreated, MediumState_Creating or MediumState_Deleting states, the value of this property is undefined and will most likely be an empty UUID.

5.50.1.2 description (read/write)

```
wstring IMedium::description
```

Optional description of the medium. For a newly created medium the value of this attribute is an empty string.

Medium types that don't support this attribute will return E_NOTIMPL in attempt to get or set this attribute's value.

Note: For some storage types, reading this attribute may return an outdated (last known) value when state is Inaccessible or LockedWrite because the value of this attribute is stored within the storage unit itself. Also note that changing the attribute value is not possible in such case, as well as when the medium is the LockedRead state.

5.50.1.3 state (read-only)

`MediumState IMedium::state`

Returns the current medium state, which is the last state set by the accessibility check performed by refreshState(). If that method has not yet been called on the medium, the state is "Inaccessible"; as opposed to truly inaccessible media, the value of lastAccessError will be an empty string in that case.

Note: As of version 3.1, this no longer performs an accessibility check automatically; call refreshState() for that.

5.50.1.4 variant (read-only)

`unsigned long IMedium::variant`

Returns the storage format variant information for this medium as a combination of the flags described at MediumVariant. Before refreshState() is called this method returns an undefined value.

5.50.1.5 location (read/write)

`wstring IMedium::location`

Location of the storage unit holding medium data.

The format of the location string is medium type specific. For medium types using regular files in a host's file system, the location string is the full file name.

Some medium types may support changing the storage unit location by simply changing the value of this property. If this operation is not supported, the implementation will return E_NOTIMPL in attempt to set this attribute's value.

When setting a value of the location attribute which is a regular file in the host's file system, the given file name may be either relative to the VirtualBox home folder or absolute. Note that if the given location specification does not contain the file extension part then a proper default extension will be automatically appended by the implementation depending on the medium type.

5.50.1.6 name (read-only)

`wstring IMedium::name`

Name of the storage unit holding medium data.

The returned string is a short version of the location attribute that is suitable for representing the medium in situations where the full location specification is too long (such as lists and comboboxes in GUI frontends). This string is also used by frontends to sort the media list alphabetically when needed.

For example, for locations that are regular files in the host's file system, the value of this attribute is just the file name (+ extension), without the path specification.

Note that as opposed to the location attribute, the name attribute will not necessary be unique for a list of media of the given type and format.

5.50.1.7 deviceType (read-only)

`DeviceType IMedium::deviceType`

Kind of device (DVD/Floppy/HardDisk) which is applicable to this medium.

5.50.1.8 hostDrive (read-only)

`boolean IMedium::hostDrive`

True if this corresponds to a drive on the host.

5.50.1.9 size (read-only)

`long long IMedium::size`

Physical size of the storage unit used to hold medium data (in bytes).

> **Note:** For media whose state is Inaccessible, the value of this property is the last known size. For NotCreated media, the returned value is zero.

5.50.1.10 format (read-only)

`wstring IMedium::format`

Storage format of this medium.

The value of this attribute is a string that specifies a backend used to store medium data. The storage format is defined when you create a new medium or automatically detected when you open an existing medium, and cannot be changed later.

The list of all storage formats supported by this VirtualBox installation can be obtained using ISystemProperties::mediumFormats[].

5.50.1.11 mediumFormat (read-only)

`IMediumFormat IMedium::mediumFormat`

Storage medium format object corresponding to this medium.

The value of this attribute is a reference to the medium format object that specifies the backend properties used to store medium data. The storage format is defined when you create a new medium or automatically detected when you open an existing medium, and cannot be changed later.

> **Note:** `null` is returned if there is no associated medium format object. This can e.g. happen for medium objects representing host drives and other special medium objects.

5.50.1.12 type (read/write)

`MediumType IMedium::type`

Type (role) of this medium.
The following constraints apply when changing the value of this attribute:

- If a medium is attached to a virtual machine (either in the current state or in one of the snapshots), its type cannot be changed.
- As long as the medium has children, its type cannot be set to Writethrough.
- The type of all differencing media is Normal and cannot be changed.

The type of a newly created or opened medium is set to Normal, except for DVD and floppy media, which have a type of Writethrough.

5.50.1.13 parent (read-only)

`IMedium IMedium::parent`

Parent of this medium (the medium this medium is directly based on).
Only differencing media have parents. For base (non-differencing) media, `null` is returned.

5.50.1.14 children (read-only)

`IMedium IMedium::children[]`

Children of this medium (all differencing media directly based on this medium). A `null` array is returned if this medium does not have any children.

5.50.1.15 base (read-only)

`IMedium IMedium::base`

Base medium of this medium.
If this is a differencing medium, its base medium is the medium the given medium branch starts from. For all other types of media, this property returns the medium object itself (i.e. the same object this property is read on).

5.50.1.16 readOnly (read-only)

`boolean IMedium::readOnly`

Returns `true` if this medium is read-only and `false` otherwise.
A medium is considered to be read-only when its contents cannot be modified without breaking the integrity of other parties that depend on this medium such as its child media or snapshots of virtual machines where this medium is attached to these machines. If there are no children and no such snapshots then there is no dependency and the medium is not read-only.
The value of this attribute can be used to determine the kind of the attachment that will take place when attaching this medium to a virtual machine. If the value is `false` then the medium will be attached directly. If the value is `true` then the medium will be attached indirectly by creating a new differencing child medium for that. See the interface description for more information.
Note that all Immutable media are always read-only while all Writethrough media are always not.

Note: The read-only condition represented by this attribute is related to the medium type and usage, not to the current medium state and not to the read-only state of the storage unit.

5.50.1.17 logicalSize (read-only)

`long long IMedium::logicalSize`

Logical size of this medium (in bytes), as reported to the guest OS running inside the virtual machine this medium is attached to. The logical size is defined when the medium is created and cannot be changed later.

Note: Reading this property on a differencing medium will return the size of its base medium.

Note: For media whose state is state is Inaccessible, the value of this property is the last known logical size. For NotCreated media, the returned value is zero.

5.50.1.18 autoReset (read/write)

`boolean IMedium::autoReset`

Whether this differencing medium will be automatically reset each time a virtual machine it is attached to is powered up. This attribute is automatically set to `true` for the last differencing image of an "immutable" medium (see MediumType).

See reset() for more information about resetting differencing media.

Note: Reading this property on a base (non-differencing) medium will always `false`. Changing the value of this property in this case is not supported.

5.50.1.19 lastAccessError (read-only)

`wstring IMedium::lastAccessError`

Text message that represents the result of the last accessibility check performed by refreshState().

An empty string is returned if the last accessibility check was successful or has not yet been called. As a result, if state is "Inaccessible" and this attribute is empty, then refreshState() has yet to be called; this is the default value of media after VirtualBox initialization. A non-empty string indicates a failure and should normally describe a reason of the failure (for example, a file read error).

5.50.1.20 machineIds (read-only)

```
uuid IMedium::machineIds[]
```

Array of UUIDs of all machines this medium is attached to.

A `null` array is returned if this medium is not attached to any machine or to any machine's snapshot.

> **Note:** The returned array will include a machine even if this medium is not attached to that machine in the current state but attached to it in one of the machine's snapshots. See getSnapshotIds() for details.

5.50.2 cloneTo

```
IProgress IMedium::cloneTo(
            [in] IMedium target,
            [in] unsigned long variant,
            [in] IMedium parent)
```

target Target medium.

variant Exact image variant which should be created (as a combination of MediumVariant flags).

parent Parent of the cloned medium.

Starts creating a clone of this medium in the format and at the location defined by the `target` argument.

The target medium must be either in NotCreated state (i.e. must not have an existing storage unit) or in Created state (i.e. created and not locked, and big enough to hold the data or else the copy will be partial). Upon successful completion, the cloned medium will contain exactly the same sector data as the medium being cloned, except that in the first case a new UUID for the clone will be randomly generated, and in the second case the UUID will remain unchanged.

The `parent` argument defines which medium will be the parent of the clone. Passing a `null` reference indicates that the clone will be a base image, i.e. completely independent. It is possible to specify an arbitrary medium for this parameter, including the parent of the medium which is being cloned. Even cloning to a child of the source medium is possible. Note that when cloning to an existing image, the `parent` irgument is ignored.

After the returned progress object reports that the operation is successfully complete, the target medium gets remembered by this VirtualBox installation and may be attached to virtual machines.

> **Note:** This medium will be placed to LockedRead state for the duration of this operation.

If this method fails, the following error codes may be reported:

- `E_NOTIMPL:` The specified cloning variant is not supported at the moment.

5.50.3 close

```
void IMedium::close()
```

Closes this medium.

The medium must not be attached to any known virtual machine and must not have any known child media, otherwise the operation will fail.

When the medium is successfully closed, it is removed from the list of registered media, but its storage unit is not deleted. In particular, this means that this medium can later be opened again using the IVirtualBox::openMedium() call.

Note that after this method successfully returns, the given medium object becomes uninitialized. This means that any attempt to call any of its methods or attributes will fail with the `"Object not ready" (E_ACCESSDENIED)` error.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_OBJECT_STATE:` Invalid medium state (other than not created, created or inaccessible).
- `VBOX_E_OBJECT_IN_USE:` Medium attached to virtual machine.
- `VBOX_E_FILE_ERROR:` Settings file not accessible.
- `VBOX_E_XML_ERROR:` Could not parse the settings file.

5.50.4 compact

```
IProgress IMedium::compact()
```

Starts compacting of this medium. This means that the medium is transformed into a possibly more compact storage representation. This potentially creates temporary images, which can require a substantial amount of additional disk space.

This medium will be placed to LockedWrite state and all its parent media (if any) will be placed to LockedRead state for the duration of this operation.

Please note that the results can be either returned straight away, or later as the result of the background operation via the object returned via the `progress` parameter.

If this method fails, the following error codes may be reported:

- `VBOX_E_NOT_SUPPORTED:` Medium format does not support compacting (but potentially needs it).

5.50.5 createBaseStorage

```
IProgress IMedium::createBaseStorage(
            [in] long long logicalSize,
            [in] unsigned long variant)
```

logicalSize Maximum logical size of the medium in bytes.

variant Exact image variant which should be created (as a combination of MediumVariant flags).

Starts creating a hard disk storage unit (fixed/dynamic, according to the variant flags) in in the background. The previous storage unit created for this object, if any, must first be deleted using deleteStorage(), otherwise the operation will fail.

Before the operation starts, the medium is placed in Creating state. If the create operation fails, the medium will be placed back in NotCreated state.

After the returned progress object reports that the operation has successfully completed, the medium state will be set to Created, the medium will be remembered by this VirtualBox installation and may be attached to virtual machines.

If this method fails, the following error codes may be reported:

- `VBOX_E_NOT_SUPPORTED:` The variant of storage creation operation is not supported. See IMediumFormat::capabilities.

5.50.6 createDiffStorage

```
IProgress IMedium::createDiffStorage(
            [in] IMedium target,
            [in] unsigned long variant)
```

target Target medium.

variant Exact image variant which should be created (as a combination of MediumVariant flags).

Starts creating an empty differencing storage unit based on this medium in the format and at the location defined by the `target` argument.

The target medium must be in NotCreated state (i.e. must not have an existing storage unit). Upon successful completion, this operation will set the type of the target medium to Normal and create a storage unit necessary to represent the differencing medium data in the given format (according to the storage format of the target object).

After the returned progress object reports that the operation is successfully complete, the target medium gets remembered by this VirtualBox installation and may be attached to virtual machines.

> **Note:** The medium will be set to LockedRead state for the duration of this operation.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_IN_USE`: Medium not in `NotCreated` state.

5.50.7 deleteStorage

```
IProgress IMedium::deleteStorage()
```

Starts deleting the storage unit of this medium.

The medium must not be attached to any known virtual machine and must not have any known child media, otherwise the operation will fail. It will also fail if there is no storage unit to delete or if deletion is already in progress, or if the medium is being in use (locked for read or for write) or inaccessible. Therefore, the only valid state for this operation to succeed is Created.

Before the operation starts, the medium is placed in Deleting state and gets removed from the list of remembered hard disks (media registry). If the delete operation fails, the medium will be remembered again and placed back to Created state.

After the returned progress object reports that the operation is complete, the medium state will be set to NotCreated and you will be able to use one of the storage creation methods to create it again.

See also: #close()

> **Note:** If the deletion operation fails, it is not guaranteed that the storage unit still exists. You may check the state value to answer this question.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_IN_USE`: Medium is attached to a virtual machine.
- `VBOX_E_NOT_SUPPORTED`: Storage deletion is not allowed because neither of storage creation operations are supported. See IMediumFormat::capabilities.

5.50.8 getProperties

```
wstring[] IMedium::getProperties(
            [in] wstring names,
            [out] wstring returnNames[])
```

names Names of properties to get.

returnNames Names of returned properties.

Returns values for a group of properties in one call.

The names of the properties to get are specified using the `names` argument which is a list of comma-separated property names or an empty string if all properties are to be returned. Note that currently the value of this argument is ignored and the method always returns all existing properties.

The list of all properties supported by the given medium format can be obtained with IMediumFormat::describeProperties().

The method returns two arrays, the array of property names corresponding to the `names` argument and the current values of these properties. Both arrays have the same number of elements with each elemend at the given index in the first array corresponds to an element at the same index in the second array.

Note that for properties that do not have assigned values, an empty string is returned at the appropriate index in the `returnValues` array.

5.50.9 getProperty

```
wstring IMedium::getProperty(
            [in] wstring name)
```

name Name of the property to get.

Returns the value of the custom medium property with the given name.

The list of all properties supported by the given medium format can be obtained with IMediumFormat::describeProperties().

Note that if this method returns an empty string in `value`, the requested property is supported but currently not assigned any value.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: Requested property does not exist (not supported by the format).
- `E_INVALIDARG`: `name` is `null` or empty.

5.50.10 getSnapshotIds

```
uuid[] IMedium::getSnapshotIds(
            [in] uuid machineId)
```

machineId UUID of the machine to query.

Returns an array of UUIDs of all snapshots of the given machine where this medium is attached to.

If the medium is attached to the machine in the current state, then the first element in the array will always be the ID of the queried machine (i.e. the value equal to the `machineId` argument), followed by snapshot IDs (if any).

If the medium is not attached to the machine in the current state, then the array will contain only snapshot IDs.

The returned array may be `null` if this medium is not attached to the given machine at all, neither in the current state nor in one of the snapshots.

5.50.11 lockRead

`MediumState IMedium::lockRead()`

Locks this medium for reading.

A read lock is shared: many clients can simultaneously lock the same medium for reading unless it is already locked for writing (see lockWrite()) in which case an error is returned.

When the medium is locked for reading, it cannot be modified from within VirtualBox. This means that any method that changes the properties of this medium or contents of the storage unit will return an error (unless explicitly stated otherwise). That includes an attempt to start a virtual machine that wants to write to the the medium.

When the virtual machine is started up, it locks for reading all media it uses in read-only mode. If some medium cannot be locked for reading, the startup procedure will fail. A medium is typically locked for reading while it is used by a running virtual machine but has a depending differencing image that receives the actual write operations. This way one base medium can have multiple child differencing images which can be written to simultaneously. Read-only media such as DVD and floppy images are also locked for reading only (so they can be in use by multiple machines simultaneously).

A medium is also locked for reading when it is the source of a write operation such as cloneTo() or mergeTo().

The medium locked for reading must be unlocked using the unlockRead() method. Calls to lockRead() can be nested and must be followed by the same number of paired unlockRead() calls.

This method sets the medium state (see state) to "LockedRead" on success. The medium's previous state must be one of "Created", "Inaccessible" or "LockedRead".

Locking an inaccessible medium is not an error; this method performs a logical lock that prevents modifications of this medium through the VirtualBox API, not a physical file-system lock of the underlying storage unit.

This method returns the current state of the medium *before* the operation.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_OBJECT_STATE`: Invalid medium state (e.g. not created, locked, inaccessible, creating, deleting).

5.50.12 lockWrite

`MediumState IMedium::lockWrite()`

Locks this medium for writing.

A write lock, as opposed to lockRead(), is exclusive: there may be only one client holding a write lock, and there may be no read locks while the write lock is held. As a result, read-locking fails if a write lock is held, and write-locking fails if either a read or another write lock is held.

When a medium is locked for writing, it cannot be modified from within VirtualBox, and it is not guaranteed that the values of its properties are up-to-date. Any method that changes the properties of this medium or contents of the storage unit will return an error (unless explicitly stated otherwise).

When a virtual machine is started up, it locks for writing all media it uses to write data to. If any medium could not be locked for writing, the startup procedure will fail. If a medium has differencing images, then while the machine is running, only the last ("leaf") differencing image is locked for writing, whereas its parents are locked for reading only.

A medium is also locked for writing when it is the target of a write operation such as cloneTo() or mergeTo().

The medium locked for writing must be unlocked using the unlockWrite() method. Write locks *cannot* be nested.

This method sets the medium state (see state) to “LockedWrite” on success. The medium’s previous state must be either “Created” or “Inaccessible”.

Locking an inaccessible medium is not an error; this method performs a logical lock that prevents modifications of this medium through the VirtualBox API, not a physical file-system lock of the underlying storage unit.

For both, success and failure, this method returns the current state of the medium *before* the operation.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_OBJECT_STATE`: Invalid medium state (e.g. not created, locked, inaccessible, creating, deleting).

5.50.13 mergeTo

```
IProgress IMedium::mergeTo(
            [in] IMedium target)
```

target Target medium.

Starts merging the contents of this medium and all intermediate differencing media in the chain to the given target medium.

The target medium must be either a descendant of this medium or its ancestor (otherwise this method will immediately return a failure). It follows that there are two logical directions of the merge operation: from ancestor to descendant (*forward merge*) and from descendant to ancestor (*backward merge*). Let us consider the following medium chain:

```
Base <- Diff_1 <- Diff_2
```

Here, calling this method on the `Base` medium object with `Diff_2` as an argument will be a forward merge; calling it on `Diff_2` with `Base` as an argument will be a backward merge. Note that in both cases the contents of the resulting medium will be the same, the only difference is the medium object that takes the result of the merge operation. In case of the forward merge in the above example, the result will be written to `Diff_2`; in case of the backward merge, the result will be written to `Base`. In other words, the result of the operation is always stored in the target medium.

Upon successful operation completion, the storage units of all media in the chain between this (source) medium and the target medium, including the source medium itself, will be automatically deleted and the relevant medium objects (including this medium) will become uninitialized. This means that any attempt to call any of their methods or attributes will fail with the `"Object not ready" (E_ACCESSDENIED)` error. Applied to the above example, the forward merge of `Base` to `Diff_2` will delete and uninitialize both `Base` and `Diff_1` media. Note that `Diff_2` in this case will become a base medium itself since it will no longer be based on any other medium.

Considering the above, all of the following conditions must be met in order for the merge operation to succeed:

- Neither this (source) medium nor any intermediate differencing medium in the chain between it and the target medium is attached to any virtual machine.
- Neither the source medium nor the target medium is an Immutable medium.
- The part of the medium tree from the source medium to the target medium is a linear chain, i.e. all medium in this chain have exactly one child which is the next medium in this chain. The only exception from this rule is the target medium in the forward merge operation; it is allowed to have any number of child media because the merge operation will not change its logical contents (as it is seen by the guest OS or by children).
- None of the involved media are in LockedRead or LockedWrite state.

Note: This (source) medium and all intermediates will be placed to Deleting state and the target medium will be placed to LockedWrite state and for the duration of this operation.

5.50.14 refreshState

```
MediumState IMedium::refreshState()
```

If the current medium state (see MediumState) is one of "Created", "Inaccessible" or "LockedRead", then this performs an accessibility check on the medium and sets the value of the state attribute accordingly; that value is also returned for convenience.

For all other state values, this does not perform a refresh but returns the state only.

The refresh, if performed, may take a long time (several seconds or even minutes, depending on the storage unit location and format) because it performs an accessibility check of the storage unit. This check may cause a significant delay if the storage unit of the given medium is, for example, a file located on a network share which is not currently accessible due to connectivity problems. In that case, the call will not return until a timeout interval defined by the host OS for this operation expires. For this reason, it is recommended to never read this attribute on the main UI thread to avoid making the UI unresponsive.

If the last known state of the medium is "Created" and the accessibility check fails, then the state would be set to "Inaccessible", and lastAccessError may be used to get more details about the failure. If the state of the medium is "LockedRead", then it remains the same, and a non-empty value of lastAccessError will indicate a failed accessibility check in this case.

Note that not all medium states are applicable to all medium types.

5.50.15 reset

```
IProgress IMedium::reset()
```

Starts erasing the contents of this differencing medium.

This operation will reset the differencing medium to its initial state when it does not contain any sector data and any read operation is redirected to its parent medium. This automatically gets called during VM power-up for every medium whose autoReset attribute is `true`.

The medium will be write-locked for the duration of this operation (see lockWrite()).

If this method fails, the following error codes may be reported:

- `VBOX_E_NOT_SUPPORTED`: This is not a differencing medium.
- `VBOX_E_INVALID_OBJECT_STATE`: Medium is not in Created or Inaccessible state.

5.50.16 resize

```
IProgress IMedium::resize(
            [in] long long logicalSize)
```

logicalSize New nominal capacity of the medium in bytes.

Starts resizing this medium. This means that the nominal size of the medium is set to the new value. Both increasing and decreasing the size is possible, and there are no safety checks, since VirtualBox does not make any assumptions about the medium contents.

Resizing usually needs additional disk space, and possibly also some temporary disk space. Note that resize does not create a full temporary copy of the medium, so the additional disk space requirement is usually much lower than using the clone operation.

This medium will be placed to LockedWrite state for the duration of this operation.

Please note that the results can be either returned straight away, or later as the result of the background operation via the object returned via the `progress` parameter.

If this method fails, the following error codes may be reported:

- `VBOX_E_NOT_SUPPORTED`: Medium format does not support resizing.

5.50.17 setIDs

```
void IMedium::setIDs(
            [in] boolean setImageId,
            [in] uuid imageId,
            [in] boolean setParentId,
            [in] uuid parentId)
```

setImageId Select whether a new image UUID is set or not.

imageId New UUID for the image. If an empty string is passed, then a new UUID is automatically created, provided that `setImageId` is `true`. Specifying a zero UUID is not allowed.

setParentId Select whether a new parent UUID is set or not.

parentId New parent UUID for the image. If an empty string is passed, then a new UUID is automatically created, provided `setParentId` is `true`. A zero UUID is valid.

Changes the UUID and parent UUID for a hard disk medium.

5.50.18 setProperties

```
void IMedium::setProperties(
            [in] wstring names[],
            [in] wstring values[])
```

names Names of properties to set.

values Values of properties to set.

Sets values for a group of properties in one call.

The names of the properties to set are passed in the `names` array along with the new values for them in the `values` array. Both arrays have the same number of elements with each elemend at the given index in the first array corresponding to an element at the same index in the second array.

If there is at least one property name in `names` that is not valid, the method will fail before changing the values of any other properties from the `names` array.

Using this method over setProperty() is preferred if you need to set several properties at once since it will result into less IPC calls.

The list of all properties supported by the given medium format can be obtained with IMediumFormat::describeProperties().

Note that setting the property value to `null` or an empty string is equivalent to deleting the existing value. A default value (if it is defined for this property) will be used by the format backend in this case.

5.50.19 setProperty

```
void IMedium::setProperty(
            [in] wstring name,
            [in] wstring value)
```

name Name of the property to set.

value Property value to set.

Sets the value of the custom medium property with the given name.

The list of all properties supported by the given medium format can be obtained with IMediumFormat::describeProperties().

Note that setting the property value to `null` or an empty string is equivalent to deleting the existing value. A default value (if it is defined for this property) will be used by the format backend in this case.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: Requested property does not exist (not supported by the format).
- `E_INVALIDARG`: `name` is `null` or empty.

5.50.20 unlockRead

`MediumState IMedium::unlockRead()`

Cancels the read lock previously set by lockRead().

For both success and failure, this method returns the current state of the medium *after* the operation.

See lockRead() for more details.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_OBJECT_STATE`: Medium not locked for reading.

5.50.21 unlockWrite

`MediumState IMedium::unlockWrite()`

Cancels the write lock previously set by lockWrite().

For both success and failure, this method returns the current state of the medium *after* the operation.

See lockWrite() for more details.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_OBJECT_STATE`: Medium not locked for writing.

5.51 IMediumAttachment

> **Note:** With the web service, this interface is mapped to a structure. Attributes that return this interface will not return an object, but a complete structure containing the attributes listed below as structure members.

The IMediumAttachment interface links storage media to virtual machines. For each medium (IMedium) which has been attached to a storage controller (IStorageController) of a machine (IMachine) via the IMachine::attachDevice() method, one instance of IMediumAttachment is added to the machine's IMachine::mediumAttachments[] array attribute.

Each medium attachment specifies the storage controller as well as a port and device number and the IMedium instance representing a virtual hard disk or floppy or DVD image.

For removable media (DVDs or floppies), there are two additional options. For one, the IMedium instance can be `null` to represent an empty drive with no media inserted (see IMachine::mountMedium()); secondly, the medium can be one of the pseudo-media for host drives listed in IHost::DVDDrives[] or IHost::floppyDrives[].

Attaching Hard Disks

Hard disks are attached to virtual machines using the IMachine::attachDevice() method and detached using the IMachine::detachDevice() method. Depending on a medium's type (see IMedium::type), hard disks are attached either *directly* or *indirectly*.

When a hard disk is being attached directly, it is associated with the virtual machine and used for hard disk operations when the machine is running. When a hard disk is being attached indirectly, a new differencing hard disk linked to it is implicitly created and this differencing hard disk is associated with the machine and used for hard disk operations. This also means that if IMachine::attachDevice() performs a direct attachment then the same hard disk will be returned in response to the subsequent IMachine::getMedium() call; however if an indirect attachment is performed then IMachine::getMedium() will return the implicitly created differencing hard disk, not the original one passed to IMachine::attachDevice(). In detail:

- **Normal base** hard disks that do not have children (i.e. differencing hard disks linked to them) and that are not already attached to virtual machines in snapshots are attached **directly**. Otherwise, they are attached **indirectly** because having dependent children or being part of the snapshot makes it impossible to modify hard disk contents without breaking the integrity of the dependent party. The IMedium::readOnly attribute allows to quickly determine the kind of the attachment for the given hard disk. Note that if a normal base hard disk is to be indirectly attached to a virtual machine with snapshots then a special procedure called *smart attachment* is performed (see below).

- **Normal differencing** hard disks are like normal base hard disks: they are attached **directly** if they do not have children and are not attached to virtual machines in snapshots, and **indirectly** otherwise. Note that the smart attachment procedure is never performed for differencing hard disks.

- **Immutable** hard disks are always attached **indirectly** because they are designed to be non-writable. If an immutable hard disk is attached to a virtual machine with snapshots then a special procedure called smart attachment is performed (see below).

- **Writethrough** hard disks are always attached **directly**, also as designed. This also means that writethrough hard disks cannot have other hard disks linked to them at all.

- **Shareable** hard disks are always attached **directly**, also as designed. This also means that shareable hard disks cannot have other hard disks linked to them at all. They behave almost like writethrough hard disks, except that shareable hard disks can be attached to several virtual machines which are running, allowing concurrent accesses. You need special cluster software running in the virtual machines to make use of such disks.

Note that the same hard disk, regardless of its type, may be attached to more than one virtual machine at a time. In this case, the machine that is started first gains exclusive access to the hard disk and attempts to start other machines having this hard disk attached will fail until the first machine is powered down.

Detaching hard disks is performed in a *deferred* fashion. This means that the given hard disk remains associated with the given machine after a successful IMachine::detachDevice() call until IMachine::saveSettings() is called to save all changes to machine settings to disk. This deferring is necessary to guarantee that the hard disk configuration may be restored at any time by a call to IMachine::discardSettings() before the settings are saved (committed).

Note that if IMachine::discardSettings() is called after indirectly attaching some hard disks to the machine but before a call to IMachine::saveSettings() is made, it will implicitly delete

all differencing hard disks implicitly created by IMachine::attachDevice() for these indirect attachments. Such implicitly created hard disks will also be immediately deleted when detached explicitly using the IMachine::detachDevice() call if it is made before IMachine::saveSettings(). This implicit deletion is safe because newly created differencing hard disks do not contain any user data.

However, keep in mind that detaching differencing hard disks that were implicitly created by IMachine::attachDevice() before the last IMachine::saveSettings() call will **not** implicitly delete them as they may already contain some data (for example, as a result of virtual machine execution). If these hard disks are no more necessary, the caller can always delete them explicitly using IMedium::deleteStorage() after they are actually de-associated from this machine by the IMachine::saveSettings() call.

Smart Attachment

When normal base or immutable hard disks are indirectly attached to a virtual machine then some additional steps are performed to make sure the virtual machine will have the most recent "view" of the hard disk being attached. These steps include walking through the machine's snapshots starting from the current one and going through ancestors up to the first snapshot. Hard disks attached to the virtual machine in all of the encountered snapshots are checked whether they are descendants of the given normal base or immutable hard disk. The first found child (which is the differencing hard disk) will be used instead of the normal base or immutable hard disk as a parent for creating a new differencing hard disk that will be actually attached to the machine. And only if no descendants are found or if the virtual machine does not have any snapshots then the normal base or immutable hard disk will be used itself as a parent for this differencing hard disk.

It is easier to explain what smart attachment does using the following example:

```
BEFORE attaching B.vdi:       AFTER attaching B.vdi:

Snapshot 1 (B.vdi)            Snapshot 1 (B.vdi)
 Snapshot 2 (D1->B.vdi)        Snapshot 2 (D1->B.vdi)
  Snapshot 3 (D2->D1.vdi)       Snapshot 3 (D2->D1.vdi)
   Snapshot 4 (none)             Snapshot 4 (none)
    CurState   (none)             CurState   (D3->D2.vdi)

                              NOT
                                 ...
                                  CurState   (D3->B.vdi)
```

The first column is the virtual machine configuration before the base hard disk `B.vdi` is attached, the second column shows the machine after this hard disk is attached. Constructs like `D1->B.vdi` and similar mean that the hard disk that is actually attached to the machine is a differencing hard disk, `D1.vdi`, which is linked to (based on) another hard disk, `B.vdi`.

As we can see from the example, the hard disk `B.vdi` was detached from the machine before taking Snapshot 4. Later, after Snapshot 4 was taken, the user decides to attach `B.vdi` again. `B.vdi` has dependent child hard disks (`D1.vdi`, `D2.vdi`), therefore it cannot be attached directly and needs an indirect attachment (i.e. implicit creation of a new differencing hard disk). Due to the smart attachment procedure, the new differencing hard disk (`D3.vdi`) will be based on `D2.vdi`, not on `B.vdi` itself, since `D2.vdi` is the most recent view of `B.vdi` existing for this snapshot branch of the given virtual machine.

Note that if there is more than one descendant hard disk of the given base hard disk found in a snapshot, and there is an exact device, channel and bus match, then this exact match will be used. Otherwise, the youngest descendant will be picked up.

There is one more important aspect of the smart attachment procedure which is not related to snapshots at all. Before walking through the snapshots as described above, the backup copy of the current list of hard disk attachment is searched for descendants. This backup copy is created when the hard disk configuration is changed for the first time after the last

IMachine::saveSettings() call and used by IMachine::discardSettings() to undo the recent hard disk changes. When such a descendant is found in this backup copy, it will be simply re-attached back, without creating a new differencing hard disk for it. This optimization is necessary to make it possible to re-attach the base or immutable hard disk to a different bus, channel or device slot without losing the contents of the differencing hard disk actually attached to the machine in place of it.

5.51.1 Attributes

5.51.1.1 medium (read-only)

```
IMedium IMediumAttachment::medium
```

Medium object associated with this attachment; it can be `null` for removable devices.

5.51.1.2 controller (read-only)

```
wstring IMediumAttachment::controller
```

Name of the storage controller of this attachment; this refers to one of the controllers in IMachine::storageControllers[] by name.

5.51.1.3 port (read-only)

```
long IMediumAttachment::port
```

Port number of this attachment. See IMachine::attachDevice() for the meaning of this value for the different controller types.

5.51.1.4 device (read-only)

```
long IMediumAttachment::device
```

Device slot number of this attachment. See IMachine::attachDevice() for the meaning of this value for the different controller types.

5.51.1.5 type (read-only)

```
DeviceType IMediumAttachment::type
```

Device type of this attachment.

5.51.1.6 passthrough (read-only)

```
boolean IMediumAttachment::passthrough
```

Pass I/O requests through to a device on the host.

5.51.1.7 bandwidthGroup (read-only)

```
IBandwidthGroup IMediumAttachment::bandwidthGroup
```

The bandwidth group this medium attachment is assigned to.

5.52 IMediumChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a medium attachment changes.

5.52.1 Attributes

5.52.1.1 mediumAttachment (read-only)

```
IMediumAttachment IMediumChangedEvent::mediumAttachment
```

Medium attachment that is subject to change.

5.53 IMediumFormat

The IMediumFormat interface represents a medium format.

Each medium format has an associated backend which is used to handle media stored in this format. This interface provides information about the properties of the associated backend.

Each medium format is identified by a string represented by the id attribute. This string is used in calls like IVirtualBox::createHardDisk() to specify the desired format.

The list of all supported medium formats can be obtained using ISystemProperties::mediumFormats[].

See also: IMedium

5.53.1 Attributes

5.53.1.1 id (read-only)

```
wstring IMediumFormat::id
```

Identifier of this format.

The format identifier is a non-null non-empty ASCII string. Note that this string is case-insensitive. This means that, for example, all of the following strings:

```
"VDI"
"vdi"
"VdI"
```

refer to the same medium format.

This string is used in methods of other interfaces where it is necessary to specify a medium format, such as IVirtualBox::createHardDisk().

5.53.1.2 name (read-only)

```
wstring IMediumFormat::name
```

Human readable description of this format.

Mainly for use in file open dialogs.

5.53.1.3 capabilities (read-only)

```
unsigned long IMediumFormat::capabilities
```

Capabilities of the format as a set of bit flags.

For the meaning of individual capability flags see MediumFormatCapabilities.

5.53.2 describeFileExtensions

```
void IMediumFormat::describeFileExtensions(
            [out] wstring extensions[],
            [out] DeviceType type[])
```

extensions The array of supported extensions.

type The array which indicates the device type for every given extension.

Returns two arrays describing the supported file extensions.
The first array contains the supported extensions and the seconds one the type each extension supports. Both have the same size.
Note that some backends do not work on files, so this array may be empty.
See also: IMediumFormat::capabilities

5.53.3 describeProperties

```
void IMediumFormat::describeProperties(
            [out] wstring names[],
            [out] wstring description[],
            [out] DataType types[],
            [out] unsigned long flags[],
            [out] wstring defaults[])
```

names Array of property names.

description Array of property descriptions.

types Array of property types.

flags Array of property flags.

defaults Array of default property values.

Returns several arrays describing the properties supported by this format.
An element with the given index in each array describes one property. Thus, the number of elements in each returned array is the same and corresponds to the number of supported properties.
The returned arrays are filled in only if the Properties flag is set. All arguments must be non-`null`.
See also: DataTypeSee also: DataFlags

5.54 IMediumRegisteredEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

The given medium was registered or unregistered within this VirtualBox installation.

5.54.1 Attributes

5.54.1.1 mediumId (read-only)

```
uuid IMediumRegisteredEvent::mediumId
```

ID of the medium this event relates to.

5.54.1.2 mediumType (read-only)

`DeviceType IMediumRegisteredEvent::mediumType`

Type of the medium this event relates to.

5.54.1.3 registered (read-only)

`boolean IMediumRegisteredEvent::registered`

If `true`, the medium was registered, otherwise it was unregistered.

5.55 IMouse

The IMouse interface represents the virtual machine's mouse. Used in IConsole::mouse.

Through this interface, the virtual machine's virtual mouse can be controlled.

5.55.1 Attributes

5.55.1.1 absoluteSupported (read-only)

`boolean IMouse::absoluteSupported`

Whether the guest OS supports absolute mouse pointer positioning or not.

> **Note:** You can use the IMouseCapabilityChangedEvent event to be instantly informed about changes of this attribute during virtual machine execution.

See also: putMouseEventAbsolute()

5.55.1.2 relativeSupported (read-only)

`boolean IMouse::relativeSupported`

Whether the guest OS supports relative mouse pointer positioning or not.

> **Note:** You can use the IMouseCapabilityChangedEvent event to be instantly informed about changes of this attribute during virtual machine execution.

See also: putMouseEvent()

5.55.1.3 needsHostCursor (read-only)

`boolean IMouse::needsHostCursor`

Whether the guest OS can currently switch to drawing it's own mouse cursor on demand.

> **Note:** You can use the IMouseCapabilityChangedEvent event to be instantly informed about changes of this attribute during virtual machine execution.

See also: putMouseEvent()

5.55.1.4 eventSource (read-only)

```
IEventSource IMouse::eventSource
```

Event source for mouse events.

5.55.2 putMouseEvent

```
void IMouse::putMouseEvent(
            [in] long dx,
            [in] long dy,
            [in] long dz,
            [in] long dw,
            [in] long buttonState)
```

dx Amount of pixels the mouse should move to the right. Negative values move the mouse to the left.

dy Amount of pixels the mouse should move downwards. Negative values move the mouse upwards.

dz Amount of mouse wheel moves. Positive values describe clockwise wheel rotations, negative values describe counterclockwise rotations.

dw Amount of horizontal mouse wheel moves. Positive values describe a movement to the left, negative values describe a movement to the right.

buttonState The current state of mouse buttons. Every bit represents a mouse button as follows: Bit 0 (0x01)left mouse buttonBit 1 (0x02)right mouse buttonBit 2 (0x04)middle mouse button A value of 1 means the corresponding button is pressed. otherwise it is released.

Initiates a mouse event using relative pointer movements along x and y axis.
If this method fails, the following error codes may be reported:

- E_ACCESSDENIED: Console not powered up.
- VBOX_E_IPRT_ERROR: Could not send mouse event to virtual mouse.

5.55.3 putMouseEventAbsolute

```
void IMouse::putMouseEventAbsolute(
            [in] long x,
            [in] long y,
            [in] long dz,
            [in] long dw,
            [in] long buttonState)
```

x X coordinate of the pointer in pixels, starting from 1.

y Y coordinate of the pointer in pixels, starting from 1.

dz Amount of mouse wheel moves. Positive values describe clockwise wheel rotations, negative values describe counterclockwise rotations.

dw Amount of horizontal mouse wheel moves. Positive values describe a movement to the left, negative values describe a movement to the right.

buttonState The current state of mouse buttons. Every bit represents a mouse button as follows: Bit 0 (0x01)left mouse buttonBit 1 (0x02)right mouse buttonBit 2 (0x04)middle mouse button A value of 1 means the corresponding button is pressed. otherwise it is released.

Positions the mouse pointer using absolute x and y coordinates. These coordinates are expressed in pixels and start from [1,1] which corresponds to the top left corner of the virtual display.

> **Note:** This method will have effect only if absolute mouse positioning is supported by the guest OS.

See also: absoluteSupported
If this method fails, the following error codes may be reported:

- E_ACCESSDENIED: Console not powered up.
- VBOX_E_IPRT_ERROR: Could not send mouse event to virtual mouse.

5.56 IMouseCapabilityChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when the mouse capabilities reported by the guest have changed. The new capabilities are passed.

5.56.1 Attributes

5.56.1.1 supportsAbsolute (read-only)

```
boolean IMouseCapabilityChangedEvent::supportsAbsolute
```

Supports absolute coordinates.

5.56.1.2 supportsRelative (read-only)

```
boolean IMouseCapabilityChangedEvent::supportsRelative
```

Supports relative coordinates.

5.56.1.3 needsHostCursor (read-only)

```
boolean IMouseCapabilityChangedEvent::needsHostCursor
```

If host cursor is needed.

5.57 IMousePointerShapeChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when the guest mouse pointer shape has changed. The new shape data is given.

5.57.1 Attributes

5.57.1.1 visible (read-only)

`boolean IMousePointerShapeChangedEvent::visible`

Flag whether the pointer is visible.

5.57.1.2 alpha (read-only)

`boolean IMousePointerShapeChangedEvent::alpha`

Flag whether the pointer has an alpha channel.

5.57.1.3 xhot (read-only)

`unsigned long IMousePointerShapeChangedEvent::xhot`

The pointer hot spot X coordinate.

5.57.1.4 yhot (read-only)

`unsigned long IMousePointerShapeChangedEvent::yhot`

The pointer hot spot Y coordinate.

5.57.1.5 width (read-only)

`unsigned long IMousePointerShapeChangedEvent::width`

Width of the pointer shape in pixels.

5.57.1.6 height (read-only)

`unsigned long IMousePointerShapeChangedEvent::height`

Height of the pointer shape in pixels.

5.57.1.7 shape (read-only)

`octet IMousePointerShapeChangedEvent::shape[]`

Shape buffer arrays.

The `shape` buffer contains a 1-bpp (bits per pixel) AND mask followed by a 32-bpp XOR (color) mask.

For pointers without alpha channel the XOR mask pixels are 32 bit values: (lsb)BGR0(msb). For pointers with alpha channel the XOR mask consists of (lsb)BGRA(msb) 32 bit values.

An AND mask is used for pointers with alpha channel, so if the callback does not support alpha, the pointer could be displayed as a normal color pointer.

The AND mask is a 1-bpp bitmap with byte aligned scanlines. The size of the AND mask therefore is `cbAnd = (width + 7) / 8 * height`. The padding bits at the end of each scanline are undefined.

The XOR mask follows the AND mask on the next 4-byte aligned offset: `uint8_t *pXor = pAnd + (cbAnd + 3) & ~3`. Bytes in the gap between the AND and the XOR mask are undefined. The XOR mask scanlines have no gap between them and the size of the XOR mask is: `cXor = width * 4 * height`.

> **Note:** If `shape` is 0, only the pointer visibility is changed.

5.58 INATEngine

Interface for managing a NAT engine which is used with a virtual machine. This allows for changing NAT behavior such as port-forwarding rules. This interface is used in the INetworkAdapter::natDriver attribute.

5.58.1 Attributes

5.58.1.1 network (read/write)

```
wstring INATEngine::network
```

The network attribute of the NAT engine (the same value is used with built-in DHCP server to fill corresponding fields of DHCP leases).

5.58.1.2 hostIP (read/write)

```
wstring INATEngine::hostIP
```

IP of host interface to bind all opened sockets to.

> **Note:** Changing this does not change binding of port forwarding.

5.58.1.3 tftpPrefix (read/write)

```
wstring INATEngine::tftpPrefix
```

TFTP prefix attribute which is used with the built-in DHCP server to fill the corresponding fields of DHCP leases.

5.58.1.4 tftpBootFile (read/write)

```
wstring INATEngine::tftpBootFile
```

TFTP boot file attribute which is used with the built-in DHCP server to fill the corresponding fields of DHCP leases.

5.58.1.5 tftpNextServer (read/write)

```
wstring INATEngine::tftpNextServer
```

TFTP server attribute which is used with the built-in DHCP server to fill the corresponding fields of DHCP leases.

> **Note:** The preferred form is IPv4 addresses.

5.58.1.6 aliasMode (read/write)

```
unsigned long INATEngine::aliasMode
```

5.58.1.7 dnsPassDomain (read/write)

```
boolean INATEngine::dnsPassDomain
```

Whether the DHCP server should pass the DNS domain used by the host.

5.58.1.8 dnsProxy (read/write)

```
boolean INATEngine::dnsProxy
```

Whether the DHCP server (and the DNS traffic by NAT) should pass the address of the DNS proxy and process traffic using DNS servers registered on the host.

5.58.1.9 dnsUseHostResolver (read/write)

```
boolean INATEngine::dnsUseHostResolver
```

Whether the DHCP server (and the DNS traffic by NAT) should pass the address of the DNS proxy and process traffic using the host resolver mechanism.

5.58.1.10 redirects (read-only)

```
wstring INATEngine::redirects[]
```

Array of NAT port-forwarding rules in string representation, in the following format: “name,protocol id,host ip,host port,guest ip,guest port”.

5.58.2 addRedirect

```
void INATEngine::addRedirect(
            [in] wstring name,
            [in] NATProtocol proto,
            [in] wstring hostIp,
            [in] unsigned short hostPort,
            [in] wstring guestIp,
            [in] unsigned short guestPort)
```

name The name of the rule. An empty name is acceptable, in which case the NAT engine auto-generates one using the other parameters.

proto Protocol handled with the rule.

hostIp IP of the host interface to which the rule should apply. An empty ip address is acceptable, in which case the NAT engine binds the handling socket to any interface.

hostPort The port number to listen on.

guestIp The IP address of the guest which the NAT engine will forward matching packets to. An empty IP address is acceptable, in which case the NAT engine will forward packets to the first DHCP lease (x.x.x.15).

guestPort The port number to forward.

Adds a new NAT port-forwarding rule.

5.58.3 getNetworkSettings

```
void INATEngine::getNetworkSettings(
            [out] unsigned long mtu,
            [out] unsigned long sockSnd,
            [out] unsigned long sockRcv,
            [out] unsigned long TcpWndSnd,
            [out] unsigned long TcpWndRcv)
```

mtu

sockSnd

sockRcv

TcpWndSnd

TcpWndRcv

Returns network configuration of NAT engine. See setNetworkSettings() for parameter descriptions.

5.58.4 removeRedirect

```
void INATEngine::removeRedirect(
            [in] wstring name)
```

name The name of the rule to delete.

Removes a port-forwarding rule that was previously registered.

5.58.5 setNetworkSettings

```
void INATEngine::setNetworkSettings(
            [in] unsigned long mtu,
            [in] unsigned long sockSnd,
            [in] unsigned long sockRcv,
            [in] unsigned long TcpWndSnd,
            [in] unsigned long TcpWndRcv)
```

mtu MTU (maximum transmission unit) of the NAT engine in bytes.

sockSnd Capacity of the socket send buffer in bytes when creating a new socket.

sockRcv Capacity of the socket receive buffer in bytes when creating a new socket.

TcpWndSnd Initial size of the NAT engine's sending TCP window in bytes when establishing a new TCP connection.

TcpWndRcv Initial size of the NAT engine's receiving TCP window in bytes when establishing a new TCP connection.

Sets network configuration of the NAT engine.

5.59 INATRedirectEvent (IMachineEvent)

> **Note:** This interface extends IMachineEvent and therefore supports all its methods and attributes as well.

Notification when NAT redirect rule added or removed.

5.59.1 Attributes

5.59.1.1 slot (read-only)

`unsigned long INATRedirectEvent::slot`

Adapter which NAT attached to.

5.59.1.2 remove (read-only)

`boolean INATRedirectEvent::remove`

Whether rule remove or add.

5.59.1.3 name (read-only)

`wstring INATRedirectEvent::name`

Name of the rule.

5.59.1.4 proto (read-only)

`NATProtocol INATRedirectEvent::proto`

Protocol (TCP or UDP) of the redirect rule.

5.59.1.5 hostIp (read-only)

`wstring INATRedirectEvent::hostIp`

Host ip address to bind socket on.

5.59.1.6 hostPort (read-only)

`long INATRedirectEvent::hostPort`

Host port to bind socket on.

5.59.1.7 guestIp (read-only)

`wstring INATRedirectEvent::guestIp`

Guest ip address to redirect to.

5.59.1.8 guestPort (read-only)

`long INATRedirectEvent::guestPort`

Guest port to redirect to.

5.60 INetworkAdapter

Represents a virtual network adapter that is attached to a virtual machine. Each virtual machine has a fixed number of network adapter slots with one instance of this attached to each of them. Call IMachine::getNetworkAdapter() to get the network adapter that is attached to a given slot in a given machine.

Each network adapter can be in one of five attachment modes, which are represented by the NetworkAttachmentType enumeration; see the attachmentType attribute.

5.60.1 Attributes

5.60.1.1 adapterType (read/write)

`NetworkAdapterType INetworkAdapter::adapterType`

Type of the virtual network adapter. Depending on this value, VirtualBox will provide a different virtual network hardware to the guest.

5.60.1.2 slot (read-only)

`unsigned long INetworkAdapter::slot`

Slot number this adapter is plugged into. Corresponds to the value you pass to IMachine::getNetworkAdapter() to obtain this instance.

5.60.1.3 enabled (read/write)

`boolean INetworkAdapter::enabled`

Flag whether the network adapter is present in the guest system. If disabled, the virtual guest hardware will not contain this network adapter. Can only be changed when the VM is not running.

5.60.1.4 MACAddress (read/write)

`wstring INetworkAdapter::MACAddress`

Ethernet MAC address of the adapter, 12 hexadecimal characters. When setting it to `null` or an empty string, VirtualBox will generate a unique MAC address.

5.60.1.5 attachmentType (read-only)

`NetworkAttachmentType INetworkAdapter::attachmentType`

5.60.1.6 hostInterface (read/write)

`wstring INetworkAdapter::hostInterface`

Name of the host network interface the VM is attached to.

5.60.1.7 internalNetwork (read/write)

`wstring INetworkAdapter::internalNetwork`

Name of the internal network the VM is attached to.

5.60.1.8 NATNetwork (read/write)

`wstring INetworkAdapter::NATNetwork`

Name of the NAT network the VM is attached to.

5.60.1.9 VDENetwork (read/write)

`wstring INetworkAdapter::VDENetwork`

Name of the VDE switch the VM is attached to.

5.60.1.10 cableConnected (read/write)

```
boolean INetworkAdapter::cableConnected
```

Flag whether the adapter reports the cable as connected or not. It can be used to report offline situations to a VM.

5.60.1.11 lineSpeed (read/write)

```
unsigned long INetworkAdapter::lineSpeed
```

Line speed reported by custom drivers, in units of 1 kbps.

5.60.1.12 traceEnabled (read/write)

```
boolean INetworkAdapter::traceEnabled
```

Flag whether network traffic from/to the network card should be traced. Can only be toggled when the VM is turned off.

5.60.1.13 traceFile (read/write)

```
wstring INetworkAdapter::traceFile
```

Filename where a network trace will be stored. If not set, VBox-pid.pcap will be used.

5.60.1.14 natDriver (read-only)

```
INATEngine INetworkAdapter::natDriver
```

Points to the NAT engine which handles the network address translation for this interface. This is active only when the interface actually uses NAT (see attachToNAT()).

5.60.1.15 bootPriority (read/write)

```
unsigned long INetworkAdapter::bootPriority
```

Network boot priority of the adapter. Priority 1 is highest. If not set, the priority is considered to be at the lowest possible setting.

5.60.1.16 bandwidthLimit (read/write)

```
unsigned long INetworkAdapter::bandwidthLimit
```

Maximum throughput allowed for this network adapter, in units of 1 mbps. A zero value means uncapped/unlimited.

5.60.2 attachToBridgedInterface

```
void INetworkAdapter::attachToBridgedInterface()
```

Attach the network adapter to a bridged host interface.

5.60.3 attachToHostOnlyInterface

```
void INetworkAdapter::attachToHostOnlyInterface()
```

Attach the network adapter to the host-only network.

5.60.4 attachToInternalNetwork

```
void INetworkAdapter::attachToInternalNetwork()
```

Attach the network adapter to an internal network.

5.60.5 attachToNAT

```
void INetworkAdapter::attachToNAT()
```

Attach the network adapter to the Network Address Translation (NAT) interface.

5.60.6 attachToVDE

```
void INetworkAdapter::attachToVDE()
```

Attach the network adapter to a VDE network.

5.60.7 detach

```
void INetworkAdapter::detach()
```

Detach the network adapter

5.61 INetworkAdapterChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a property of one of the virtual network adapters changes. Interested callees should use INetworkAdapter methods and attributes to find out what has changed.

5.61.1 Attributes

5.61.1.1 networkAdapter (read-only)

```
INetworkAdapter INetworkAdapterChangedEvent::networkAdapter
```

Network adapter that is subject to change.

5.62 IParallelPort

The IParallelPort interface represents the virtual parallel port device.

The virtual parallel port device acts like an ordinary parallel port inside the virtual machine. This device communicates to the real parallel port hardware using the name of the parallel device on the host computer specified in the #path attribute.

Each virtual parallel port device is assigned a base I/O address and an IRQ number that will be reported to the guest operating system and used to operate the given parallel port from within the virtual machine.

See also: IMachine::getParallelPort

5.62.1 Attributes

5.62.1.1 slot (read-only)

`unsigned long IParallelPort::slot`

Slot number this parallel port is plugged into. Corresponds to the value you pass to IMachine::getParallelPort() to obtain this instance.

5.62.1.2 enabled (read/write)

`boolean IParallelPort::enabled`

Flag whether the parallel port is enabled. If disabled, the parallel port will not be reported to the guest OS.

5.62.1.3 IOBase (read/write)

`unsigned long IParallelPort::IOBase`

Base I/O address of the parallel port.

5.62.1.4 IRQ (read/write)

`unsigned long IParallelPort::IRQ`

IRQ number of the parallel port.

5.62.1.5 path (read/write)

`wstring IParallelPort::path`

Host parallel device name. If this parallel port is enabled, setting a `null` or an empty string as this attribute's value will result into an error.

5.63 IParallelPortChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a property of one of the virtual parallel ports changes. Interested callees should use ISerialPort methods and attributes to find out what has changed.

5.63.1 Attributes

5.63.1.1 parallelPort (read-only)

`IParallelPort IParallelPortChangedEvent::parallelPort`

Parallel port that is subject to change.

5.64 IPciAddress

> **Note:** With the web service, this interface is mapped to a structure. Attributes that return this interface will not return an object, but a complete structure containing the attributes listed below as structure members.

Address on the PCI bus.

5.64.1 Attributes

5.64.1.1 bus (read/write)

```
short IPciAddress::bus
```

Bus number.

5.64.1.2 device (read/write)

```
short IPciAddress::device
```

Device number.

5.64.1.3 devFunction (read/write)

```
short IPciAddress::devFunction
```

Device function number.

5.64.2 asLong

```
long IPciAddress::asLong()
```

Convert PCI address into long.

5.64.3 fromLong

```
void IPciAddress::fromLong(
            [in] long number)
```

number

Make PCI address from long.

5.65 IPciDeviceAttachment

> **Note:** With the web service, this interface is mapped to a structure. Attributes that return this interface will not return an object, but a complete structure containing the attributes listed below as structure members.

Information about PCI attachments.

5.65.1 Attributes

5.65.1.1 name (read-only)

```
wstring IPciDeviceAttachment::name
```

Device name.

5.65.1.2 isPhysicalDevice (read-only)

```
boolean IPciDeviceAttachment::isPhysicalDevice
```

If this is physical or virtual device.

5.65.1.3 hostAddress (read-only)

```
long IPciDeviceAttachment::hostAddress
```

Address of device on the host, applicable only to host devices.

5.65.1.4 guestAddress (read-only)

```
long IPciDeviceAttachment::guestAddress
```

Address of device on the guest.

5.66 IPerformanceCollector

The IPerformanceCollector interface represents a service that collects and stores performance metrics data.

Performance metrics are associated with objects of interfaces like IHost and IMachine. Each object has a distinct set of performance metrics. The set can be obtained with getMetrics().

Metric data is collected at the specified intervals and is retained internally. The interval and the number of retained samples can be set with setupMetrics(). Both metric data and collection settings are not persistent, they are discarded as soon as VBoxSVC process terminates. Moreover, metric settings and data associated with a particular VM only exist while VM is running. They disappear as soon as VM shuts down. It is not possible to set up metrics for machines that are powered off. One needs to start VM first, then set up metric collection parameters.

Metrics are organized hierarchically, with each level separated by a slash (/) character. Generally, the scheme for metric names is like this:

`Category/Metric[/SubMetric][:aggregation]`

"Category/Metric" together form the base metric name. A base metric is the smallest unit for which a sampling interval and the number of retained samples can be set. Only base metrics can be enabled and disabled. All sub-metrics are collected when their base metric is collected. Collected values for any set of sub-metrics can be queried with queryMetricsData().

For example "CPU/Load/User:avg" metric name stands for the "CPU" category, "Load" metric, "User" submetric, "average" aggregate. An aggregate function is computed over all retained data. Valid aggregate functions are:

- avg – average
- min – minimum
- max – maximum

When setting up metric parameters, querying metric data, enabling or disabling metrics wildcards can be used in metric names to specify a subset of metrics. For example, to select all CPU-related metrics use `CPU/*`, all averages can be queried using `*:avg` and so on. To query metric values without aggregates `*:` can be used.

The valid names for base metrics are:

- CPU/Load
- CPU/MHz
- RAM/Usage

The general sequence for collecting and retrieving the metrics is:

- Obtain an instance of IPerformanceCollector with IVirtualBox::performanceCollector
- Allocate and populate an array with references to objects the metrics will be collected for. Use references to IHost and IMachine objects.
- Allocate and populate an array with base metric names the data will be collected for.
- Call setupMetrics(). From now on the metric data will be collected and stored.
- Wait for the data to get collected.
- Allocate and populate an array with references to objects the metric values will be queried for. You can re-use the object array used for setting base metrics.
- Allocate and populate an array with metric names the data will be collected for. Note that metric names differ from base metric names.
- Call queryMetricsData(). The data that have been collected so far are returned. Note that the values are still retained internally and data collection continues.

For an example of usage refer to the following files in VirtualBox SDK:

- Java: `bindings/webservice/java/jax-ws/samples/metrictest.java`
- Python: `bindings/xpcom/python/sample/shellcommon.py`

5.66.1 Attributes

5.66.1.1 metricNames (read-only)

```
wstring IPerformanceCollector::metricNames[]
```

Array of unique names of metrics.

This array represents all metrics supported by the performance collector. Individual objects do not necessarily support all of them. getMetrics() can be used to get the list of supported metrics for a particular object.

5.66.2 disableMetrics

```
IPerformanceMetric[] IPerformanceCollector::disableMetrics(
            [in] wstring metricNames[],
            [in] $unknown objects[])
```

metricNames Metric name filter. Comma-separated list of metrics with wildcard support.

objects Set of objects to disable metrics for.

Turns off collecting specified base metrics. Returns an array of IPerformanceMetric describing the metrics have been affected.

> **Note:** `Null` or empty metric name array means all metrics. `Null` or empty object array means all existing objects. If metric name array contains a single element and object array contains many, the single metric name array element is applied to each object array element to form metric/object pairs.

5.66.3 enableMetrics

```
IPerformanceMetric[] IPerformanceCollector::enableMetrics(
            [in] wstring metricNames[],
            [in] $unknown objects[])
```

metricNames Metric name filter. Comma-separated list of metrics with wildcard support.

objects Set of objects to enable metrics for.

Turns on collecting specified base metrics. Returns an array of IPerformanceMetric describing the metrics have been affected.

> **Note:** `Null` or empty metric name array means all metrics. `Null` or empty object array means all existing objects. If metric name array contains a single element and object array contains many, the single metric name array element is applied to each object array element to form metric/object pairs.

5.66.4 getMetrics

```
IPerformanceMetric[] IPerformanceCollector::getMetrics(
            [in] wstring metricNames[],
            [in] $unknown objects[])
```

metricNames Metric name filter. Currently, only a comma-separated list of metrics is supported.

objects Set of objects to return metric parameters for.

Returns parameters of specified metrics for a set of objects.

> **Note:** `Null` metrics array means all metrics. `Null` object array means all existing objects.

5.66.5 queryMetricsData

```
long[] IPerformanceCollector::queryMetricsData(
            [in] wstring metricNames[],
            [in] $unknown objects[],
            [out] wstring returnMetricNames[],
            [out] $unknown returnObjects[],
            [out] wstring returnUnits[],
            [out] unsigned long returnScales[],
            [out] unsigned long returnSequenceNumbers[],
            [out] unsigned long returnDataIndices[],
            [out] unsigned long returnDataLengths[])
```

metricNames Metric name filter. Comma-separated list of metrics with wildcard support.

objects Set of objects to query metrics for.

returnMetricNames Names of metrics returned in `returnData`.

returnObjects Objects associated with metrics returned in `returnData`.

returnUnits Units of measurement for each returned metric.

returnScales Divisor that should be applied to return values in order to get floating point values. For example: `(double)returnData[returnDataIndices[0]+i] / returnScales[0]` will retrieve the floating point value of i-th sample of the first metric.

returnSequenceNumbers Sequence numbers of the first elements of value sequences of particular metrics returned in `returnData`. For aggregate metrics it is the sequence number of the sample the aggregate started calculation from.

returnDataIndices Indices of the first elements of value sequences of particular metrics returned in `returnData`.

returnDataLengths Lengths of value sequences of particular metrics.

Queries collected metrics data for a set of objects.

The data itself and related metric information are returned in seven parallel and one flattened array of arrays. Elements of `returnMetricNames, returnObjects, returnUnits, returnScales, returnSequenceNumbers, returnDataIndices` and `returnDataLengths` with the same index describe one set of values corresponding to a single metric.

The `returnData` parameter is a flattened array of arrays. Each start and length of a subarray is indicated by `returnDataIndices` and `returnDataLengths`. The first value for metric `metricNames[i]` is at `returnData[returnIndices[i]]`.

> **Note:** `Null` or empty metric name array means all metrics. `Null` or empty object array means all existing objects. If metric name array contains a single element and object array contains many, the single metric name array element is applied to each object array element to form metric/object pairs.

> **Note:** Data collection continues behind the scenes after call to @c queryMetricsData. The return data can be seen as the snapshot of the current state at the time of `queryMetricsData` call. The internally kept metric values are not cleared by the call. This makes possible querying different subsets of metrics or aggregates with subsequent calls. If periodic querying is needed it is highly suggested to query the values with `interval`*count period to avoid confusion. This way a completely new set of data values will be provided by each query.

5.66.6 setupMetrics

```
IPerformanceMetric[] IPerformanceCollector::setupMetrics(
            [in] wstring metricNames[],
            [in] $unknown objects[],
            [in] unsigned long period,
            [in] unsigned long count)
```

metricNames Metric name filter. Comma-separated list of metrics with wildcard support.

objects Set of objects to setup metric parameters for.

period Time interval in seconds between two consecutive samples of performance data.

count Number of samples to retain in performance data history. Older samples get discarded.

Sets parameters of specified base metrics for a set of objects. Returns an array of IPerformanceMetric describing the metrics have been affected.

> **Note:** `Null` or empty metric name array means all metrics. `Null` or empty object array means all existing objects. If metric name array contains a single element and object array contains many, the single metric name array element is applied to each object array element to form metric/object pairs.

5.67 IPerformanceMetric

The IPerformanceMetric interface represents parameters of the given performance metric.

5.67.1 Attributes

5.67.1.1 metricName (read-only)

```
wstring IPerformanceMetric::metricName
```

Name of the metric.

5.67.1.2 object (read-only)

```
$unknown IPerformanceMetric::object
```

Object this metric belongs to.

5.67.1.3 description (read-only)

```
wstring IPerformanceMetric::description
```

Textual description of the metric.

5.67.1.4 period (read-only)

```
unsigned long IPerformanceMetric::period
```

Time interval between samples, measured in seconds.

5.67.1.5 count (read-only)

```
unsigned long IPerformanceMetric::count
```

Number of recent samples retained by the performance collector for this metric.
When the collected sample count exceeds this number, older samples are discarded.

5.67.1.6 unit (read-only)

```
wstring IPerformanceMetric::unit
```

Unit of measurement.

5.67.1.7 minimumValue (read-only)

```
long IPerformanceMetric::minimumValue
```

Minimum possible value of this metric.

5.67.1.8 maximumValue (read-only)

```
long IPerformanceMetric::maximumValue
```

Maximum possible value of this metric.

5.68 IProgress

The IProgress interface is used to track and control asynchronous tasks within VirtualBox.

An instance of this is returned every time VirtualBox starts an asynchronous task (in other words, a separate thread) which continues to run after a method call returns. For example, IConsole::saveState(), which saves the state of a running virtual machine, can take a long time to complete. To be able to display a progress bar, a user interface such as the VirtualBox graphical user interface can use the IProgress object returned by that method.

Note that IProgress is a "read-only" interface in the sense that only the VirtualBox internals behind the Main API can create and manipulate progress objects, whereas client code can only use the IProgress object to monitor a task's progress and, if cancelable is `true`, cancel the task by calling cancel().

A task represented by IProgress consists of either one or several sub-operations that run sequentially, one by one (see operation and operationCount). Every operation is identified by a number (starting from 0) and has a separate description.

You can find the individual percentage of completion of the current operation in operationPercent and the percentage of completion of the task as a whole in percent.

Similarly, you can wait for the completion of a particular operation via waitForOperationCompletion() or for the completion of the whole task via waitForCompletion().

5.68.1 Attributes

5.68.1.1 id (read-only)

```
uuid IProgress::id
```

ID of the task.

5.68.1.2 description (read-only)

```
wstring IProgress::description
```

Description of the task.

5.68.1.3 initiator (read-only)

```
$unknown IProgress::initiator
```

Initiator of the task.

5.68.1.4 cancelable (read-only)

```
boolean IProgress::cancelable
```

Whether the task can be interrupted.

5.68.1.5 percent (read-only)

```
unsigned long IProgress::percent
```

Current progress value of the task as a whole, in percent. This value depends on how many operations are already complete. Returns 100 if completed is `true`.

5.68.1.6 timeRemaining (read-only)

```
long IProgress::timeRemaining
```

Estimated remaining time until the task completes, in seconds. Returns 0 once the task has completed; returns -1 if the remaining time cannot be computed, in particular if the current progress is 0.

Even if a value is returned, the estimate will be unreliable for low progress values. It will become more reliable as the task progresses; it is not recommended to display an ETA before at least 20% of a task have completed.

5.68.1.7 completed (read-only)

```
boolean IProgress::completed
```

Whether the task has been completed.

5.68.1.8 canceled (read-only)

```
boolean IProgress::canceled
```

Whether the task has been canceled.

5.68.1.9 resultCode (read-only)

```
long IProgress::resultCode
```

Result code of the progress task. Valid only if completed is `true`.

5.68.1.10 errorInfo (read-only)

```
IVirtualBoxErrorInfo IProgress::errorInfo
```

Extended information about the unsuccessful result of the progress operation. May be `null` if no extended information is available. Valid only if completed is `true` and resultCode indicates a failure.

5.68.1.11 operationCount (read-only)

```
unsigned long IProgress::operationCount
```

Number of sub-operations this task is divided into. Every task consists of at least one suboperation.

5.68.1.12 operation (read-only)

```
unsigned long IProgress::operation
```

Number of the sub-operation being currently executed.

5.68.1.13 operationDescription (read-only)

```
wstring IProgress::operationDescription
```

Description of the sub-operation being currently executed.

5.68.1.14 operationPercent (read-only)

```
unsigned long IProgress::operationPercent
```

Progress value of the current sub-operation only, in percent.

5.68.1.15 operationWeight (read-only)

```
unsigned long IProgress::operationWeight
```

Weight value of the current sub-operation only.

5.68.1.16 timeout (read/write)

```
unsigned long IProgress::timeout
```

When non-zero, this specifies the number of milliseconds after which the operation will automatically be canceled. This can only be set on cancelable objects.

5.68.2 cancel

```
void IProgress::cancel()
```

Cancels the task.

> **Note:** If cancelable is `false`, then this method will fail.

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_OBJECT_STATE`: Operation cannot be canceled.

5.68.3 setCurrentOperationProgress

```
void IProgress::setCurrentOperationProgress(
           [in] unsigned long percent)
```

percent

Internal method, not to be called externally.

5.68.4 setNextOperation

```
void IProgress::setNextOperation(
           [in] wstring nextOperationDescription,
           [in] unsigned long nextOperationsWeight)
```

nextOperationDescription

nextOperationsWeight

Internal method, not to be called externally.

5.68.5 waitForCompletion

```
void IProgress::waitForCompletion(
           [in] long timeout)
```

timeout Maximum time in milliseconds to wait or -1 to wait indefinitely.

Waits until the task is done (including all sub-operations) with a given timeout in milliseconds; specify -1 for an indefinite wait.

Note that the VirtualBox/XPCOM/COM/native event queues of the calling thread are not processed while waiting. Neglecting event queues may have dire consequences (degrade performance, resource hogs, deadlocks, etc.), this is specially so for the main thread on platforms using XPCOM. Callers are adviced wait for short periods and service their event queues between calls, or to create a worker thread to do the waiting.

If this method fails, the following error codes may be reported:

- `VBOX_E_IPRT_ERROR:` Failed to wait for task completion.

5.68.6 waitForOperationCompletion

```
void IProgress::waitForOperationCompletion(
           [in] unsigned long operation,
           [in] long timeout)
```

operation Number of the operation to wait for. Must be less than operationCount.

timeout Maximum time in milliseconds to wait or -1 to wait indefinitely.

Waits until the given operation is done with a given timeout in milliseconds; specify -1 for an indefinite wait.

See for event queue considerations.

If this method fails, the following error codes may be reported:

- `VBOX_E_IPRT_ERROR:` Failed to wait for operation completion.

5.69 IReusableEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Base abstract interface for all reusable events.

5.69.1 Attributes

5.69.1.1 generation (read-only)

`unsigned long IReusableEvent::generation`

Current generation of event, incremented on reuse.

5.69.2 reuse

`void IReusableEvent::reuse()`

Marks an event as reused, increments 'generation', fields shall no longer be considered valid.

5.70 IRuntimeErrorEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when an error happens during the virtual machine execution.
There are three kinds of runtime errors:

- *fatal*
- *non-fatal with retry*
- *non-fatal warnings*

Fatal errors are indicated by the `fatal` parameter set to `true`. In case of fatal errors, the virtual machine execution is always paused before calling this notification, and the notification handler is supposed either to immediately save the virtual machine state using IConsole::saveState() or power it off using IConsole::powerDown(). Resuming the execution can lead to unpredictable results.

Non-fatal errors and warnings are indicated by the `fatal` parameter set to `false`. If the virtual machine is in the Paused state by the time the error notification is received, it means that the user can *try to resume* the machine execution after attempting to solve the problem that caused the error. In this case, the notification handler is supposed to show an appropriate message to the user (depending on the value of the `id` parameter) that offers several actions such as *Retry*, *Save* or *Power Off*. If the user wants to retry, the notification handler should continue the machine execution using the IConsole::resume() call. If the machine execution is not Paused during this notification, then it means this notification is a *warning* (for example, about a fatal condition that can happen very soon); no immediate action is required from the user, the machine continues its normal execution.

Note that in either case the notification handler **must not** perform any action directly on a thread where this notification is called. Everything it is allowed to do is to post a message to another thread that will then talk to the user and take the corresponding action.

Currently, the following error identifiers are known:

- `"HostMemoryLow"`
- `"HostAudioNotResponding"`
- `"VDIStorageFull"`
- `"3DSupportIncompatibleAdditions"`

5.70.1 Attributes

5.70.1.1 fatal (read-only)

```
boolean IRuntimeErrorEvent::fatal
```

Whether the error is fatal or not.

5.70.1.2 id (read-only)

```
wstring IRuntimeErrorEvent::id
```

Error identifier.

5.70.1.3 message (read-only)

```
wstring IRuntimeErrorEvent::message
```

Optional error message.

5.71 ISerialPort

The ISerialPort interface represents the virtual serial port device.

The virtual serial port device acts like an ordinary serial port inside the virtual machine. This device communicates to the real serial port hardware in one of two modes: host pipe or host device.

In host pipe mode, the #path attribute specifies the path to the pipe on the host computer that represents a serial port. The #server attribute determines if this pipe is created by the virtual machine process at machine startup or it must already exist before starting machine execution.

In host device mode, the #path attribute specifies the name of the serial port device on the host computer.

There is also a third communication mode: the disconnected mode. In this mode, the guest OS running inside the virtual machine will be able to detect the serial port, but all port write operations will be discarded and all port read operations will return no data.

See also: IMachine::getSerialPort

5.71.1 Attributes

5.71.1.1 slot (read-only)

```
unsigned long ISerialPort::slot
```

Slot number this serial port is plugged into. Corresponds to the value you pass to IMachine::getSerialPort() to obtain this instance.

5.71.1.2 enabled (read/write)

```
boolean ISerialPort::enabled
```

Flag whether the serial port is enabled. If disabled, the serial port will not be reported to the guest OS.

5.71.1.3 IOBase (read/write)

```
unsigned long ISerialPort::IOBase
```

Base I/O address of the serial port.

5.71.1.4 IRQ (read/write)

```
unsigned long ISerialPort::IRQ
```

IRQ number of the serial port.

5.71.1.5 hostMode (read/write)

```
PortMode ISerialPort::hostMode
```

How is this port connected to the host.

> **Note:** Changing this attribute may fail if the conditions for path are not met.

5.71.1.6 server (read/write)

```
boolean ISerialPort::server
```

Flag whether this serial port acts as a server (creates a new pipe on the host) or as a client (uses the existing pipe). This attribute is used only when hostMode is PortMode_HostPipe.

5.71.1.7 path (read/write)

```
wstring ISerialPort::path
```

Path to the serial port's pipe on the host when hostMode is PortMode_HostPipe, or the host serial device name when hostMode is PortMode_HostDevice. For both cases, setting a `null` or empty string as the attribute's value is an error. Otherwise, the value of this property is ignored.

5.72 ISerialPortChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a property of one of the virtual serial ports changes. Interested callees should use ISerialPort methods and attributes to find out what has changed.

5.72.1 Attributes

5.72.1.1 serialPort (read-only)

```
ISerialPort ISerialPortChangedEvent::serialPort
```

Serial port that is subject to change.

5.73 ISession

The ISession interface represents a client process and allows for locking virtual machines (represented by IMachine objects) to prevent conflicting changes to the machine.

Any caller wishing to manipulate a virtual machine needs to create a session object first, which lives in its own process space. Such session objects are then associated with IMachine objects living in the VirtualBox server process to coordinate such changes.

There are two typical scenarios in which sessions are used:

- To alter machine settings or control a running virtual machine, one needs to lock a machine for a given session (client process) by calling IMachine::lockMachine().

  Whereas multiple sessions may control a running virtual machine, only one process can obtain a write lock on the machine to prevent conflicting changes. A write lock is also needed if a process wants to actually run a virtual machine in its own context, such as the VirtualBox GUI or VBoxHeadless front-ends. They must also lock a machine for their own sessions before they are allowed to power up the virtual machine.

  As a result, no machine settings can be altered while another process is already using it, either because that process is modifying machine settings or because the machine is running.

- To start a VM using one of the existing VirtualBox front-ends (e.g. the VirtualBox GUI or VBoxHeadless), one would use IMachine::launchVMProcess(), which also takes a session object as its first parameter. This session then identifies the caller and lets the caller control the started machine (for example, pause machine execution or power it down) as well as be notified about machine execution state changes.

How sessions objects are created in a client process depends on whether you use the Main API via COM or via the webservice:

- When using the COM API directly, an object of the Session class from the VirtualBox type library needs to be created. In regular COM C++ client code, this can be done by calling `createLocalObject()`, a standard COM API. This object will then act as a local session object in further calls to open a session.

- In the webservice, the session manager (IWebsessionManager) instead creates a session object automatically whenever IWebsessionManager::logon() is called. A managed object reference to that session object can be retrieved by calling IWebsessionManager::getSessionObject().

5.73.1 Attributes

5.73.1.1 state (read-only)

`SessionState ISession::state`

Current state of this session.

5.73.1.2 type (read-only)

`SessionType ISession::type`

Type of this session. The value of this attribute is valid only if the session currently has a machine locked (i.e. its state is Locked), otherwise an error will be returned.

5.73.1.3 machine (read-only)

`IMachine ISession::machine`

Machine object associated with this session.

5.73.1.4 console (read-only)

`IConsole ISession::console`

Console object associated with this session.

5.73.2 unlockMachine

`void ISession::unlockMachine()`

Unlocks a machine that was previously locked for the current session.

Calling this method is required every time a machine has been locked for a particular session using the IMachine::launchVMProcess() or IMachine::lockMachine() calls. Otherwise the state of the machine will be set to Aborted on the server, and changes made to the machine settings will be lost.

Generally, it is recommended to unlock all machines explicitly before terminating the application (regardless of the reason for the termination).

> **Note:** Do not expect the session state (state to return to "Unlocked" immediately after you invoke this method, particularly if you have started a new VM process. The session state will automatically return to "Unlocked" once the VM is no longer executing, which can of course take a very long time.

If this method fails, the following error codes may be reported:

- `E_UNEXPECTED:` Session is not locked.

5.74 ISessionStateChangedEvent (IMachineEvent)

> **Note:** This interface extends IMachineEvent and therefore supports all its methods and attributes as well.

The state of the session for the given machine was changed. See also: IMachine::sessionState

5.74.1 Attributes

5.74.1.1 state (read-only)

`SessionState ISessionStateChangedEvent::state`

New session state.

5.75 ISharedFolder

> **Note:** With the web service, this interface is mapped to a structure. Attributes that return this interface will not return an object, but a complete structure containing the attributes listed below as structure members.

The ISharedFolder interface represents a folder in the host computer's file system accessible from the guest OS running inside a virtual machine using an associated logical name.

There are three types of shared folders:

- *Global* (IVirtualBox::sharedFolders[]), shared folders available to all virtual machines.
- *Permanent* (IMachine::sharedFolders[]), VM-specific shared folders available to the given virtual machine at startup.
- *Transient* (IConsole::sharedFolders[]), VM-specific shared folders created in the session context (for example, when the virtual machine is running) and automatically discarded when the session is closed (the VM is powered off).

Logical names of shared folders must be unique within the given scope (global, permanent or transient). However, they do not need to be unique across scopes. In this case, the definition of the shared folder in a more specific scope takes precedence over definitions in all other scopes. The order of precedence is (more specific to more general):

1. Transient definitions
2. Permanent definitions
3. Global definitions

For example, if MyMachine has a shared folder named `C_DRIVE` (that points to `C:\\`), then creating a transient shared folder named `C_DRIVE` (that points to `C:\\\\WINDOWS`) will change the definition of `C_DRIVE` in the guest OS so that `\\\\VBOXSVR\\C_DRIVE` will give access to `C:\\WINDOWS` instead of `C:\\` on the host PC. Removing the transient shared folder `C_DRIVE` will restore the previous (permanent) definition of `C_DRIVE` that points to `C:\\` if it still exists.

Note that permanent and transient shared folders of different machines are in different name spaces, so they don't overlap and don't need to have unique logical names.

Note: Global shared folders are not implemented in the current version of the product.

5.75.1 Attributes

5.75.1.1 name (read-only)

```
wstring ISharedFolder::name
```

Logical name of the shared folder.

5.75.1.2 hostPath (read-only)

```
wstring ISharedFolder::hostPath
```

Full path to the shared folder in the host file system.

5.75.1.3 accessible (read-only)

```
boolean ISharedFolder::accessible
```

Whether the folder defined by the host path is currently accessible or not. For example, the folder can be inaccessible if it is placed on the network share that is not available by the time this property is read.

5.75.1.4 writable (read-only)

```
boolean ISharedFolder::writable
```

Whether the folder defined by the host path is writable or not.

5.75.1.5 autoMount (read-only)

`boolean ISharedFolder::autoMount`

Whether the folder gets automatically mounted by the guest or not.

5.75.1.6 lastAccessError (read-only)

`wstring ISharedFolder::lastAccessError`

Text message that represents the result of the last accessibility check.

Accessibility checks are performed each time the accessible attribute is read. An empty string is returned if the last accessibility check was successful. A non-empty string indicates a failure and should normally describe a reason of the failure (for example, a file read error).

5.76 ISharedFolderChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a shared folder is added or removed. The `scope` argument defines one of three scopes: global shared folders (Global), permanent shared folders of the machine (Machine) or transient shared folders of the machine (Session). Interested callees should use query the corresponding collections to find out what has changed.

5.76.1 Attributes

5.76.1.1 scope (read-only)

`Scope ISharedFolderChangedEvent::scope`

Scope of the notification.

5.77 IShowWindowEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a call to IMachine::showConsoleWindow() requests the console window to be activated and brought to foreground on the desktop of the host PC.

This notification should cause the VM console process to perform the requested action as described above. If it is impossible to do it at a time of this notification, this method should return a failure.

Note that many modern window managers on many platforms implement some sort of focus stealing prevention logic, so that it may be impossible to activate a window without the help of the currently active application (which is supposedly an initiator of this notification). In this case, this method must return a non-zero identifier that represents the top-level window of the VM console process. The caller, if it represents a currently active process, is responsible to use this identifier (in a platform-dependent manner) to perform actual window activation.

This method must set `winId` to zero if it has performed all actions necessary to complete the request and the console window is now active and in foreground, to indicate that no further action is required on the caller's side.

5.77.1 Attributes

5.77.1.1 winId (read/write)

```
long long IShowWindowEvent::winId
```

Platform-dependent identifier of the top-level VM console window, or zero if this method has performed all actions necessary to implement the *show window* semantics for the given platform and/or this VirtualBox front-end.

5.78 ISnapshot

The ISnapshot interface represents a snapshot of the virtual machine.

Together with the differencing media that are created when a snapshot is taken, a machine can be brought back to the exact state it was in when the snapshot was taken.

The ISnapshot interface has no methods, only attributes; snapshots are controlled through methods of the IConsole interface which also manage the media associated with the snapshot. The following operations exist:

- IConsole::takeSnapshot() creates a new snapshot by creating new, empty differencing images for the machine's media and saving the VM settings and (if the VM is running) the current VM state in the snapshot.

  The differencing images will then receive all data written to the machine's media, while their parent (base) images remain unmodified after the snapshot has been taken (see IMedium for details about differencing images). This simplifies restoring a machine to the state of a snapshot: only the differencing images need to be deleted.

  The current machine state is not changed by taking a snapshot except that IMachine::currentSnapshot is set to the newly created snapshot, which is also added to the machine's snapshots tree.

- IConsole::restoreSnapshot() resets a machine to the state of a previous snapshot by deleting the differencing image of each of the machine's media and setting the machine's settings and state to the state that was saved in the snapshot (if any).

  This destroys the machine's current state. After calling this, IMachine::currentSnapshot points to the snapshot that was restored.

- IConsole::deleteSnapshot() deletes a snapshot without affecting the current machine state.

  This does not change the current machine state, but instead frees the resources allocated when the snapshot was taken: the settings and machine state file are deleted (if any), and the snapshot's differencing image for each of the machine's media gets merged with its parent image.

  Neither the current machine state nor other snapshots are affected by this operation, except that parent media will be modified to contain the disk data associated with the snapshot being deleted.

  When deleting the current snapshot, the IMachine::currentSnapshot attribute is set to the current snapshot's parent or NULL if it has no parent. Otherwise the attribute is unchanged.

Each snapshot contains a copy of virtual machine's settings (hardware configuration etc.). This copy is contained in an immutable (read-only) instance of IMachine which is available from the snapshot's machine attribute. When restoring the snapshot, these settings are copied back to the original machine.

In addition, if the machine was running when the snapshot was taken (IMachine::state is Running), the current VM state is saved in the snapshot (similarly to what happens when a VM's

state is saved). The snapshot is then said to be *online* because when restoring it, the VM will be running.

If the machine was in Saved saved, the snapshot receives a copy of the execution state file (IMachine::stateFilePath).

Otherwise, if the machine was not running (PoweredOff or Aborted), the snapshot is *offline*; it then contains a so-called "zero execution state", representing a machine that is powered off.

5.78.1 Attributes

5.78.1.1 id (read-only)

`uuid ISnapshot::id`

UUID of the snapshot.

5.78.1.2 name (read/write)

`wstring ISnapshot::name`

Short name of the snapshot.

> **Note:** Setting this attribute causes IMachine::saveSettings() to be called implicitly.

5.78.1.3 description (read/write)

`wstring ISnapshot::description`

Optional description of the snapshot.

> **Note:** Setting this attribute causes IMachine::saveSettings() to be called implicitly.

5.78.1.4 timeStamp (read-only)

`long long ISnapshot::timeStamp`

Time stamp of the snapshot, in milliseconds since 1970-01-01 UTC.

5.78.1.5 online (read-only)

`boolean ISnapshot::online`

`true` if this snapshot is an online snapshot and `false` otherwise.

When this attribute is `true`, the IMachine::stateFilePath attribute of the machine object associated with this snapshot will point to the saved state file. Otherwise, it will be an empty string.

5.78.1.6 machine (read-only)

`IMachine ISnapshot::machine`

Virtual machine this snapshot is taken on. This object stores all settings the machine had when taking this snapshot.

> **Note:** The returned machine object is immutable, i.e. no any settings can be changed.

5.78.1.7 parent (read-only)

`ISnapshot ISnapshot::parent`

Parent snapshot (a snapshot this one is based on), or `null` if the snapshot has no parent (i.e. is the first snapshot).

5.78.1.8 children (read-only)

`ISnapshot ISnapshot::children[]`

Child snapshots (all snapshots having this one as a parent). By inspecting this attribute starting with a machine's root snapshot (which can be obtained by calling IMachine::findSnapshot() with a `null` UUID), a machine's snapshots tree can be iterated over.

5.79 ISnapshotChangedEvent (ISnapshotEvent)

> **Note:** This interface extends ISnapshotEvent and therefore supports all its methods and attributes as well.

Snapshot properties (name and/or description) have been changed. See also: ISnapshot

5.80 ISnapshotDeletedEvent (ISnapshotEvent)

> **Note:** This interface extends ISnapshotEvent and therefore supports all its methods and attributes as well.

Snapshot of the given machine has been deleted.

> **Note:** This notification is delivered **after** the snapshot object has been uninitialized on the server (so that any attempt to call its methods will return an error).

See also: ISnapshot

5.81 ISnapshotEvent (IMachineEvent)

> **Note:** This interface extends IMachineEvent and therefore supports all its methods and attributes as well.

Base interface for all snapshot events.

5.81.1 Attributes

5.81.1.1 snapshotId (read-only)

`uuid ISnapshotEvent::snapshotId`

ID of the snapshot this event relates to.

5.82 ISnapshotTakenEvent (ISnapshotEvent)

Note: This interface extends ISnapshotEvent and therefore supports all its methods and attributes as well.

A new snapshot of the machine has been taken. See also: ISnapshot

5.83 IStateChangedEvent (IEvent)

Note: This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when the execution state of the machine has changed. The new state is given.

5.83.1 Attributes

5.83.1.1 state (read-only)

```
MachineState IStateChangedEvent::state
```

New machine state.

5.84 IStorageController

Represents a storage controller that is attached to a virtual machine (IMachine). Just as drives (hard disks, DVDs, FDs) are attached to storage controllers in a real computer, virtual drives (represented by IMediumAttachment) are attached to virtual storage controllers, represented by this interface.

As opposed to physical hardware, VirtualBox has a very generic concept of a storage controller, and for purposes of the Main API, all virtual storage is attached to virtual machines via instances of this interface. There are five types of such virtual storage controllers: IDE, SCSI, SATA, SAS and Floppy (see bus). Depending on which of these four is used, certain sub-types may be available and can be selected in controllerType.

Depending on these settings, the guest operating system might see significantly different virtual hardware.

5.84.1 Attributes

5.84.1.1 name (read-only)

```
wstring IStorageController::name
```

Name of the storage controller, as originally specified with IMachine::addStorageController(). This then uniquely identifies this controller with other method calls such as IMachine::attachDevice() and IMachine::mountMedium().

5.84.1.2 maxDevicesPerPortCount (read-only)

```
unsigned long IStorageController::maxDevicesPerPortCount
```

Maximum number of devices which can be attached to one port.

5.84.1.3 minPortCount (read-only)

```
unsigned long IStorageController::minPortCount
```

Minimum number of ports that portCount can be set to.

5.84.1.4 maxPortCount (read-only)

```
unsigned long IStorageController::maxPortCount
```

Maximum number of ports that portCount can be set to.

5.84.1.5 instance (read/write)

```
unsigned long IStorageController::instance
```

The instance number of the device in the running VM.

5.84.1.6 portCount (read/write)

```
unsigned long IStorageController::portCount
```

The number of currently usable ports on the controller. The minimum and maximum number of ports for one controller are stored in minPortCount and maxPortCount.

5.84.1.7 bus (read-only)

```
StorageBus IStorageController::bus
```

The bus type of the storage controller (IDE, SATA, SCSI, SAS or Floppy).

5.84.1.8 controllerType (read/write)

```
StorageControllerType IStorageController::controllerType
```

The exact variant of storage controller hardware presented to the guest. Depending on this value, VirtualBox will provide a different virtual storage controller hardware to the guest. For SATA, SAS and floppy controllers, only one variant is available, but for IDE and SCSI, there are several.

For SCSI controllers, the default type is LsiLogic.

5.84.1.9 useHostIOCache (read/write)

```
boolean IStorageController::useHostIOCache
```

If true, the storage controller emulation will use a dedicated I/O thread, enable the host I/O caches and use synchronous file APIs on the host. This was the only option in the API before VirtualBox 3.2 and is still the default for IDE controllers.

If false, the host I/O cache will be disabled for image files attached to this storage controller. Instead, the storage controller emulation will use asynchronous I/O APIs on the host. This makes it possible to turn off the host I/O caches because the emulation can handle unaligned access to the file. This should be used on OS X and Linux hosts if a high I/O load is expected or many virtual machines are running at the same time to prevent I/O cache related hangs. This option new with the API of VirtualBox 3.2 and is now the default for non-IDE storage controllers.

5.84.1.10 bootable (read-only)

```
boolean IStorageController::bootable
```

Returns whether it is possible to boot from disks attached to this controller.

5.84.2 getIDEEmulationPort

```
long IStorageController::getIDEEmulationPort(
           [in] long devicePosition)
```

devicePosition

Gets the corresponding port number which is emulated as an IDE device. Works only with SATA controllers.

If this method fails, the following error codes may be reported:

- `E_INVALIDARG`: The `devicePosition` is not in the range 0 to 3.
- `E_NOTIMPL`: The storage controller type is not SATAIntelAhci.

5.84.3 setIDEEmulationPort

```
void IStorageController::setIDEEmulationPort(
           [in] long devicePosition,
           [in] long portNumber)
```

devicePosition

portNumber

Sets the port number which is emulated as an IDE device. Works only with SATA controllers.

If this method fails, the following error codes may be reported:

- `E_INVALIDARG`: The `devicePosition` is not in the range 0 to 3 or the `portNumber` is not in the range 0 to 29.
- `E_NOTIMPL`: The storage controller type is not SATAIntelAhci.

5.85 IStorageControllerChangedEvent (IEvent)

Note: This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a medium attachment changes.

5.86 ISystemProperties

The ISystemProperties interface represents global properties of the given VirtualBox installation.

These properties define limits and default values for various attributes and parameters. Most of the properties are read-only, but some can be changed by a user.

5.86.1 Attributes

5.86.1.1 minGuestRAM (read-only)

```
unsigned long ISystemProperties::minGuestRAM
```

Minimum guest system memory in Megabytes.

5.86.1.2 maxGuestRAM (read-only)

```
unsigned long ISystemProperties::maxGuestRAM
```

Maximum guest system memory in Megabytes.

5.86.1.3 minGuestVRAM (read-only)

```
unsigned long ISystemProperties::minGuestVRAM
```

Minimum guest video memory in Megabytes.

5.86.1.4 maxGuestVRAM (read-only)

```
unsigned long ISystemProperties::maxGuestVRAM
```

Maximum guest video memory in Megabytes.

5.86.1.5 minGuestCPUCount (read-only)

```
unsigned long ISystemProperties::minGuestCPUCount
```

Minimum CPU count.

5.86.1.6 maxGuestCPUCount (read-only)

```
unsigned long ISystemProperties::maxGuestCPUCount
```

Maximum CPU count.

5.86.1.7 maxGuestMonitors (read-only)

```
unsigned long ISystemProperties::maxGuestMonitors
```

Maximum of monitors which could be connected.

5.86.1.8 infoVDSize (read-only)

```
long long ISystemProperties::infoVDSize
```

Maximum size of a virtual disk image in bytes. Informational value, does not reflect the limits of any virtual disk image format.

5.86.1.9 networkAdapterCount (read-only)

```
unsigned long ISystemProperties::networkAdapterCount
```

Number of network adapters associated with every IMachine instance.

5.86.1.10 serialPortCount (read-only)

```
unsigned long ISystemProperties::serialPortCount
```

Number of serial ports associated with every IMachine instance.

5.86.1.11 parallelPortCount (read-only)

```
unsigned long ISystemProperties::parallelPortCount
```

Number of parallel ports associated with every IMachine instance.

5.86.1.12 maxBootPosition (read-only)

```
unsigned long ISystemProperties::maxBootPosition
```

Maximum device position in the boot order. This value corresponds to the total number of devices a machine can boot from, to make it possible to include all possible devices to the boot list. See also: IMachine::setBootOrder()

5.86.1.13 defaultMachineFolder (read/write)

```
wstring ISystemProperties::defaultMachineFolder
```

Full path to the default directory used to create new or open existing machines when a machine settings file name contains no path.

Starting with VirtualBox 4.0, by default, this attribute contains the full path of folder named "VirtualBox VMs" in the user's home directory, which depends on the host platform.

When setting this attribute, a full path must be specified. Setting this property to `null` or an empty string or the special value "Machines" (for compatibility reasons) will restore that default value.

If the folder specified herein does not exist, it will be created automatically as needed.

See also: IVirtualBox::createMachine(), IVirtualBox::openMachine()

5.86.1.14 mediumFormats (read-only)

```
IMediumFormat ISystemProperties::mediumFormats[]
```

List of all medium storage formats supported by this VirtualBox installation.

Keep in mind that the medium format identifier (IMediumFormat::id) used in other API calls like IVirtualBox::createHardDisk() to refer to a particular medium format is a case-insensitive string. This means that, for example, all of the following strings:

```
"VDI"
"vdi"
"VdI"
```

refer to the same medium format.

Note that the virtual medium framework is backend-based, therefore the list of supported formats depends on what backends are currently installed.

See also: IMediumFormat,

5.86.1.15 defaultHardDiskFormat (read/write)

```
wstring ISystemProperties::defaultHardDiskFormat
```

Identifier of the default medium format used by VirtualBox.

The medium format set by this attribute is used by VirtualBox when the medium format was not specified explicitly. One example is IVirtualBox::createHardDisk() with the empty format argument. A more complex example is implicit creation of differencing media when taking a snapshot of a virtual machine: this operation will try to use a format of the parent medium first and if this format does not support differencing media the default format specified by this argument will be used.

The list of supported medium formats may be obtained by the mediumFormats[] call. Note that the default medium format must have a capability to create differencing media; otherwise operations that create media implicitly may fail unexpectedly.

The initial value of this property is `"VDI"` in the current version of the VirtualBox product, but may change in the future.

> **Note:** Setting this property to `null` or empty string will restore the initial value.

See also: mediumFormats[], IMediumFormat::id, IVirtualBox::createHardDisk()

5.86.1.16 freeDiskSpaceWarning (read/write)

```
long long ISystemProperties::freeDiskSpaceWarning
```

Issue a warning if the free disk space is below (or in some disk intensive operation is expected to go below) the given size in bytes.

5.86.1.17 freeDiskSpacePercentWarning (read/write)

```
unsigned long ISystemProperties::freeDiskSpacePercentWarning
```

Issue a warning if the free disk space is below (or in some disk intensive operation is expected to go below) the given percentage.

5.86.1.18 freeDiskSpaceError (read/write)

```
long long ISystemProperties::freeDiskSpaceError
```

Issue an error if the free disk space is below (or in some disk intensive operation is expected to go below) the given size in bytes.

5.86.1.19 freeDiskSpacePercentError (read/write)

```
unsigned long ISystemProperties::freeDiskSpacePercentError
```

Issue an error if the free disk space is below (or in some disk intensive operation is expected to go below) the given percentage.

5.86.1.20 VRDEAuthLibrary (read/write)

```
wstring ISystemProperties::VRDEAuthLibrary
```

Library that provides authentication for Remote Desktop clients. The library is used if a virtual machine's authentication type is set to "external" in the VM RemoteDisplay configuration.

The system library extension (".DLL" or ".so") must be omitted. A full path can be specified; if not, then the library must reside on the system's default library path.

The default value of this property is `"VBoxAuth"`. There is a library of that name in one of the default VirtualBox library directories.

For details about VirtualBox authentication libraries and how to implement them, please refer to the VirtualBox manual.

> **Note:** Setting this property to `null` or empty string will restore the initial value.

5.86.1.21 webServiceAuthLibrary (read/write)

```
wstring ISystemProperties::webServiceAuthLibrary
```

Library that provides authentication for webservice clients. The library is used if a virtual machine's authentication type is set to "external" in the VM RemoteDisplay configuration and will be called from within the IWebsessionManager::logon() implementation.

As opposed to VRDEAuthLibrary, there is no per-VM setting for this, as the webservice is a global resource (if it is running). Only for this setting (for the webservice), setting this value to a literal `"null"` string disables authentication, meaning that IWebsessionManager::logon() will always succeed, no matter what user name and password are supplied.

The initial value of this property is `"VBoxAuth"`, meaning that the webservice will use the same authentication library that is used by default for VRDE (again, see VRDEAuthLibrary). The format and calling convention of authentication libraries is the same for the webservice as it is for VRDE.

> **Note:** Setting this property to `null` or empty string will restore the initial value.

5.86.1.22 defaultVRDEExtPack (read/write)

```
wstring ISystemProperties::defaultVRDEExtPack
```

The name of the extension pack providing the default VRDE.

This attribute is for choosing between multiple extension packs providing VRDE. If only one is installed, it will automatically be the default one. The attribute value can be empty if no VRDE extension pack is installed.

For details about VirtualBox Remote Desktop Extension and how to implement one, please refer to the VirtualBox SDK.

5.86.1.23 LogHistoryCount (read/write)

```
unsigned long ISystemProperties::LogHistoryCount
```

This value specifies how many old release log files are kept.

5.86.1.24 defaultAudioDriver (read-only)

```
AudioDriverType ISystemProperties::defaultAudioDriver
```

This value hold the default audio driver for the current system.

5.86.2 getDefaultIoCacheSettingForStorageController

```
boolean ISystemProperties::getDefaultIoCacheSettingForStorageController(
            [in] StorageControllerType controllerType)
```

controllerType The storage controller to the setting for.

Returns the default I/O cache setting for the given storage controller

5.86.3 getDeviceTypesForStorageBus

```
DeviceType[] ISystemProperties::getDeviceTypesForStorageBus(
            [in] StorageBus bus)
```

bus The storage bus type to get the value for.

Returns list of all the supported device types (DeviceType) for the given type of storage bus.

5.86.4 getMaxDevicesPerPortForStorageBus

```
unsigned long ISystemProperties::getMaxDevicesPerPortForStorageBus(
            [in] StorageBus bus)
```

bus The storage bus type to get the value for.

Returns the maximum number of devices which can be attached to a port for the given storage bus.

5.86.5 getMaxInstancesOfStorageBus

```
unsigned long ISystemProperties::getMaxInstancesOfStorageBus(
            [in] ChipsetType chipset,
            [in] StorageBus bus)
```

chipset The chipset type to get the value for.

bus The storage bus type to get the value for.

Returns the maximum number of storage bus instances which can be configured for each VM. This corresponds to the number of storage controllers one can have. Value may depend on chipset type used.

5.86.6 getMaxPortCountForStorageBus

```
unsigned long ISystemProperties::getMaxPortCountForStorageBus(
            [in] StorageBus bus)
```

bus The storage bus type to get the value for.

Returns the maximum number of ports the given storage bus supports.

5.86.7 getMinPortCountForStorageBus

```
unsigned long ISystemProperties::getMinPortCountForStorageBus(
           [in] StorageBus bus)
```

bus The storage bus type to get the value for.

Returns the minimum number of ports the given storage bus supports.

5.87 IUSBController

5.87.1 Attributes

5.87.1.1 enabled (read/write)

```
boolean IUSBController::enabled
```

Flag whether the USB controller is present in the guest system. If disabled, the virtual guest hardware will not contain any USB controller. Can only be changed when the VM is powered off.

5.87.1.2 enabledEhci (read/write)

```
boolean IUSBController::enabledEhci
```

Flag whether the USB EHCI controller is present in the guest system. If disabled, the virtual guest hardware will not contain a USB EHCI controller. Can only be changed when the VM is powered off.

5.87.1.3 proxyAvailable (read-only)

```
boolean IUSBController::proxyAvailable
```

Flag whether there is an USB proxy available.

5.87.1.4 USBStandard (read-only)

```
unsigned short IUSBController::USBStandard
```

USB standard version which the controller implements. This is a BCD which means that the major version is in the high byte and minor version is in the low byte.

5.87.1.5 deviceFilters (read-only)

```
IUSBDeviceFilter IUSBController::deviceFilters[]
```

List of USB device filters associated with the machine.

If the machine is currently running, these filters are activated every time a new (supported) USB device is attached to the host computer that was not ignored by global filters (IHost::USBDeviceFilters[]).

These filters are also activated when the machine is powered up. They are run against a list of all currently available USB devices (in states Available, Busy, Held) that were not previously ignored by global filters.

If at least one filter matches the USB device in question, this device is automatically captured (attached to) the virtual USB controller of this machine.

See also: IUSBDeviceFilter, ::IUSBController

5.87.2 createDeviceFilter

```
IUSBDeviceFilter IUSBController::createDeviceFilter(
            [in] wstring name)
```

name Filter name. See IUSBDeviceFilter::name for more info.

Creates a new USB device filter. All attributes except the filter name are set to empty (any match), *active* is `false` (the filter is not active).

The created filter can then be added to the list of filters using insertDeviceFilter().

See also: #deviceFilters

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` The virtual machine is not mutable.

5.87.3 insertDeviceFilter

```
void IUSBController::insertDeviceFilter(
            [in] unsigned long position,
            [in] IUSBDeviceFilter filter)
```

position Position to insert the filter to.

filter USB device filter to insert.

Inserts the given USB device to the specified position in the list of filters.

Positions are numbered starting from `0`. If the specified position is equal to or greater than the number of elements in the list, the filter is added to the end of the collection.

> **Note:** Duplicates are not allowed, so an attempt to insert a filter that is already in the collection, will return an error.

See also: #deviceFilters

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Virtual machine is not mutable.
- `E_INVALIDARG:` USB device filter not created within this VirtualBox instance.
- `VBOX_E_INVALID_OBJECT_STATE:` USB device filter already in list.

5.87.4 removeDeviceFilter

```
IUSBDeviceFilter IUSBController::removeDeviceFilter(
            [in] unsigned long position)
```

position Position to remove the filter from.

Removes a USB device filter from the specified position in the list of filters.

Positions are numbered starting from `0`. Specifying a position equal to or greater than the number of elements in the list will produce an error.

See also: #deviceFilters

If this method fails, the following error codes may be reported:

- `VBOX_E_INVALID_VM_STATE:` Virtual machine is not mutable.
- `E_INVALIDARG:` USB device filter list empty or invalid `position`.

5.88 IUSBControllerChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a property of the virtual USB controller changes. Interested callees should use IUSBController methods and attributes to find out what has changed.

5.89 IUSBDevice

The IUSBDevice interface represents a virtual USB device attached to the virtual machine.

A collection of objects implementing this interface is stored in the IConsole::USBDevices[] attribute which lists all USB devices attached to a running virtual machine's USB controller.

5.89.1 Attributes

5.89.1.1 id (read-only)

```
uuid IUSBDevice::id
```

Unique USB device ID. This ID is built from #vendorId, #productId, #revision and #serialNumber.

5.89.1.2 vendorId (read-only)

```
unsigned short IUSBDevice::vendorId
```

Vendor ID.

5.89.1.3 productId (read-only)

```
unsigned short IUSBDevice::productId
```

Product ID.

5.89.1.4 revision (read-only)

```
unsigned short IUSBDevice::revision
```

Product revision number. This is a packed BCD represented as unsigned short. The high byte is the integer part and the low byte is the decimal.

5.89.1.5 manufacturer (read-only)

```
wstring IUSBDevice::manufacturer
```

Manufacturer string.

5.89.1.6 product (read-only)

```
wstring IUSBDevice::product
```

Product string.

5.89.1.7 serialNumber (read-only)

```
wstring IUSBDevice::serialNumber
```

Serial number string.

5.89.1.8 address (read-only)

```
wstring IUSBDevice::address
```

Host specific address of the device.

5.89.1.9 port (read-only)

```
unsigned short IUSBDevice::port
```

Host USB port number the device is physically connected to.

5.89.1.10 version (read-only)

```
unsigned short IUSBDevice::version
```

The major USB version of the device - 1 or 2.

5.89.1.11 portVersion (read-only)

```
unsigned short IUSBDevice::portVersion
```

The major USB version of the host USB port the device is physically connected to - 1 or 2. For devices not connected to anything this will have the same value as the version attribute.

5.89.1.12 remote (read-only)

```
boolean IUSBDevice::remote
```

Whether the device is physically connected to a remote VRDE client or to a local host machine.

5.90 IUSBDeviceFilter

The IUSBDeviceFilter interface represents an USB device filter used to perform actions on a group of USB devices.

This type of filters is used by running virtual machines to automatically capture selected USB devices once they are physically attached to the host computer.

A USB device is matched to the given device filter if and only if all attributes of the device match the corresponding attributes of the filter (that is, attributes are joined together using the logical AND operation). On the other hand, all together, filters in the list of filters carry the semantics of the logical OR operation. So if it is desirable to create a match like “this vendor id OR this product id”, one needs to create two filters and specify “any match” (see below) for unused attributes.

All filter attributes used for matching are strings. Each string is an expression representing a set of values of the corresponding device attribute, that will match the given filter. Currently, the following filtering expressions are supported:

- *Interval filters*. Used to specify valid intervals for integer device attributes (Vendor ID, Product ID and Revision). The format of the string is:

  `int:((m)|([m]-[n]))(,(m)|([m]-[n]))*`

  where `m` and `n` are integer numbers, either in octal (starting from `0`), hexadecimal (starting from `0x`) or decimal (otherwise) form, so that `m < n`. If `m` is omitted before a dash (`-`), the minimum possible integer is assumed; if `n` is omitted after a dash, the maximum possible integer is assumed.

- *Boolean filters*. Used to specify acceptable values for boolean device attributes. The format of the string is:

  `true|false|yes|no|0|1`

- *Exact match*. Used to specify a single value for the given device attribute. Any string that doesn't start with `int:` represents the exact match. String device attributes are compared to this string including case of symbols. Integer attributes are first converted to a string (see individual filter attributes) and then compared ignoring case.

- *Any match*. Any value of the corresponding device attribute will match the given filter. An empty or `null` string is used to construct this type of filtering expressions.

> **Note:** On the Windows host platform, interval filters are not currently available. Also all string filter attributes (manufacturer, product, serialNumber) are ignored, so they behave as *any match* no matter what string expression is specified.

See also: IUSBController::deviceFilters, IHostUSBDeviceFilter

5.90.1 Attributes

5.90.1.1 name (read/write)

```
wstring IUSBDeviceFilter::name
```

Visible name for this filter. This name is used to visually distinguish one filter from another, so it can neither be `null` nor an empty string.

5.90.1.2 active (read/write)

```
boolean IUSBDeviceFilter::active
```

Whether this filter active or has been temporarily disabled.

5.90.1.3 vendorId (read/write)

```
wstring IUSBDeviceFilter::vendorId
```

Vendor ID filter. The string representation for the *exact matching* has the form `XXXX`, where `X` is the hex digit (including leading zeroes).

5.90.1.4 productId (read/write)

```
wstring IUSBDeviceFilter::productId
```

Product ID filter. The string representation for the *exact matching* has the form `XXXX`, where `X` is the hex digit (including leading zeroes).

5.90.1.5 revision (read/write)

```
wstring IUSBDeviceFilter::revision
```

Product revision number filter. The string representation for the *exact matching* has the form `IIFF`, where `I` is the decimal digit of the integer part of the revision, and `F` is the decimal digit of its fractional part (including leading and trailing zeros). Note that for interval filters, it's best to use the hexadecimal form, because the revision is stored as a 16 bit packed BCD value; so the expression `int:0x0100-0x0199` will match any revision from `1.0` to `1.99`.

5.90.1.6 manufacturer (read/write)

```
wstring IUSBDeviceFilter::manufacturer
```

Manufacturer filter.

5.90.1.7 product (read/write)

```
wstring IUSBDeviceFilter::product
```

Product filter.

5.90.1.8 serialNumber (read/write)

```
wstring IUSBDeviceFilter::serialNumber
```

Serial number filter.

5.90.1.9 port (read/write)

```
wstring IUSBDeviceFilter::port
```

Host USB port filter.

5.90.1.10 remote (read/write)

```
wstring IUSBDeviceFilter::remote
```

Remote state filter.

> **Note:** This filter makes sense only for machine USB filters, i.e. it is ignored by IHostUSBDeviceFilter objects.

5.90.1.11 maskedInterfaces (read/write)

```
unsigned long IUSBDeviceFilter::maskedInterfaces
```

This is an advanced option for hiding one or more USB interfaces from the guest. The value is a bit mask where the bits that are set means the corresponding USB interface should be hidden, masked off if you like. This feature only works on Linux hosts.

5.91 IUSBDeviceStateChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a USB device is attached to or detached from the virtual USB controller.

This notification is sent as a result of the indirect request to attach the device because it matches one of the machine USB filters, or as a result of the direct request issued by IConsole::attachUSBDevice() or IConsole::detachUSBDevice().

This notification is sent in case of both a succeeded and a failed request completion. When the request succeeds, the `error` parameter is `null`, and the given device has been already added to (when `attached` is `true`) or removed from (when `attached` is `false`) the collection represented by IConsole::USBDevices[]. On failure, the collection doesn't change and the `error` parameter represents the error message describing the failure.

5.91.1 Attributes

5.91.1.1 device (read-only)

`IUSBDevice IUSBDeviceStateChangedEvent::device`

Device that is subject to state change.

5.91.1.2 attached (read-only)

`boolean IUSBDeviceStateChangedEvent::attached`

`true` if the device was attached and `false` otherwise.

5.91.1.3 error (read-only)

`IVirtualBoxErrorInfo IUSBDeviceStateChangedEvent::error`

`null` on success or an error message object on failure.

5.92 IVBoxSVCAvailabilityChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when VBoxSVC becomes unavailable (due to a crash or similar unexpected circumstances) or available again.

5.92.1 Attributes

5.92.1.1 available (read-only)

`boolean IVBoxSVCAvailabilityChangedEvent::available`

Whether VBoxSVC is available now.

5.93 IVFSExplorer

The VFSExplorer interface unifies access to different file system types. This includes local file systems as well remote file systems like S3. For a list of supported types see VFSType. An instance of this is returned by IAppliance::createVFSExplorer().

5.93.1 Attributes

5.93.1.1 path (read-only)

```
wstring IVFSExplorer::path
```

Returns the current path in the virtual file system.

5.93.1.2 type (read-only)

```
VFSType IVFSExplorer::type
```

Returns the file system type which is currently in use.

5.93.2 cd

```
IProgress IVFSExplorer::cd(
            [in] wstring aDir)
```

aDir The name of the directory to go in.

Change the current directory level.

5.93.3 cdUp

```
IProgress IVFSExplorer::cdUp()
```

Go one directory upwards from the current directory level.

5.93.4 entryList

```
void IVFSExplorer::entryList(
            [out] wstring aNames[],
            [out] unsigned long aTypes[],
            [out] unsigned long aSizes[],
            [out] unsigned long aModes[])
```

aNames The list of names for the entries.

aTypes The list of types for the entries.

aSizes The list of sizes (in bytes) for the entries.

aModes The list of file modes (in octal form) for the entries.

Returns a list of files/directories after a call to update(). The user is responsible for keeping this internal list up do date.

5.93.5 exists

```
wstring[] IVFSExplorer::exists(
            [in] wstring aNames[])
```

aNames The names to check.

Checks if the given file list exists in the current directory level.

5.93.6 remove

```
IProgress IVFSExplorer::remove(
            [in] wstring aNames[])
```

aNames The names to remove.

Deletes the given files in the current directory level.

5.93.7 update

```
IProgress IVFSExplorer::update()
```

Updates the internal list of files/directories from the current directory level. Use entryList() to get the full list after a call to this method.

5.94 IVRDEServer

5.94.1 Attributes

5.94.1.1 enabled (read/write)

```
boolean IVRDEServer::enabled
```

VRDE server status.

5.94.1.2 authType (read/write)

```
AuthType IVRDEServer::authType
```

VRDE authentication method.

5.94.1.3 authTimeout (read/write)

```
unsigned long IVRDEServer::authTimeout
```

Timeout for guest authentication. Milliseconds.

5.94.1.4 allowMultiConnection (read/write)

```
boolean IVRDEServer::allowMultiConnection
```

Flag whether multiple simultaneous connections to the VM are permitted. Note that this will be replaced by a more powerful mechanism in the future.

5.94.1.5 reuseSingleConnection (read/write)

```
boolean IVRDEServer::reuseSingleConnection
```

Flag whether the existing connection must be dropped and a new connection must be established by the VRDE server, when a new client connects in single connection mode.

5.94.1.6 VRDEExtPack (read/write)

```
wstring IVRDEServer::VRDEExtPack
```

The name of Extension Pack providing VRDE for this VM. Overrides ISystemProperties::defaultVRDEExtPack.

5.94.1.7 AuthLibrary (read/write)

```
wstring IVRDEServer::AuthLibrary
```

Library used for authentication of RDP clients by this VM. Overrides ISystemProperties::VRDEAuthLibrary.

5.94.1.8 VRDEProperties (read-only)

```
wstring IVRDEServer::VRDEProperties[]
```

Array of names of properties, which are supported by this VRDE server.

5.94.2 getVRDEProperty

```
wstring IVRDEServer::getVRDEProperty(
            [in] wstring key)
```

key Name of the key to get.

Returns a VRDE specific property string.
If the requested data `key` does not exist, this function will succeed and return an empty string in the `value` argument.

5.94.3 setVRDEProperty

```
void IVRDEServer::setVRDEProperty(
            [in] wstring key,
            [in] wstring value)
```

key Name of the key to set.

value Value to assign to the key.

Sets a VRDE specific property string.
If you pass `null` or empty string as a key `value`, the given `key` will be deleted.

5.95 IVRDEServerChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when a property of the VRDE server changes. Interested callees should use IVRDEServer methods and attributes to find out what has changed.

5.96 IVRDEServerInfo

> **Note:** With the web service, this interface is mapped to a structure. Attributes that return this interface will not return an object, but a complete structure containing the attributes listed below as structure members.

Contains information about the remote desktop (VRDE) server capabilities and status. This is used in the IConsole::VRDEServerInfo attribute.

5.96.1 Attributes

5.96.1.1 active (read-only)

```
boolean IVRDEServerInfo::active
```

Whether the remote desktop connection is active.

5.96.1.2 port (read-only)

```
long IVRDEServerInfo::port
```

VRDE server port number. If this property is equal to `0`, then the VRDE server failed to start, usually because there are no free IP ports to bind to. If this property is equal to `-1`, then the VRDE server has not yet been started.

5.96.1.3 numberOfClients (read-only)

```
unsigned long IVRDEServerInfo::numberOfClients
```

How many times a client connected.

5.96.1.4 beginTime (read-only)

```
long long IVRDEServerInfo::beginTime
```

When the last connection was established, in milliseconds since 1970-01-01 UTC.

5.96.1.5 endTime (read-only)

```
long long IVRDEServerInfo::endTime
```

When the last connection was terminated or the current time, if connection is still active, in milliseconds since 1970-01-01 UTC.

5.96.1.6 bytesSent (read-only)

```
long long IVRDEServerInfo::bytesSent
```

How many bytes were sent in last or current, if still active, connection.

5.96.1.7 bytesSentTotal (read-only)

```
long long IVRDEServerInfo::bytesSentTotal
```

How many bytes were sent in all connections.

5.96.1.8 bytesReceived (read-only)

```
long long IVRDEServerInfo::bytesReceived
```

How many bytes were received in last or current, if still active, connection.

5.96.1.9 bytesReceivedTotal (read-only)

```
long long IVRDEServerInfo::bytesReceivedTotal
```

How many bytes were received in all connections.

5.96.1.10 user (read-only)

```
wstring IVRDEServerInfo::user
```

Login user name supplied by the client.

5.96.1.11 domain (read-only)

```
wstring IVRDEServerInfo::domain
```

Login domain name supplied by the client.

5.96.1.12 clientName (read-only)

```
wstring IVRDEServerInfo::clientName
```

The client name supplied by the client.

5.96.1.13 clientIP (read-only)

```
wstring IVRDEServerInfo::clientIP
```

The IP address of the client.

5.96.1.14 clientVersion (read-only)

```
unsigned long IVRDEServerInfo::clientVersion
```

The client software version number.

5.96.1.15 encryptionStyle (read-only)

```
unsigned long IVRDEServerInfo::encryptionStyle
```

Public key exchange method used when connection was established. Values: 0 - RDP4 public key exchange scheme. 1 - X509 certificates were sent to client.

5.97 IVRDEServerInfoChangedEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Notification when the status of the VRDE server changes. Interested callees should use IVRDEServerInfo attributes to find out what is the current status.

5.98 IVetoEvent (IEvent)

> **Note:** This interface extends IEvent and therefore supports all its methods and attributes as well.

Base abstract interface for veto events.

5.98.1 addVeto

```
void IVetoEvent::addVeto(
           [in] wstring reason)
```

reason Reason for veto, could be null or empty string.

Adds a veto on this event.

5.98.2 getVetos

```
wstring[] IVetoEvent::getVetos()
```

Current veto reason list, if size is 0 - no veto.

5.98.3 isVetoed

```
boolean IVetoEvent::isVetoed()
```

If this event was vetoed.

5.99 IVirtualBox

The IVirtualBox interface represents the main interface exposed by the product that provides virtual machine management.

An instance of IVirtualBox is required for the product to do anything useful. Even though the interface does not expose this, internally, IVirtualBox is implemented as a singleton and actually lives in the process of the VirtualBox server (VBoxSVC.exe). This makes sure that IVirtualBox can track the state of all virtual machines on a particular host, regardless of which frontend started them.

To enumerate all the virtual machines on the host, use the machines[] attribute.

5.99.1 Attributes

5.99.1.1 version (read-only)

```
wstring IVirtualBox::version
```

A string representing the version number of the product. The format is 3 integer numbers divided by dots (e.g. 1.0.1). The last number represents the build number and will frequently change.

5.99.1.2 revision (read-only)

```
unsigned long IVirtualBox::revision
```

The internal build revision number of the product.

5.99.1.3 packageType (read-only)

```
wstring IVirtualBox::packageType
```

A string representing the package type of this product. The format is OS_ARCH_DIST where OS is either WINDOWS, LINUX, SOLARIS, DARWIN. ARCH is either 32BITS or 64BITS. DIST is either GENERIC, UBUNTU_606, UBUNTU_710, or something like this.

5.99.1.4 homeFolder (read-only)

```
wstring IVirtualBox::homeFolder
```

Full path to the directory where the global settings file, `VirtualBox.xml`, is stored.

In this version of VirtualBox, the value of this property is always `<user_dir>/.VirtualBox` (where `<user_dir>` is the path to the user directory, as determined by the host OS), and cannot be changed.

This path is also used as the base to resolve relative paths in places where relative paths are allowed (unless otherwise expressly indicated).

5.99.1.5 settingsFilePath (read-only)

```
wstring IVirtualBox::settingsFilePath
```

Full name of the global settings file. The value of this property corresponds to the value of homeFolder plus `/VirtualBox.xml`.

5.99.1.6 host (read-only)

```
IHost IVirtualBox::host
```

Associated host object.

5.99.1.7 systemProperties (read-only)

```
ISystemProperties IVirtualBox::systemProperties
```

Associated system information object.

5.99.1.8 machines (read-only)

```
IMachine IVirtualBox::machines[]
```

Array of machine objects registered within this VirtualBox instance.

5.99.1.9 hardDisks (read-only)

```
IMedium IVirtualBox::hardDisks[]
```

Array of medium objects known to this VirtualBox installation.

This array contains only base media. All differencing media of the given base medium can be enumerated using IMedium::children[].

5.99.1.10 DVDImages (read-only)

```
IMedium IVirtualBox::DVDImages[]
```

Array of CD/DVD image objects currently in use by this VirtualBox instance.

5.99.1.11 floppyImages (read-only)

`IMedium IVirtualBox::floppyImages[]`

Array of floppy image objects currently in use by this VirtualBox instance.

5.99.1.12 progressOperations (read-only)

`IProgress IVirtualBox::progressOperations[]`

5.99.1.13 guestOSTypes (read-only)

`IGuestOSType IVirtualBox::guestOSTypes[]`

5.99.1.14 sharedFolders (read-only)

`ISharedFolder IVirtualBox::sharedFolders[]`

Collection of global shared folders. Global shared folders are available to all virtual machines.
New shared folders are added to the collection using createSharedFolder(). Existing shared folders can be removed using removeSharedFolder().

> **Note:** In the current version of the product, global shared folders are not implemented and therefore this collection is always empty.

5.99.1.15 performanceCollector (read-only)

`IPerformanceCollector IVirtualBox::performanceCollector`

Associated performance collector object.

5.99.1.16 DHCPServers (read-only)

`IDHCPServer IVirtualBox::DHCPServers[]`

DHCP servers.

5.99.1.17 eventSource (read-only)

`IEventSource IVirtualBox::eventSource`

Event source for VirtualBox events.

5.99.1.18 extensionPackManager (read-only)

`IExtPackManager IVirtualBox::extensionPackManager`

> **Note:** This attribute is not supported in the web service.

The extension pack manager.

5.99.2 checkFirmwarePresent

```
boolean IVirtualBox::checkFirmwarePresent(
            [in] FirmwareType firmwareType,
            [in] wstring version,
            [out] wstring url,
            [out] wstring file)
```

firmwareType Type of firmware to check.

version Expected version number, usually empty string (presently ignored).

url Suggested URL to download this firmware from.

file Filename of firmware, only valid if result == TRUE.

Check if this VirtualBox installation has a firmware of the given type available, either system-wide or per-user. Optionally, this may return a hint where this firmware can be downloaded from.

5.99.3 composeMachineFilename

```
wstring IVirtualBox::composeMachineFilename(
            [in] wstring name,
            [in] wstring baseFolder)
```

name Suggested machine name.

baseFolder Base machine folder (optional).

Returns a recommended full path of the settings file name for a new virtual machine.
This API serves two purposes:

- It gets called by createMachine() if NULL is specified for the `settingsFile` argument there, which means that API should use a recommended default file name.
- It can be called manually by a client software before creating a machine, e.g. if that client wants to pre-create the machine directory to create virtual hard disks in that directory together with the new machine settings file. In that case, the file name should be stripped from the full settings file path returned by this function to obtain the machine directory.

See IMachine::name and createMachine() for more details about the machine name.
If `baseFolder` is a `null` or empty string (which is recommended), the default machine settings folder (see ISystemProperties::defaultMachineFolder) will be used as a base folder for the created machine, resulting in a file name like "/home/user/VirtualBox VMs/name/name.vbox". Otherwise the given base folder will be used.
This method does not access the host disks. In particular, it does not check for whether a machine of this name already exists.

5.99.4 createAppliance

```
IAppliance IVirtualBox::createAppliance()
```

Creates a new appliance object, which represents an appliance in the Open Virtual Machine Format (OVF). This can then be used to import an OVF appliance into VirtualBox or to export machines as an OVF appliance; see the documentation for IAppliance for details.

5.99.5 createDHCPServer

```
IDHCPServer IVirtualBox::createDHCPServer(
            [in] wstring name)
```

name server name

Creates a dhcp server settings to be used for the given internal network name
If this method fails, the following error codes may be reported:

- E_INVALIDARG: Host network interface name already exists.

5.99.6 createHardDisk

```
IMedium IVirtualBox::createHardDisk(
            [in] wstring format,
            [in] wstring location)
```

format Identifier of the storage format to use for the new medium.

location Location of the storage unit for the new medium.

Creates a new base medium object that will use the given storage format and location for medium data.
Note that the actual storage unit is not created by this method. In order to do it, and before you are able to attach the created medium to virtual machines, you must call one of the following methods to allocate a format-specific storage unit at the specified location:

- IMedium::createBaseStorage()
- IMedium::createDiffStorage()

Some medium attributes, such as IMedium::id, may remain uninitialized until the medium storage unit is successfully created by one of the above methods.
After the storage unit is successfully created, it will be accessible through the findMedium() method and can be found in the hardDisks[] array.
The list of all storage formats supported by this VirtualBox installation can be obtained using ISystemProperties::mediumFormats[]. If the format attribute is empty or null then the default storage format specified by ISystemProperties::defaultHardDiskFormat will be used for creating a storage unit of the medium.
Note that the format of the location string is storage format specific. See IMedium::location and IMedium for more details.
If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: format identifier is invalid. See ISystemProperties::mediumFormats[].
- VBOX_E_FILE_ERROR: location is a not valid file name (for file-based formats only).

5.99.7 createMachine

```
IMachine IVirtualBox::createMachine(
            [in] wstring settingsFile,
            [in] wstring name,
            [in] wstring osTypeId,
            [in] uuid id,
            [in] boolean forceOverwrite)
```

settingsFile Fully qualified path where the settings file should be created, or NULL for a default folder and file based on the `name` argument (see composeMachineFilename()).

name Machine name.

osTypeId Guest OS Type ID.

id Machine UUID (optional).

forceOverwrite If true, an existing machine settings file will be overwritten.

Creates a new virtual machine by creating a machine settings file at the given location.

VirtualBox machine settings files use a custom XML dialect. Starting with VirtualBox 4.0, a ".vbox" extension is recommended, but not enforced, and machine files can be created at arbitrary locations.

However, it is is recommended that machines be created in the default machine folder (e.g. "/home/user/VirtualBox VMs/name/name.vbox"; see ISystemProperties::defaultMachineFolder). If you specify NULL for the `settingsFile` argument, composeMachineFilename() is called automatically to have such a recommended name composed based on the machine name given in the `name` argument.

If the resulting settings file already exists, this method will fail, unless `forceOverwrite` is set.

The new machine is created unregistered, with the initial configuration set according to the specified guest OS type. A typical sequence of actions to create a new virtual machine is as follows:

1. Call this method to have a new machine created. The returned machine object will be "mutable" allowing to change any machine property.
2. Configure the machine using the appropriate attributes and methods.
3. Call IMachine::saveSettings() to write the settings to the machine's XML settings file. The configuration of the newly created machine will not be saved to disk until this method is called.
4. Call registerMachine() to add the machine to the list of machines known to VirtualBox.

The specified guest OS type identifier must match an ID of one of known guest OS types listed in the guestOSTypes[] array.

Optionally, you may specify an UUID of to assign to the created machine. However, this is not recommended and you should normally pass an empty (`null`) UUID to this method so that a new UUID will be automatically generated for every created machine. You can use UUID 00000000-0000-0000-0000-000000000000 as `null` value.

> **Note:** There is no way to change the name of the settings file or subfolder of the created machine directly.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND: osTypeId` is invalid.
- `VBOX_E_FILE_ERROR:` Resulting settings file name is invalid or the settings file already exists or could not be created due to an I/O error.
- `E_INVALIDARG: name` is empty or `null`.

5.99.8 createSharedFolder

```
void IVirtualBox::createSharedFolder(
           [in] wstring name,
           [in] wstring hostPath,
           [in] boolean writable,
           [in] boolean automount)
```

name Unique logical name of the shared folder.

hostPath Full path to the shared folder in the host file system.

writable Whether the share is writable or readonly

automount Whether the share gets automatically mounted by the guest or not.

Creates a new global shared folder by associating the given logical name with the given host path, adds it to the collection of shared folders and starts sharing it. Refer to the description of ISharedFolder to read more about logical names.

> **Note:** In the current implementation, this operation is not implemented.

5.99.9 findDHCPServerByNetworkName

```
IDHCPServer IVirtualBox::findDHCPServerByNetworkName(
           [in] wstring name)
```

name server name

Searches a dhcp server settings to be used for the given internal network name
If this method fails, the following error codes may be reported:

- `E_INVALIDARG:` Host network interface `name` already exists.

5.99.10 findMachine

```
IMachine IVirtualBox::findMachine(
           [in] wstring nameOrId)
```

nameOrId What to search for. This can either be the UUID or the name of a virtual machine.

Attempts to find a virtual machine given its name or UUID.

> **Note:** Inaccessible machines cannot be found by name, only by UUID, because their name cannot safely be determined.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND:` Could not find registered machine matching `nameOrId`.

5.99.11 findMedium

```
IMedium IVirtualBox::findMedium(
            [in] wstring location,
            [in] DeviceType type)
```

location What to search for. This can either be the UUID or the location string of an open medium.

type Device type (must be HardDisk, DVD or Floppy)

Returns a medium of the given type that uses the given fully qualified location or UUID to store medium data.

The given medium must be known to this VirtualBox installation, i.e. it must be previously created by createHardDisk() or opened by openMedium().

The search is done by comparing the value of the `location` argument to the IMedium::location and IMedium::id attributes of each known medium.

On case sensitive file systems, a case sensitive comparison is performed, otherwise the case of symbols in the file path is ignored.

If this method fails, the following error codes may be reported:

- `VBOX_E_OBJECT_NOT_FOUND`: No medium object matching `location` found.

5.99.12 getExtraData

```
wstring IVirtualBox::getExtraData(
            [in] wstring key)
```

key Name of the data key to get.

Returns associated global extra data.

If the requested data `key` does not exist, this function will succeed and return an empty string in the `value` argument.

If this method fails, the following error codes may be reported:

- `VBOX_E_FILE_ERROR`: Settings file not accessible.
- `VBOX_E_XML_ERROR`: Could not parse the settings file.

5.99.13 getExtraDataKeys

```
wstring[] IVirtualBox::getExtraDataKeys()
```

Returns an array representing the global extra data keys which currently have values defined.

5.99.14 getGuestOSType

```
IGuestOSType IVirtualBox::getGuestOSType(
            [in] uuid id)
```

id Guest OS type ID string.

Returns an object describing the specified guest OS type.

The requested guest OS type is specified using a string which is a mnemonic identifier of the guest operating system, such as `"win31"` or `"ubuntu"`. The guest OS type ID of a particular virtual machine can be read or set using the IMachine::OSTypeId attribute.

The guestOSTypes[] collection contains all available guest OS type objects. Each object has an IGuestOSType::id attribute which contains an identifier of the guest OS this object describes.

If this method fails, the following error codes may be reported:

- `E_INVALIDARG`: `id` is not a valid Guest OS type.

5.99.15 openMachine

```
IMachine IVirtualBox::openMachine(
           [in] wstring settingsFile)
```

settingsFile Name of the machine settings file.

Opens a virtual machine from the existing settings file. The opened machine remains unregistered until you call registerMachine().

The specified settings file name must be fully qualified. The file must exist and be a valid machine XML settings file whose contents will be used to construct the machine object.

If this method fails, the following error codes may be reported:

- VBOX_E_FILE_ERROR: Settings file name invalid, not found or sharing violation.

5.99.16 openMedium

```
IMedium IVirtualBox::openMedium(
           [in] wstring location,
           [in] DeviceType deviceType,
           [in] AccessMode accessMode)
```

location Location of the storage unit that contains medium data in one of the supported storage formats.

deviceType Must be one of "HardDisk", "DVD" or "Floppy".

accessMode Whether to open the image in read/write or read-only mode. For a "DVD" device type, this is ignored and read-only mode is always assumed.

Opens a medium from an existing storage location.

Once a medium has been opened, it can be passed to other VirtualBox methods, in particular to IMachine::attachDevice().

Depending on the given device type, the file at the storage location must be in one of the media formats understood by VirtualBox:

- With a "HardDisk" device type, the file must be a hard disk image in one of the formats supported by VirtualBox (see ISystemProperties::mediumFormats[]). After this method succeeds, if the medium is a base medium, it will be added to the hardDisks[] array attribute.
- With a "DVD" device type, the file must be an ISO 9960 CD/DVD image. After this method succeeds, the medium will be added to the DVDImages[] array attribute.
- With a "Floppy" device type, the file must be an RAW floppy image. After this method succeeds, the medium will be added to the floppyImages[] array attribute.

After having been opened, the medium can be found by the findMedium() method and can be attached to virtual machines. See IMedium for more details.

The UUID of the newly opened medium will either be retrieved from the storage location, if the format supports it (e.g. for hard disk images), or a new UUID will be randomly generated (e.g. for ISO and RAW files). If for some reason you need to change the medium's UUID, use IMedium::setIDs().

If a differencing hard disk medium is to be opened by this method, the operation will succeed only if its parent medium and all ancestors, if any, are already known to this VirtualBox installation (for example, were opened by this method before).

This method attempts to guess the storage format of the specified medium by reading medium data at the specified location.

If accessMode is ReadWrite (which it should be for hard disks and floppies), the image is opened for read/write access and must have according permissions, as VirtualBox may actually write status information into the disk's metadata sections.

Note that write access is required for all typical hard disk usage in VirtualBox, since VirtualBox may need to write metadata such as a UUID into the image. The only exception is opening a source image temporarily for copying and cloning (see IMedium::cloneTo() when the image will be closed again soon.

The format of the location string is storage format specific. See IMedium::location and IMedium for more details.

If this method fails, the following error codes may be reported:

- VBOX_E_FILE_ERROR: Invalid medium storage file location or could not find the medium at the specified location.
- VBOX_E_IPRT_ERROR: Could not get medium storage format.
- E_INVALIDARG: Invalid medium storage format.
- VBOX_E_INVALID_OBJECT_STATE: Medium has already been added to a media registry.

5.99.17 registerMachine

```
void IVirtualBox::registerMachine(
            [in] IMachine machine)
```

machine

Registers the machine previously created using createMachine() or opened using openMachine() within this VirtualBox installation. After successful method invocation, the IMachineRegisteredEvent event is fired.

> **Note:** This method implicitly calls IMachine::saveSettings() to save all current machine settings before registering it.

If this method fails, the following error codes may be reported:

- VBOX_E_OBJECT_NOT_FOUND: No matching virtual machine found.
- VBOX_E_INVALID_OBJECT_STATE: Virtual machine was not created within this VirtualBox instance.

5.99.18 removeDHCPServer

```
void IVirtualBox::removeDHCPServer(
            [in] IDHCPServer server)
```

server Dhcp server settings to be removed

Removes the dhcp server settings

If this method fails, the following error codes may be reported:

- E_INVALIDARG: Host network interface name already exists.

5.99.19 removeSharedFolder

```
void IVirtualBox::removeSharedFolder(
           [in] wstring name)
```

name Logical name of the shared folder to remove.

Removes the global shared folder with the given name previously created by createSharedFolder() from the collection of shared folders and stops sharing it.

> **Note:** In the current implementation, this operation is not implemented.

5.99.20 setExtraData

```
void IVirtualBox::setExtraData(
           [in] wstring key,
           [in] wstring value)
```

key Name of the data key to set.

value Value to assign to the key.

Sets associated global extra data.
If you pass `null` or empty string as a key `value`, the given `key` will be deleted.

> **Note:** Before performing the actual data change, this method will ask all registered event listener using the IExtraDataCanChangeEvent notification for a permission. If one of the listeners refuses the new value, the change will not be performed.

> **Note:** On success, the IExtraDataChangedEvent notification is called to inform all registered listeners about a successful data change.

If this method fails, the following error codes may be reported:

- `VBOX_E_FILE_ERROR:` Settings file not accessible.
- `VBOX_E_XML_ERROR:` Could not parse the settings file.
- `E_ACCESSDENIED:` Modification request refused.

5.100 IVirtualBoxClient

> **Note:** This interface is not supported in the web service.

Convenience interface for client applications. Treat this as a singleton, i.e. never create more than one instance of this interface.
At the moment only available for clients of the local API (not usable via the webservice). Once the session logic is redesigned this might change.

5.100.1 Attributes

5.100.1.1 virtualBox (read-only)

`IVirtualBox IVirtualBoxClient::virtualBox`

Reference to the server-side API root object.

5.100.1.2 session (read-only)

`ISession IVirtualBoxClient::session`

Create a new session object and return the reference to it.

5.100.1.3 eventSource (read-only)

`IEventSource IVirtualBoxClient::eventSource`

Event source for VirtualBoxClient events.

5.101 IVirtualBoxErrorInfo

The IVirtualBoxErrorInfo interface represents extended error information.

Extended error information can be set by VirtualBox components after unsuccessful or partially successful method invocation. This information can be retrieved by the calling party as an IVirtualBoxErrorInfo object and then shown to the client in addition to the plain 32-bit result code.

In MS COM, this interface extends the IErrorInfo interface, in XPCOM, it extends the nsIException interface. In both cases, it provides a set of common attributes to retrieve error information.

Sometimes invocation of some component's method may involve methods of other components that may also fail (independently of this method's failure), or a series of non-fatal errors may precede a fatal error that causes method failure. In cases like that, it may be desirable to preserve information about all errors happened during method invocation and deliver it to the caller. The next attribute is intended specifically for this purpose and allows to represent a chain of errors through a single IVirtualBoxErrorInfo object set after method invocation.

Note that errors are stored to a chain in the reverse order, i.e. the initial error object you query right after method invocation is the last error set by the callee, the object it points to in the `next` attribute is the previous error and so on, up to the first error (which is the last in the chain).

5.101.1 Attributes

5.101.1.1 resultCode (read-only)

`long IVirtualBoxErrorInfo::resultCode`

Result code of the error. Usually, it will be the same as the result code returned by the method that provided this error information, but not always. For example, on Win32, CoCreateInstance() will most likely return E_NOINTERFACE upon unsuccessful component instantiation attempt, but not the value the component factory returned. Value is typed 'long', not 'result', to make interface usable from scripting languages.

> **Note:** In MS COM, there is no equivalent. In XPCOM, it is the same as nsIException::result.

5.101.1.2 interfaceID (read-only)

`uuid IVirtualBoxErrorInfo::interfaceID`

UUID of the interface that defined the error.

> **Note:** In MS COM, it is the same as IErrorInfo::GetGUID, except for the data type. In XPCOM, there is no equivalent.

5.101.1.3 component (read-only)

`wstring IVirtualBoxErrorInfo::component`

Name of the component that generated the error.

> **Note:** In MS COM, it is the same as IErrorInfo::GetSource. In XPCOM, there is no equivalent.

5.101.1.4 text (read-only)

`wstring IVirtualBoxErrorInfo::text`

Text description of the error.

> **Note:** In MS COM, it is the same as IErrorInfo::GetDescription. In XPCOM, it is the same as nsIException::message.

5.101.1.5 next (read-only)

`IVirtualBoxErrorInfo IVirtualBoxErrorInfo::next`

Next error object if there is any, or `null` otherwise.

> **Note:** In MS COM, there is no equivalent. In XPCOM, it is the same as nsIException::inner.

5.102 IVirtualSystemDescription

Represents one virtual system (machine) in an appliance. This interface is used in the IAppliance::virtualSystemDescriptions[] array. After IAppliance::interpret() has been called, that array contains information about how the virtual systems described in the OVF should best be imported into VirtualBox virtual machines. See IAppliance for the steps required to import an OVF into VirtualBox.

5.102.1 Attributes

5.102.1.1 count (read-only)

`unsigned long IVirtualSystemDescription::count`

Return the number of virtual system description entries.

5.102.2 addDescription

```
void IVirtualSystemDescription::addDescription(
           [in] VirtualSystemDescriptionType aType,
           [in] wstring aVBoxValue,
           [in] wstring aExtraConfigValue)
```

aType

aVBoxValue

aExtraConfigValue

This method adds an additional description entry to the stack of already available descriptions for this virtual system. This is handy for writing values which aren't directly supported by VirtualBox. One example would be the License type of VirtualSystemDescriptionType.

5.102.3 getDescription

```
void IVirtualSystemDescription::getDescription(
           [out] VirtualSystemDescriptionType aTypes[],
           [out] wstring aRefs[],
           [out] wstring aOvfValues[],
           [out] wstring aVBoxValues[],
           [out] wstring aExtraConfigValues[])
```

aTypes

aRefs

aOvfValues

aVBoxValues

aExtraConfigValues

Returns information about the virtual system as arrays of instruction items. In each array, the items with the same indices correspond and jointly represent an import instruction for VirtualBox.

The list below identifies the value sets that are possible depending on the VirtualSystemDescriptionType enum value in the array item in `aTypes[]`. In each case, the array item with the same index in `aOvfValues[]` will contain the original value as contained in the OVF file (just for informational purposes), and the corresponding item in `aVBoxValues[]` will contain a suggested value to be used for VirtualBox. Depending on the description type, the `aExtraConfigValues[]` array item may also be used.

- "OS": the guest operating system type. There must be exactly one such array item on import. The corresponding item in `aVBoxValues[]` contains the suggested guest operating system for VirtualBox. This will be one of the values listed in IVirtualBox::guestOSTypes[]. The corresponding item in `aOvfValues[]` will contain a numerical value that described the operating system in the OVF.
- "Name": the name to give to the new virtual machine. There can be at most one such array item; if none is present on import, then an automatic name will be created from the operating system type. The corresponding item im `aOvfValues[]` will contain the suggested virtual machine name from the OVF file, and `aVBoxValues[]` will contain a suggestion for a unique VirtualBox IMachine name that does not exist yet.
- "Description": an arbitrary description.

- "License": the EULA section from the OVF, if present. It is the responsibility of the calling code to display such a license for agreement; the Main API does not enforce any such policy.
- Miscellaneous: reserved for future use.
- "CPU": the number of CPUs. There can be at most one such item, which will presently be ignored.
- "Memory": the amount of guest RAM, in bytes. There can be at most one such array item; if none is present on import, then VirtualBox will set a meaningful default based on the operating system type.
- "HardDiskControllerIDE": an IDE hard disk controller. There can be at most two such items. An optional value in `aOvfValues[]` and `aVBoxValues[]` can be "PIIX3" or "PIIX4" to specify the type of IDE controller; this corresponds to the ResourceSubType element which VirtualBox writes into the OVF. The matching item in the `aRefs[]` array will contain an integer that items of the "Harddisk" type can use to specify which hard disk controller a virtual disk should be connected to. Note that in OVF, an IDE controller has two channels, corresponding to "master" and "slave" in traditional terminology, whereas the IDE storage controller that VirtualBox supports in its virtual machines supports four channels (primary master, primary slave, secondary master, secondary slave) and thus maps to two IDE controllers in the OVF sense.
- "HardDiskControllerSATA": an SATA hard disk controller. There can be at most one such item. This has no value in `aOvfValues[]` or `aVBoxValues[]`. The matching item in the `aRefs[]` array will be used as with IDE controllers (see above).
- "HardDiskControllerSCSI": a SCSI hard disk controller. There can be at most one such item. The items in `aOvfValues[]` and `aVBoxValues[]` will either be "LsiLogic", "BusLogic" or "LsiLogicSas". (Note that in OVF, the LsiLogicSas controller is treated as a SCSI controller whereas VirtualBox considers it a class of storage controllers of its own; see StorageControllerType). The matching item in the `aRefs[]` array will be used as with IDE controllers (see above).
- "HardDiskImage": a virtual hard disk, most probably as a reference to an image file. There can be an arbitrary number of these items, one for each virtual disk image that accompanies the OVF.

  The array item in `aOvfValues[]` will contain the file specification from the OVF file (without a path since the image file should be in the same location as the OVF file itself), whereas the item in `aVBoxValues[]` will contain a qualified path specification to where VirtualBox uses the hard disk image. This means that on import the image will be copied and converted from the "ovf" location to the "vbox" location; on export, this will be handled the other way round.

  The matching item in the `aExtraConfigValues[]` array must contain a string of the following format: "controller=<index>;channel=<c>" In this string, <index> must be an integer specifying the hard disk controller to connect the image to. That number must be the index of an array item with one of the hard disk controller types (HardDiskControllerSCSI, HardDiskControllerSATA, HardDiskControllerIDE). In addition, <c> must specify the channel to use on that controller. For IDE controllers, this can be 0 or 1 for master or slave, respectively. For compatibility with VirtualBox versions before 3.2, the values 2 and 3 (for secondary master and secondary slave) are also supported, but no longer exported. For SATA and SCSI controllers, the channel can range from 0-29.
- "CDROM": a virtual CD-ROM drive. The matching item in `aExtraConfigValue[]` contains the same attachment information as with "HardDiskImage" items.

- "CDROM": a virtual floppy drive. The matching item in `aExtraConfigValue[]` contains the same attachment information as with "HardDiskImage" items.
- "NetworkAdapter": a network adapter. The array item in `aVBoxValues[]` will specify the hardware for the network adapter, whereas the array item in `aExtraConfigValues[]` will have a string of the "type=<X>" format, where <X> must be either "NAT" or "Bridged".
- "USBController": a USB controller. There can be at most one such item. If and only if such an item ispresent, USB support will be enabled for the new virtual machine.
- "SoundCard": a sound card. There can be at most one such item. If and only if such an item is present, sound support will be enabled for the new virtual machine. Note that the virtual machine in VirtualBox will always be presented with the standard VirtualBox soundcard, which may be different from the virtual soundcard expected by the appliance.

5.102.4 getDescriptionByType

```
void IVirtualSystemDescription::getDescriptionByType(
            [in] VirtualSystemDescriptionType aType,
            [out] VirtualSystemDescriptionType aTypes[],
            [out] wstring aRefs[],
            [out] wstring aOvfValues[],
            [out] wstring aVBoxValues[],
            [out] wstring aExtraConfigValues[])
```

aType

aTypes

aRefs

aOvfValues

aVBoxValues

aExtraConfigValues

This is the same as getDescription() except that you can specify which types should be returned.

5.102.5 getValuesByType

```
wstring[] IVirtualSystemDescription::getValuesByType(
            [in] VirtualSystemDescriptionType aType,
            [in] VirtualSystemDescriptionValueType aWhich)
```

aType

aWhich

This is the same as getDescriptionByType() except that you can specify which value types should be returned. See VirtualSystemDescriptionValueType for possible values.

5.102.6 setFinalValues

```
void IVirtualSystemDescription::setFinalValues(
            [in] boolean aEnabled[],
            [in] wstring aVBoxValues[],
            [in] wstring aExtraConfigValues[])
```

aEnabled

aVBoxValues

aExtraConfigValues

This method allows the appliance's user to change the configuration for the virtual system descriptions. For each array item returned from getDescription(), you must pass in one boolean value and one configuration value.

Each item in the boolean array determines whether the particular configuration item should be enabled. You can only disable items of the types HardDiskControllerIDE, HardDiskControllerSATA, HardDiskControllerSCSI, HardDiskImage, CDROM, Floppy, NetworkAdapter, USBController and SoundCard.

For the "vbox" and "extra configuration" values, if you pass in the same arrays as returned in the aVBoxValues and aExtraConfigValues arrays from getDescription(), the configuration remains unchanged. Please see the documentation for getDescription() for valid configuration values for the individual array item types. If the corresponding item in the aEnabled array is `false`, the configuration value is ignored.

5.103 IWebsessionManager

Note: This interface is supported in the web service only, not in COM/XPCOM.

Websession manager. This provides essential services to webservice clients.

5.103.1 getSessionObject

```
ISession IWebsessionManager::getSessionObject(
            [in] IVirtualBox refIVirtualBox)
```

refIVirtualBox

Returns a managed object reference to the internal ISession object that was created for this web service session when the client logged on.

See also: ISession

5.103.2 logoff

```
void IWebsessionManager::logoff(
            [in] IVirtualBox refIVirtualBox)
```

refIVirtualBox

Logs off the client who has previously logged on with logoff() and destroys all resources associated with the session (most importantly, all managed objects created in the server while the session was active).

5.103.3 logon

```
IVirtualBox IWebsessionManager::logon(
            [in] wstring username,
            [in] wstring password)
```

username

password

Logs a new client onto the webservice and returns a managed object reference to the IVirtualBox instance, which the client can then use as a basis to further queries, since all calls to the VirtualBox API are based on the IVirtualBox interface, in one way or the other.

6 Enumerations (enums)

6.1 AccessMode

Access mode for opening files.

ReadOnly

ReadWrite

6.2 AdditionsRunLevelType

Guest Additions run level type.

None Guest Additions are not loaded.

System Guest drivers are loaded.

Userland Common components (such as application services) are loaded.

Desktop Per-user desktop components are loaded.

6.3 AdditionsUpdateFlag

Guest Additions update flags.

None No flag set.

WaitForUpdateStartOnly Only wait for the update process being started and do not wait while peforming the actual update.

6.4 AudioControllerType

Virtual audio controller type.

AC97

SB16

HDA

6.5 AudioDriverType

Host audio driver type.

Null Null value, also means “dummy audio driver”.

WinMM Windows multimedia (Windows hosts only).

OSS Open Sound System (Linux hosts only).

ALSA Advanced Linux Sound Architecture (Linux hosts only).

DirectSound DirectSound (Windows hosts only).

CoreAudio CoreAudio (Mac hosts only).

MMPM Reserved for historical reasons.

Pulse PulseAudio (Linux hosts only).

SolAudio Solaris audio (Solaris hosts only).

6.6 AuthType

VirtualBox authentication type.

Null Null value, also means “no authentication”.

External

Guest

6.7 BIOSBootMenuMode

BIOS boot menu mode.

Disabled

MenuOnly

MessageAndMenu

6.8 BandwidthGroupType

Type of a bandwidth control group.

Null Null type, must be first.

Disk The bandwidth group controls disk I/O.

Network The bandwidth group controls network I/O.

6.9 CPUPropertyType

Virtual CPU property type. This enumeration represents possible values of the IMachine get- and setCPUProperty methods.

Null Null value (never used by the API).

PAE This setting determines whether VirtualBox will expose the Physical Address Extension (PAE) feature of the host CPU to the guest. Note that in case PAE is not available, it will not be reported.

Synthetic This setting determines whether VirtualBox will expose a synthetic CPU to the guest to allow teleporting between host systems that differ significantly.

6.10 ChipsetType

Type of emulated chipset (mostly southbridge).

Null `null` value. Never used by the API.

PIIX3 A PIIX3 (PCI IDE ISA Xcelerator) chipset.

ICH9 A ICH9 (I/O Controller Hub) chipset.

6.11 CleanupMode

Cleanup mode, used with IMachine::unregister().

UnregisterOnly Unregister only the machine, but neither delete snapshots nor detach media.

DetachAllReturnNone Delete all snapshots and detach all media but return none; this will keep all media registered.

DetachAllReturnHardDisksOnly Delete all snapshots, detach all media and return hard disks for closing, but not removable media.

Full Delete all snapshots, detach all media and return all media for closing.

6.12 ClipboardMode

Host-Guest clipboard interchange mode.

Disabled

HostToGuest

GuestToHost

Bidirectional

6.13 CopyFileFlag

Host/Guest copy flags. At the moment none of these flags are implemented.

None No flag set.

Recursive Copy directories recursively.

Update Only copy when the source file is newer than the destination file or when the destination file is missing.

FollowLinks Follow symbolic links.

6.14 CreateDirectoryFlag

Directory creation flags.

None No flag set.

Parents No error if existing, make parent directories as needed.

6.15 DataFlags

None

Mandatory

Expert

Array

FlagMask

6.16 DataType

Int32

Int8

String

6.17 DeviceActivity

Device activity for IConsole::getDeviceActivity().

Null

Idle

Reading

Writing

6.18 DeviceType

Device type.

Null Null value, may also mean “no device” (not allowed for IConsole::getDeviceActivity()).

Floppy Floppy device.

DVD CD/DVD-ROM device.

HardDisk Hard disk device.

Network Network device.

USB USB device.

SharedFolder Shared folder device.

6.19 ExecuteProcessFlag

Guest process execution flags.

None No flag set.

WaitForProcessStartOnly Only use the specified timeout value to wait for starting the guest process - the guest process itself then uses an infinite timeout.

IgnoreOrphanedProcesses Do not report an error when executed processes are still alive when VBoxService or the guest OS is shutting down.

Hidden Don't show the started process according to the guest OS guidelines.

NoProfile Do not use the user's profile data when exeuting a process.

6.20 FaultToleranceState

Used with IMachine::faultToleranceState.

Inactive No fault tolerance enabled.

Master Fault tolerant master VM.

Standby Fault tolerant standby VM.

6.21 FirmwareType

Firmware type.

BIOS BIOS Firmware.

EFI EFI Firmware, bitness detected basing on OS type.

EFI32 Efi firmware, 32-bit.

EFI64 Efi firmware, 64-bit.

EFIDUAL Efi firmware, combined 32 and 64-bit.

6.22 FramebufferPixelFormat

Format of the video memory buffer. Constants represented by this enum can be used to test for particular values of IFramebuffer::pixelFormat. See also IFramebuffer::requestResize().

See also www.fourcc.org for more information about FOURCC pixel formats.

Opaque Unknown buffer format (the user may not assume any particular format of the buffer).

FOURCC_RGB Basic RGB format (IFramebuffer::bitsPerPixel determines the bit layout).

6.23 GuestMonitorChangedEventType

How the guest monitor has been changed.

Enabled The guest monitor has been enabled by the guest.

Disabled The guest monitor has been disabled by the guest.

NewOrigin The guest monitor origin has changed in the guest.

6.24 HWVirtExPropertyType

Hardware virtualization property type. This enumeration represents possible values for the IMachine::getHWVirtExProperty() and IMachine::setHWVirtExProperty() methods.

Null Null value (never used by the API).

Enabled Whether hardware virtualization (VT-x/AMD-V) is enabled at all. If such extensions are not available, they will not be used.

Exclusive Whether hardware virtualization is used exclusively by VirtualBox. When enabled, VirtualBox assumes it can acquire full and exclusive access to the VT-x or AMD-V feature of the host. To share these with other hypervisors, you must disable this property.

VPID Whether VT-x VPID is enabled. If this extension is not available, it will not be used.

NestedPaging Whether Nested Paging is enabled. If this extension is not available, it will not be used.

LargePages Whether large page allocation is enabled; requires nested paging and a 64 bits host.

Force Whether the VM should fail to start if hardware virtualization (VT-x/AMD-V) cannot be used. If not set, there will be an automatic fallback to software virtualization.

6.25 HostNetworkInterfaceMediumType

Type of encapsulation. Ethernet encapsulation includes both wired and wireless Ethernet connections. See also: IHostNetworkInterface

Unknown The type of interface cannot be determined.

Ethernet Ethernet frame encapsulation.

PPP Point-to-point protocol encapsulation.

SLIP Serial line IP encapsulation.

6.26 HostNetworkInterfaceStatus

Current status of the interface. See also: IHostNetworkInterface

Unknown The state of interface cannot be determined.

Up The interface is fully operational.

Down The interface is not functioning.

6.27 HostNetworkInterfaceType

Network interface type.

Bridged

HostOnly

6.28 KeyboardHidType

Type of keyboard device used in a virtual machine.

None No keyboard.

PS2Keyboard PS/2 keyboard.

USBKeyboard USB keyboard.

ComboKeyboard Combined device, working as PS/2 or USB keyboard, depending on guest behavior. Using of such device can have negative performance implications.

6.29 LockType

Used with IMachine::lockMachine().

Write Lock the machine for writing.

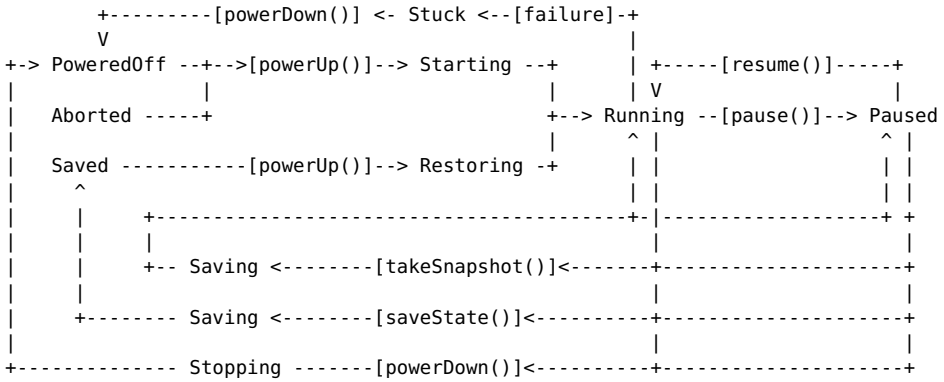
Shared Request only a shared read lock for remote-controlling the machine.

6.30 MachineState

Virtual machine execution state.

This enumeration represents possible values of the IMachine::state attribute.

Below is the basic virtual machine state diagram. It shows how the state changes during virtual machine execution. The text in square braces shows a method of the IConsole interface that performs the given state transition.



Note that states to the right from PoweredOff, Aborted and Saved in the above diagram are called *online VM states*. These states represent the virtual machine which is being executed in a dedicated process (usually with a GUI window attached to it where you can see the activity of the virtual machine and interact with it). There are two special pseudo-states, FirstOnline and LastOnline, that can be used in relational expressions to detect if the given machine state is online or not:

```
if (machine.GetState() >= MachineState_FirstOnline &&
    machine.GetState() <= MachineState_LastOnline)
{
    ...the machine is being executed...
}
```

When the virtual machine is in one of the online VM states (that is, being executed), only a few machine settings can be modified. Methods working with such settings contain an explicit note about that. An attempt to change any other setting or perform a modifying operation during this time will result in the VBOX_E_INVALID_VM_STATE error.

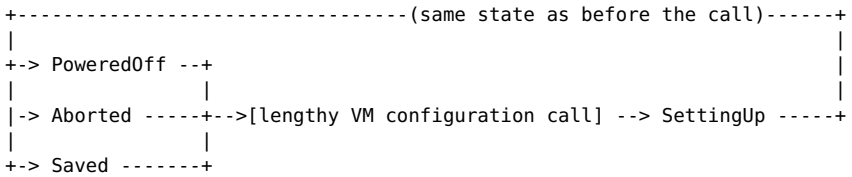
All online states except Running, Paused and Stuck are transitional: they represent temporary conditions of the virtual machine that will last as long as the operation that initiated such a condition.

The Stuck state is a special case. It means that execution of the machine has reached the "Guru Meditation" condition. This condition indicates an internal VMM (virtual machine manager) failure which may happen as a result of either an unhandled low-level virtual hardware exception or one of the recompiler exceptions (such as the *too-many-traps* condition).

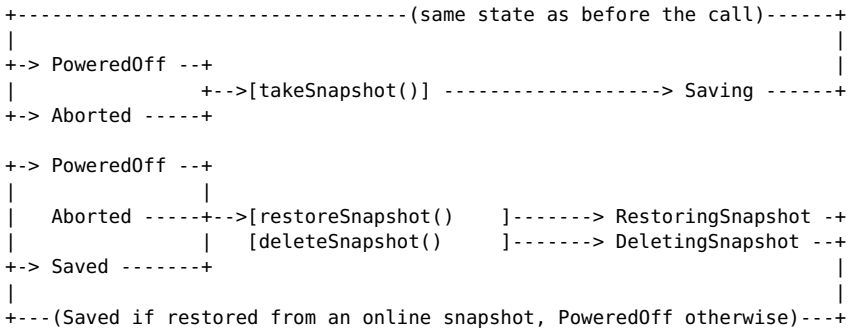
Note also that any online VM state may transit to the Aborted state. This happens if the process that is executing the virtual machine terminates unexpectedly (for example, crashes). Other than that, the Aborted state is equivalent to PoweredOff.

There are also a few additional state diagrams that do not deal with virtual machine execution and therefore are shown separately. The states shown on these diagrams are called *offline VM states* (this includes PoweredOff, Aborted and Saved too).

The first diagram shows what happens when a lengthy setup operation is being executed (such as IMachine::attachDevice()).



The next two diagrams demonstrate the process of taking a snapshot of a powered off virtual machine, restoring the state to that as of a snapshot or deleting a snapshot, respectively.



Note that the Saving state is present in both the offline state group and online state group. Currently, the only way to determine what group is assumed in a particular case is to remember the previous machine state: if it was Running or Paused, then Saving is an online state, otherwise it is an offline state. This inconsistency may be removed in one of the future versions of VirtualBox by adding a new state.

Null Null value (never used by the API).

PoweredOff The machine is not running and has no saved execution state; it has either never been started or been shut down successfully.

Saved The machine is not currently running, but the execution state of the machine has been saved to an external file when it was running, from where it can be resumed.

Teleported The machine was teleported to a different host (or process) and then powered off. Take care when powering it on again may corrupt resources it shares with the teleportation target (e.g. disk and network).

Aborted The process running the machine has terminated abnormally. This may indicate a crash of the VM process in host execution context, or the VM process has been terminated externally.

Running The machine is currently being executed.

Paused Execution of the machine has been paused.

Stuck Execution of the machine has reached the "Guru Meditation" condition. This indicates a severe error in the hypervisor itself.

Teleporting The machine is about to be teleported to a different host or process. It is possible to pause a machine in this state, but it will go to the `TeleportingPausedVM` state and it will not be possible to resume it again unless the teleportation fails.

LiveSnapshotting A live snapshot is being taken. The machine is running normally, but some of the runtime configuration options are inaccessible. Also, if paused while in this state it will transition to `Saving` and it will not be resume the execution until the snapshot operation has completed.

Starting Machine is being started after powering it on from a zero execution state.

Stopping Machine is being normally stopped powering it off, or after the guest OS has initiated a shutdown sequence.

Saving Machine is saving its execution state to a file, or an online snapshot of the machine is being taken.

Restoring Execution state of the machine is being restored from a file after powering it on from the saved execution state.

TeleportingPausedVM The machine is being teleported to another host or process, but it is not running. This is the paused variant of the `state`.

TeleportingIn Teleporting the machine state in from another host or process.

FaultTolerantSyncing The machine is being synced with a fault tolerant VM running elsewhere.

DeletingSnapshotOnline Like `DeletingSnapshot`, but the merging of media is ongoing in the background while the machine is running.

DeletingSnapshotPaused Like `DeletingSnapshotOnline`, but the machine was paused when the merging of differencing media was started.

RestoringSnapshot A machine snapshot is being restored; this typically does not take long.

DeletingSnapshot A machine snapshot is being deleted; this can take a long time since this may require merging differencing media. This value indicates that the machine is not running while the snapshot is being deleted.

SettingUp Lengthy setup operation is in progress.

FirstOnline Pseudo-state: first online state (for use in relational expressions).

LastOnline Pseudo-state: last online state (for use in relational expressions).

FirstTransient Pseudo-state: first transient state (for use in relational expressions).

LastTransient Pseudo-state: last transient state (for use in relational expressions).

6.31 MediumFormatCapabilities

Medium format capability flags.

Uuid Supports UUIDs as expected by VirtualBox code.

CreateFixed Supports creating fixed size images, allocating all space instantly.

CreateDynamic Supports creating dynamically growing images, allocating space on demand.

CreateSplit2G Supports creating images split in chunks of a bit less than 2 GBytes.

Differencing Supports being used as a format for differencing media (see IMedium::createDiffStorage()).

Asynchronous Supports asynchronous I/O operations for at least some configurations.

File The format backend operates on files (the IMedium::location attribute of the medium specifies a file used to store medium data; for a list of supported file extensions see IMediumFormat::describeFileExtensions()).

Properties The format backend uses the property interface to configure the storage location and properties (the IMediumFormat::describeProperties() method is used to get access to properties supported by the given medium format).

TcpNetworking The format backend uses the TCP networking interface for network access.

VFS The format backend supports virtual filesystem functionality.

CapabilityMask

6.32 MediumState

Virtual medium state. See also: IMedium

NotCreated Associated medium storage does not exist (either was not created yet or was deleted).

Created Associated storage exists and accessible; this gets set if the accessibility check performed by IMedium::refreshState() was successful.

LockedRead Medium is locked for reading (see IMedium::lockRead()), no data modification is possible.

LockedWrite Medium is locked for writing (see IMedium::lockWrite()), no concurrent data reading or modification is possible.

Inaccessible Medium accessibility check (see IMedium::refreshState()) has not yet been performed, or else, associated medium storage is not accessible. In the first case, IMedium::lastAccessError is empty, in the second case, it describes the error that occurred.

Creating Associated medium storage is being created.

Deleting Associated medium storage is being deleted.

6.33 MediumType

Virtual medium type. For each IMedium, this defines how the medium is attached to a virtual machine (see IMediumAttachment) and what happens when a snapshot (see ISnapshot) is taken of a virtual machine which has the medium attached. At the moment DVD and floppy media are always of type "writethrough".

Normal Normal medium (attached directly or indirectly, preserved when taking snapshots).

Immutable Immutable medium (attached indirectly, changes are wiped out the next time the virtual machine is started).

Writethrough Write through medium (attached directly, ignored when taking snapshots).

Shareable Allow using this medium concurrently by several machines.

> **Note:** Present since VirtualBox 3.2.0, and accepted since 3.2.8.

Readonly A readonly medium, which can of course be used by several machines.

> **Note:** Present and accepted since VirtualBox 4.0.

MultiAttach A medium which is is indirectly attached, so that one base medium can be used for several VMs which have their own differencing medium to store their modifications. In some sense a variant of Immutable with unset AutoReset flag in each differencing medium.

> **Note:** Present and accepted since VirtualBox 4.0.

6.34 MediumVariant

Virtual medium image variant. More than one flag may be set. See also: IMedium

Standard No particular variant requested, results in using the backend default.

VmdkSplit2G VMDK image split in chunks of less than 2GByte.

VmdkStreamOptimized VMDK streamOptimized image. Special import/export format which is read-only/append-only.

VmdkESX VMDK format variant used on ESX products.

Fixed Fixed image. Only allowed for base images.

Diff Differencing image. Only allowed for child images.

6.35 MouseButtonState

Mouse button state.

LeftButton

RightButton

MiddleButton

WheelUp

WheelDown

XButton1

XButton2

MouseStateMask

6.36 NATAliasMode

AliasLog

AliasProxyOnly

AliasUseSamePorts

6.37 NATProtocol

Protocol definitions used with NAT port-forwarding rules.

UDP Port-forwarding uses UDP protocol.

TCP Port-forwarding uses TCP protocol.

6.38 NetworkAdapterType

Network adapter type.

Null Null value (never used by the API).

Am79C970A AMD PCNet-PCI II network card (Am79C970A).

Am79C973 AMD PCNet-FAST III network card (Am79C973).

I82540EM Intel PRO/1000 MT Desktop network card (82540EM).

I82543GC Intel PRO/1000 T Server network card (82543GC).

I82545EM Intel PRO/1000 MT Server network card (82545EM).

Virtio Virtio network device.

6.39 NetworkAttachmentType

Network attachment type.

Null Null value, also means “not attached”.

NAT

Bridged

Internal

HostOnly

VDE

6.40 PointingHidType

Type of pointing device used in a virtual machine.

None No mouse.

PS2Mouse PS/2 auxiliary device, a.k.a. mouse.

USBMouse USB mouse (relative pointer).

USBTablet USB tablet (absolute pointer).

ComboMouse Combined device, working as PS/2 or USB mouse, depending on guest behavior. Using of such device can have negative performance implications.

6.41 PortMode

The PortMode enumeration represents possible communication modes for the virtual serial port device.

Disconnected Virtual device is not attached to any real host device.

HostPipe Virtual device is attached to a host pipe.

HostDevice Virtual device is attached to a host device.

RawFile Virtual device is attached to a raw file.

6.42 ProcessInputFlag

Guest process input flags.

None No flag set.

EndOfFile End of file (input) reached.

6.43 ProcessorFeature

CPU features.

HWVirtEx

PAE

LongMode

NestedPaging

6.44 Scope

Scope of the operation.
A generic enumeration used in various methods to define the action or argument scope.

Global

Machine

Session

6.45 SessionState

Session state. This enumeration represents possible values of IMachine::sessionState and ISession::state attributes.

Null Null value (never used by the API).

Unlocked In IMachine::sessionState, this means that the machine is not locked for any sessions.

In ISession::state, this means that no machine is currently locked for this session.

Locked In IMachine::sessionState, this means that the machine is currently locked for a session, whose process identifier can then be found in the IMachine::sessionPid attribute.

In ISession::state, this means that a machine is currently locked for this session, and the mutable machine object can be found in the ISession::machine attribute (see IMachine::lockMachine() for details).

Spawning A new process is being spawned for the machine as a result of IMachine::launchVMProcess() call. This state also occurs as a short transient state during an IMachine::lockMachine() call.

Unlocking The session is being unlocked.

6.46 SessionType

Session type. This enumeration represents possible values of the ISession::type attribute.

Null Null value (never used by the API).

WriteLock Session has acquired an exclusive write lock on a machine using IMachine::lockMachine().

Remote Session has launched a VM process using IMachine::launchVMProcess()

Shared Session has obtained a link to another session using IMachine::lockMachine()

6.47 SettingsVersion

Settings version of VirtualBox settings files. This is written to the "version" attribute of the root "VirtualBox" element in the settings file XML and indicates which VirtualBox version wrote the file.

Null Null value, indicates invalid version.

v1_0 Legacy settings version, not currently supported.

v1_1 Legacy settings version, not currently supported.

v1_2 Legacy settings version, not currently supported.

v1_3pre Legacy settings version, not currently supported.

v1_3 Settings version "1.3", written by VirtualBox 2.0.12.

v1_4 Intermediate settings version, understood by VirtualBox 2.1.x.

v1_5 Intermediate settings version, understood by VirtualBox 2.1.x.

v1_6 Settings version "1.6", written by VirtualBox 2.1.4 (at least).

v1_7 Settings version "1.7", written by VirtualBox 2.2.x and 3.0.x.

v1_8 Intermediate settings version "1.8", understood by VirtualBox 3.1.x.

v1_9 Settings version "1.9", written by VirtualBox 3.1.x.

v1_10 Settings version "1.10", written by VirtualBox 3.2.x.

v1_11 Settings version "1.11", written by VirtualBox 4.0.x.

Future Settings version greater than "1.11", written by a future VirtualBox version.

6.48 StorageBus

The bus type of the storage controller (IDE, SATA, SCSI, SAS or Floppy); see IStorageController::bus.

Null `null` value. Never used by the API.

IDE

SATA

SCSI

Floppy

SAS

6.49 StorageControllerType

The exact variant of storage controller hardware presented to the guest; see IStorageController::controllerType.

Null `null` value. Never used by the API.

LsiLogic A SCSI controller of the LsiLogic variant.

BusLogic A SCSI controller of the BusLogic variant.

IntelAhci An Intel AHCI SATA controller; this is the only variant for SATA.

PIIX3 An IDE controller of the PIIX3 variant.

PIIX4 An IDE controller of the PIIX4 variant.

ICH6 An IDE controller of the ICH6 variant.

I82078 A floppy disk controller; this is the only variant for floppy drives.

LsiLogicSas A variant of the LsiLogic controller using SAS.

6.50 USBDeviceFilterAction

Actions for host USB device filters. See also: IHostUSBDeviceFilter, USBDeviceState

Null Null value (never used by the API).

Ignore Ignore the matched USB device.

Hold Hold the matched USB device.

6.51 USBDeviceState

USB device state. This enumeration represents all possible states of the USB device physically attached to the host computer regarding its state on the host computer and availability to guest computers (all currently running virtual machines).

Once a supported USB device is attached to the host, global USB filters (IHost::USBDeviceFilters[]) are activated. They can either ignore the device, or put it to USBDeviceState_Held state, or do nothing. Unless the device is ignored by global filters, filters of all currently running guests (IUSBController::deviceFilters[]) are activated that can put it to USBDeviceState_Captured state.

If the device was ignored by global filters, or didn't match any filters at all (including guest ones), it is handled by the host in a normal way. In this case, the device state is determined by the host and can be one of USBDeviceState_Unavailable, USBDeviceState_Busy or USBDeviceState_Available, depending on the current device usage.

Besides auto-capturing based on filters, the device can be manually captured by guests (IConsole::attachUSBDevice()) if its state is USBDeviceState_Busy, USBDeviceState_Available or USBDeviceState_Held.

> **Note:** Due to differences in USB stack implementations in Linux and Win32, states USBDeviceState_Busy and USBDeviceState_Unavailable are applicable only to the Linux version of the product. This also means that (IConsole::attachUSBDevice()) can only succeed on Win32 if the device state is USBDeviceState_Held.

See also: IHostUSBDevice, IHostUSBDeviceFilter

NotSupported Not supported by the VirtualBox server, not available to guests.

Unavailable Being used by the host computer exclusively, not available to guests.

Busy Being used by the host computer, potentially available to guests.

Available Not used by the host computer, available to guests (the host computer can also start using the device at any time).

Held Held by the VirtualBox server (ignored by the host computer), available to guests.

Captured Captured by one of the guest computers, not available to anybody else.

6.52 VBoxEventType

Type of an event. See IEvent for an introduction to VirtualBox event handling.

Invalid Invalid event, must be first.

Any Wildcard for all events. Events of this type are never delivered, and only used in registerListener() call to simplify registration.

Vetoable Wildcard for all vetoable events. Events of this type are never delivered, and only used in registerListener() call to simplify registration.

MachineEvent Wildcard for all machine events. Events of this type are never delivered, and only used in registerListener() call to simplify registration.

SnapshotEvent Wildcard for all snapshot events. Events of this type are never delivered, and only used in registerListener() call to simplify registration.

InputEvent Wildcard for all input device (keyboard, mouse) events. Events of this type are never delivered, and only used in registerListener() call to simplify registration.

LastWildcard Last wildcard.

OnMachineStateChanged See IMachineStateChangedEvent.

OnMachineDataChanged See IMachineDataChangedEvent.

OnExtraDataChanged See IExtraDataChangedEvent.

OnExtraDataCanChange See IExtraDataCanChangeEvent.

OnMediumRegistered See IMediumRegisteredEvent.

OnMachineRegistered See IMachineRegisteredEvent.

OnSessionStateChanged See ISessionStateChangedEvent.

OnSnapshotTaken See ISnapshotTakenEvent.

OnSnapshotDeleted See ISnapshotDeletedEvent.

OnSnapshotChanged See ISnapshotChangedEvent.

OnGuestPropertyChanged See IGuestPropertyChangedEvent.

OnMousePointerShapeChanged See IMousePointerShapeChangedEvent.

OnMouseCapabilityChanged See IMouseCapabilityChangedEvent.

OnKeyboardLedsChanged See IKeyboardLedsChangedEvent.

OnStateChanged See IStateChangedEvent.

OnAdditionsStateChanged See IAdditionsStateChangedEvent.

OnNetworkAdapterChanged See INetworkAdapterChangedEvent.

OnSerialPortChanged See ISerialPortChangedEvent.

OnParallelPortChanged See IParallelPortChangedEvent.

OnStorageControllerChanged See IStorageControllerChangedEvent.

OnMediumChanged See IMediumChangedEvent.

OnVRDEServerChanged See IVRDEServerChangedEvent.

OnUSBControllerChanged See IUSBControllerChangedEvent.

OnUSBDeviceStateChanged See IUSBDeviceStateChangedEvent.

OnSharedFolderChanged See ISharedFolderChangedEvent.

OnRuntimeError See IRuntimeErrorEvent.

OnCanShowWindow See ICanShowWindowEvent.

OnShowWindow See IShowWindowEvent.

OnCPUChanged See ICPUChangedEvent.

OnVRDEServerInfoChanged See IVRDEServerInfoChangedEvent.

OnEventSourceChanged See IEventSourceChangedEvent.

OnCPUExecutionCapChanged See ICPUExecutionCapChangedEvent.

OnGuestKeyboard See IGuestKeyboardEvent.

OnGuestMouse See IGuestMouseEvent.

OnNATRedirect See INATRedirectEvent.

OnHostPciDevicePlug See IHostPciDevicePlugEvent.

OnVBoxSVCAvailabilityChanged See IVBoxSVCAvailabilityChangedEvent.

OnBandwidthGroupChanged See IBandwidthGroupChangedEvent.

OnGuestMonitorChanged See IGuestMonitorChangedEvent.

Last Must be last event, used for iterations and structures relying on numerical event values.

6.53 VFSFileType

File types known by VFSExplorer.

Unknown

Fifo

DevChar

Directory

DevBlock

File

SymLink

Socket

WhiteOut

6.54 VFSType

Virtual file systems supported by VFSExplorer.

File

Cloud

S3

WebDav

6.55 VirtualSystemDescriptionType

Used with IVirtualSystemDescription to describe the type of a configuration value.

Ignore

OS

Name

Product

Vendor

Version

ProductUrl

VendorUrl

Description

License

Miscellaneous

CPU

Memory

HardDiskControllerIDE

HardDiskControllerSATA

HardDiskControllerSCSI

HardDiskControllerSAS

HardDiskImage

Floppy

CDROM

NetworkAdapter

USBController

SoundCard

6.56 VirtualSystemDescriptionValueType

Used with IVirtualSystemDescription::getValuesByType() to describe the value type to fetch.

Reference

Original

Auto

ExtraConfig

7 Host-Guest Communication Manager

The VirtualBox Host-Guest Communication Manager (HGCM) allows a guest application or a guest driver to call a host shared library. The following features of VirtualBox are implemented using HGCM:

- Shared Folders
- Shared Clipboard
- Guest configuration interface

The shared library contains a so called HGCM service. The guest HGCM clients establish connections to the service to call it. When calling a HGCM service the client supplies a function code and a number of parameters for the function.

7.1 Virtual hardware implementation

HGCM uses the VMM virtual PCI device to exchange data between the guest and the host. The guest always acts as an initiator of requests. A request is constructed in the guest physical memory, which must be locked by the guest. The physical address is passed to the VMM device using a 32 bit `out edx, eax` instruction. The physical memory must be allocated below 4GB by 64 bit guests.

The host parses the request header and data and queues the request for a host HGCM service. The guest continues execution and usually waits on a HGCM event semaphore.

When the request has been processed by the HGCM service, the VMM device sets the completion flag in the request header, sets the HGCM event and raises an IRQ for the guest. The IRQ handler signals the HGCM event semaphore and all HGCM callers check the completion flag in the corresponding request header. If the flag is set, the request is considered completed.

7.2 Protocol specification

The HGCM protocol definitions are contained in the `VBox/VBoxGuest.h`

7.2.1 Request header

HGCM request structures contains a generic header (VMMDevHGCMRequestHeader):

Name	Description
size	Size of the entire request.
version	Version of the header, must be set to `0x10001`.
type	Type of the request.
rc	HGCM return code, which will be set by the VMM device.
reserved1	A reserved field 1.
reserved2	A reserved field 2.
flags	HGCM flags, set by the VMM device.
result	The HGCM result code, set by the VMM device.

Note:

- All fields are 32 bit.
- Fields from `size` to `reserved2` are a standard VMM device request header, which is used for other interfaces as well.

The **type** field indicates the type of the HGCM request:

Name (decimal value)	Description
VMMDevReq_HGCMConnect (60)	Connect to a HGCM service.
VMMDevReq_HGCMDisconnect (61)	Disconnect from the service.
VMMDevReq_HGCMCall32 (62)	Call a HGCM function using the 32 bit interface.
VMMDevReq_HGCMCall64 (63)	Call a HGCM function using the 64 bit interface.
VMMDevReq_HGCMCancel (64)	Cancel a HGCM request currently being processed by a host HGCM service.

The **flags** field may contain:

Name (hexadecimal value)	Description
VBOX_HGCM_REQ_DONE (0x00000001)	The request has been processed by the host service.
VBOX_HGCM_REQ_CANCELLED (0x00000002)	This request was cancelled.

7.2.2 Connect

The connection request must be issued by the guest HGCM client before it can call the HGCM service (VMMDevHGCMConnect):

Name	Description
header	The generic HGCM request header with type equal to VMMDevReq_HGCMConnect (60).
type	The type of the service location information (32 bit).
location	The service location information (128 bytes).
clientId	The client identifier assigned to the connecting client by the HGCM subsystem (32 bit).

The **type** field tells the HGCM how to look for the requested service:

Name (hexadecimal value)	Description
VMMDevHGCMLoc_LocalHost (0x1)	The requested service is a shared library located on the host and the location information contains the library name.
VMMDevHGCMLoc_LocalHost_Existing (0x2)	The requested service is a preloaded one and the location information contains the service name.

Note: Currently preloaded HGCM services are hard-coded in VirtualBox:

- VBoxSharedFolders
- VBoxSharedClipboard
- VBoxGuestPropSvc
- VBoxSharedOpenGL

There is no difference between both types of HGCM services, only the location mechanism is different.

The client identifier is returned by the host and must be used in all subsequent requests by the client.

7.2.3 Disconnect

This request disconnects the client and makes the client identifier invalid (VMMDevHGCMDisconnect):

Name	Description
header	The generic HGCM request header with type equal to VMMDevReq_HGCMDisconnect (61).
clientId	The client identifier previously returned by the connect request (32 bit).

7.2.4 Call32 and Call64

Calls the HGCM service entry point (VMMDevHGCMCall) using 32 bit or 64 bit addresses:

Name	Description
header	The generic HGCM request header with type equal to either VMMDevReq_HGCMCall32 (62) or VMMDevReq_HGCMCall64 (63).
clientId	The client identifier previously returned by the connect request (32 bit).
function	The function code to be processed by the service (32 bit).
cParms	The number of following parameters (32 bit). This value is 0 if the function requires no parameters.
parms	An array of parameter description structures (HGCMFunctionParameter32 or HGCMFunctionParameter64).

The 32 bit parameter description (HGCMFunctionParameter32) consists of 32 bit type field and 8 bytes of an opaque value, so 12 bytes in total. The 64 bit variant (HGCMFunctionParameter64) consists of the type and 12 bytes of a value, so 16 bytes in total.

Type	Format of the value
VMMDevHGCMParm-Type_32bit (1)	A 32 bit value.
VMMDevHGCMParm-Type_64bit (2)	A 64 bit value.
VMMDevHGCMParm-Type_PhysAddr (3)	A 32 bit size followed by a 32 bit or 64 bit guest physical address.
VMMDevHGCMParm-Type_LinAddr (4)	A 32 bit size followed by a 32 bit or 64 bit guest linear address. The buffer is used both for guest to host and for host to guest data.
VMMDevHGCMParm-Type_LinAddr_In (5)	Same as VMMDevHGCMParmType_LinAddr but the buffer is used only for host to guest data.
VMMDevHGCMParm-Type_LinAddr_Out (6)	Same as VMMDevHGCMParmType_LinAddr but the buffer is used only for guest to host data.
VMMDevHGCMParm-Type_LinAddr_Locked (7)	Same as VMMDevHGCMParmType_LinAddr but the buffer is already locked by the guest.
VMMDevHGCMParm-Type_LinAddr_Locked_In (1)	Same as VMMDevHGCMParmType_LinAddr_In but the buffer is already locked by the guest.
VMMDevHGCMParm-Type_LinAddr_Locked_Out (1)	Same as VMMDevHGCMParmType_LinAddr_Out but the buffer is already locked by the guest.

The

7.2.5 Cancel

This request cancels a call request (VMMDevHGCMCancel):

Name	Description
header	The generic HGCM request header with type equal to VMMDevReq_HGCMCancel (64).

7.3 Guest software interface

The guest HGCM clients can call HGCM services from both drivers and applications.

7.3.1 The guest driver interface

The driver interface is implemented in the VirtualBox guest additions driver (VBoxGuest), which works with the VMM virtual device. Drivers must use the VBox Guest Library (VBGL), which provides an API for HGCM clients (`VBox/VBoxGuestLib.h` and `VBox/VBoxGuest.h`).

```
DECLVBGL(int) VbglHGCMConnect (VBGLHGCMHANDLE *pHandle, VBoxGuestHGCMConnectInfo *pData);
```

Connects to the service:

```
VBoxGuestHGCMConnectInfo data;
```

```
    memset (&data, sizeof (VBoxGuestHGCMConnectInfo));

    data.result   = VINF_SUCCESS;
    data.Loc.type = VMMDevHGCMLoc_LocalHost_Existing;
    strcpy (data.Loc.u.host.achName, "VBoxSharedFolders");

    rc = VbglHGCMConnect (&handle, &data);

    if (RT_SUCCESS (rc))
    {
        rc = data.result;
    }

    if (RT_SUCCESS (rc))
    {
        /* Get the assigned client identifier. */
        ulClientID = data.u32ClientID;
    }
```

```
DECLVBGL(int) VbglHGCMDisconnect (VBGLHGCMHANDLE handle, VBoxGuestHGCMDisconnectInfo *pData);
```

Disconnects from the service.

```
    VBoxGuestHGCMDisconnectInfo data;

    RtlZeroMemory (&data, sizeof (VBoxGuestHGCMDisconnectInfo));

    data.result      = VINF_SUCCESS;
    data.u32ClientID = ulClientID;

    rc = VbglHGCMDisconnect (handle, &data);
```

```
DECLVBGL(int) VbglHGCMCall (VBGLHGCMHANDLE handle, VBoxGuestHGCMCallInfo *pData, uint32_t cbData);
```

Calls a function in the service.

```
typedef struct _VBoxSFRead
{
    VBoxGuestHGCMCallInfo callInfo;

    /** pointer, in: SHFLROOT
     * Root handle of the mapping which name is queried.
     */
    HGCMFunctionParameter root;

    /** value64, in:
     * SHFLHANDLE of object to read from.
     */
    HGCMFunctionParameter handle;

    /** value64, in:
     * Offset to read from.
     */
    HGCMFunctionParameter offset;

    /** value64, in/out:
     * Bytes to read/How many were read.
     */
    HGCMFunctionParameter cb;
```

```
    /** pointer, out:
     * Buffer to place data to.
     */
    HGCMFunctionParameter buffer;

} VBoxSFRead;

/** Number of parameters */
#define SHFL_CPARMS_READ (5)

...

    VBoxSFRead data;

    /* The call information. */
    data.callInfo.result      = VINF_SUCCESS;     /* Will be returned by HGCM. */
    data.callInfo.u32ClientID = ulClientID;       /* Client identifier. */
    data.callInfo.u32Function = SHFL_FN_READ;     /* The function code. */
    data.callInfo.cParms      = SHFL_CPARMS_READ; /* Number of parameters. */

    /* Initialize parameters. */
    data.root.type                      = VMMDevHGCMParmType_32bit;
    data.root.u.value32                 = pMap->root;

    data.handle.type                    = VMMDevHGCMParmType_64bit;
    data.handle.u.value64               = hFile;

    data.offset.type                    = VMMDevHGCMParmType_64bit;
    data.offset.u.value64               = offset;

    data.cb.type                        = VMMDevHGCMParmType_32bit;
    data.cb.u.value32                   = *pcbBuffer;

    data.buffer.type                    = VMMDevHGCMParmType_LinAddr_Out;
    data.buffer.u.Pointer.size          = *pcbBuffer;
    data.buffer.u.Pointer.u.linearAddr  = (uintptr_t)pBuffer;

    rc = VbglHGCMCall (handle, &data.callInfo, sizeof (data));

    if (RT_SUCCESS (rc))
    {
        rc = data.callInfo.result;
        *pcbBuffer = data.cb.u.value32; /* This is returned by the HGCM service. */
    }
```

7.3.2 Guest application interface

Applications call the VirtualBox Guest Additions driver to utilize the HGCM interface. There are IOCTL's which correspond to the `Vbgl*` functions:

- `VBOXGUEST_IOCTL_HGCM_CONNECT`
- `VBOXGUEST_IOCTL_HGCM_DISCONNECT`
- `VBOXGUEST_IOCTL_HGCM_CALL`

These IOCTL's get the same input buffer as `VbglHGCM*` functions and the output buffer has the same format as the input buffer. The same address can be used as the input and output buffers.

For example see the guest part of shared clipboard, which runs as an application and uses the HGCM interface.

7.4 HGCM Service Implementation

The HGCM service is a shared library with a specific set of entry points. The library must export the `VBoxHGCMSvcLoad` entry point:

```
extern "C" DECLCALLBACK(DECLEXPORT(int)) VBoxHGCMSvcLoad (VBOXHGCMSVCFNTABLE *ptable)
```

The service must check the `ptable->cbSize` and `ptable->u32Version` fields of the input structure and fill the remaining fields with function pointers of entry points and the size of the required client buffer size.

The HGCM service gets a dedicated thread, which calls service entry points synchronously, that is the service will be called again only when a previous call has returned. However, the guest calls can be processed asynchronously. The service must call a completion callback when the operation is actually completed. The callback can be issued from another thread as well.

Service entry points are listed in the `VBox/hgcmsvc.h` in the `VBOXHGCMSVCFNTABLE` structure.

Entry	Description
pfnUnload	The service is being unloaded.
pfnConnect	A client `u32ClientID` is connected to the service. The `pvClient` parameter points to an allocated memory buffer which can be used by the service to store the client information.
pfnDisconnect	A client is being disconnected.
pfnCall	A guest client calls a service function. The `callHandle` must be used in the VBOXHGCMSVCHELPERS::pfnCallComplete callback when the call has been processed.
pfnHostCall	Called by the VirtualBox host components to perform functions which should be not accessible by the guest. Usually this entry point is used by VirtualBox to configure the service.
pfnSaveState	The VM state is being saved and the service must save relevant information using the SSM API (`VBox/ssm.h`).
pfnLoadState	The VM is being restored from the saved state and the service must load the saved information and be able to continue operations from the saved state.

8 RDP Web Control

The VirtualBox *RDP Web Control* (RDPWeb) provides remote access to a running VM. RDPWeb is a RDP (Remote Desktop Protocol) client based on Flash technology and can be used from a Web browser with a Flash plugin.

8.1 RDPWeb features

RDPWeb is embedded into a Web page and can connect to VRDP server in order to displays the VM screen and pass keyboard and mouse events to the VM.

8.2 RDPWeb reference

RDPWeb consists of two required components:

- Flash movie `RDPClientUI.swf`
- JavaScript helpers `webclient.js`

The VirtualBox SDK contains sample HTML code including:

- JavaScript library for embedding Flash content `SWFObject.js`
- Sample HTML page `webclient3.html`

8.2.1 RDPWeb functions

`RDPClientUI.swf` and `webclient.js` work with each other. JavaScript code is responsible for a proper SWF initialization, delivering mouse events to the SWF and processing resize requests from the SWF. On the other hand, the SWF contains a few JavaScript callable methods, which are used both from `webclient.js` and the user HTML page.

8.2.1.1 JavaScript functions

`webclient.js` contains helper functions. In the following table ElementId refers to an HTML element name or attribute, and Element to the HTML element itself. HTML code

```
<div id="FlashRDP">
</div>
```

would have ElementId equal to FlashRDP and Element equal to the div element.

- `RDPWebClient.embedSWF(SWFFileName, ElementId)`
  Uses SWFObject library to replace the HTML element with the Flash movie.
- `RDPWebClient.isRDPWebControlById(ElementId)`
  Returns true if the given id refers to a RDPWeb Flash element.

- `RDPWebClient.isRDPWebControlByElement(Element)`

  Returns true if the given element is a RDPWeb Flash element.

- `RDPWebClient.getFlashById(ElementId)`

  Returns an element, which is referenced by the given id. This function will try to resolve any element, event if it is not a Flash movie.

8.2.1.2 Flash methods callable from JavaScript

`RDPWebClienUI.swf` methods can be called directly from JavaScript code on a HTML page.

- getProperty(Name)
- setProperty(Name)
- connect()
- disconnect()
- keyboardSendCAD()

8.2.1.3 Flash JavaScript callbacks

`RDPWebClienUI.swf` calls JavaScript functions provided by the HTML page.

8.2.2 Embedding RDPWeb in an HTML page

It is necessary to include `webclient.js` helper script. If SWFObject library is used, the `swfobject.js` must be also included and RDPWeb flash content can be embedded to a Web page using dynamic HTML. The HTML must include a "placeholder", which consists of 2 `div` elements.

8.3 RDPWeb change log

8.3.1 Version 1.2.28

- `keyboardLayout`, `keyboardLayouts`, `UUID` properties.
- Support for German keyboard layout on the client.
- Rebranding to Oracle.

8.3.2 Version 1.1.26

- `webclient.js` is a part of the distribution package.
- `lastError` property.
- `keyboardSendScancodes` and `keyboardSendCAD` methods.

8.3.3 Version 1.0.24

- Initial release.

9 VirtualBox external authentication modules

VirtualBox supports arbitrary external modules to perform authentication. The module is used when the authentication method is set to "external" for a particular VM VRDE access and the library was specified with `VBoxManage setproperty vrdeauthlibrary`. Web service also use the authentication module which was specified with `VBoxManage setproperty websrvauthlibrary`.

This library will be loaded by the VM or web service process on demand, i.e. when the first remote desktop connection is made by a client or when a client that wants to use the web service logs on.

External authentication is the most flexible as the external handler can both choose to grant access to everyone (like the "null" authentication method would) and delegate the request to the guest authentication component. When delegating the request to the guest component, the handler will still be called afterwards with the option to override the result.

An authentication library is required to implement exactly one entry point:

```
#include "VBoxAuth.h"

/**
 * Authentication library entry point.
 *
 * Parameters:
 *
 *   szCaller         The name of the component which calls the library (UTF8).
 *   pUuid            Pointer to the UUID of the accessed virtual machine. Can be NULL.
 *   guestJudgement   Result of the guest authentication.
 *   szUser           User name passed in by the client (UTF8).
 *   szPassword       Password passed in by the client (UTF8).
 *   szDomain         Domain passed in by the client (UTF8).
 *   fLogon           Boolean flag. Indicates whether the entry point is called
 *                    for a client logon or the client disconnect.
 *   clientId         Server side unique identifier of the client.
 *
 * Return code:
 *
 *   AuthResultAccessDenied    Client access has been denied.
 *   AuthResultAccessGranted   Client has the right to use the
 *                             virtual machine.
 *   AuthResultDelegateToGuest Guest operating system must
 *                             authenticate the client and the
 *                             library must be called again with
 *                             the result of the guest
 *                             authentication.
 *
 * Note: When 'fLogon' is 0, only pszCaller, pUuid and clientId are valid and the return
 *       code is ignored.
 */
AuthResult AUTHCALL AuthEntry(
    const char *szCaller,
    PAUTHUUID pUuid,
    AuthGuestJudgement guestJudgement,
    const char *szUser,
    const char *szPassword
    const char *szDomain
    int fLogon,
```

```
    unsigned clientId)
{
    /* Process request against your authentication source of choice. */
    // if (authSucceeded(...))
    //     return AuthResultAccessGranted;
    return AuthResultAccessDenied;
}
```

A note regarding the UUID implementation of the `pUuid` argument: VirtualBox uses a consistent binary representation of UUIDs on all platforms. For this reason the integer fields comprising the UUID are stored as little endian values. If you want to pass such UUIDs to code which assumes that the integer fields are big endian (often also called network byte order), you need to adjust the contents of the UUID to e.g. achieve the same string representation. The required changes are:

- reverse the order of byte 0, 1, 2 and 3
- reverse the order of byte 4 and 5
- reverse the order of byte 6 and 7.

Using this conversion you will get identical results when converting the binary UUID to the string representation.

The `guestJudgement` argument contains information about the guest authentication status. For the first call, it is always set to `AuthGuestNotAsked`. In case the `AuthEntry` function returns `AuthResultDelegateToGuest`, a guest authentication will be attempted and another call to the `AuthEntry` is made with its result. This can be either granted / denied or no judgement (the guest component chose for whatever reason to not make a decision). In case there is a problem with the guest authentication module (e.g. the Additions are not installed or not running or the guest did not respond within a timeout), the “not reacted” status will be returned.

10 Using Java API

10.1 Introduction

VirtualBox can be controlled by a Java API, both locally (COM/XPCOM) and from remote (SOAP) clients. As with the Python bindings, a generic glue layer tries to hide all platform differences, allowing for source and binary compatibility on different platforms.

10.2 Requirements

To use the Java bindings, there are certain requirements depending on the platform. First of all, you need JDK 1.5 (Java 5) or later. Also please make sure that the version of the VirtualBox API .jar file exactly matches the version of VirtualBox you use. To avoid confusion, the VirtualBox API provides versioning in the Java package name, e.g. the package is named `org.virtualbox_3_2` for VirtualBox version 3.2.

- **XPCOM:** - for all platforms, but Microsoft Windows. A Java bridge based on JavaXPCOM is shipped with VirtualBox. The classpath must contain `vboxjxpcom.jar` and the `vbox.home` property must be set to location where the VirtualBox binaries are. Please make sure that the JVM bitness matches bitness of VirtualBox you use as the XPCOM bridge relies on native libraries.

  Start your application like this:

  ```
  java -cp vboxjxpcom.jar -Dvbox.home=/opt/virtualbox MyProgram
  ```

- **COM:** - for Microsoft Windows. We rely on `Jacob` - a generic Java to COM bridge - which has to be installed seperately. See http://sourceforge.net/projects/jacob-project/ for installation instructions. Also, the VirtualBox provided `vboxjmscom.jar` must be in the class path.

  Start your application like this:

  ```
  java -cp vboxjmscom.jar;c:\jacob\jacob.jar -Djava.library.path=c:\jacob MyProgram
  ```

- **SOAP** - all platforms. Java 6 is required, as it comes with builtin support for SOAP via the JAX-WS library. Also, the VirtualBox provided `vbojws.jar` must be in the class path. In the SOAP case it's possible to create several VirtualBoxManager instances to communicate with multiple VirtualBox hosts.

  Start your application like this:

  ```
  java -cp vboxjws.jar MyProgram
  ```

Exception handling is also generalized by the generic glue layer, so that all methods could throw `VBoxException` containing human-readable text message (see `getMessage()` method) along with wrapped original exception (see `getWrapped()` method).

10.3 Example

This example shows a simple use case of the Java API. Differences for SOAP vs. local version are minimal, and limited to the connection setup phase (see `ws` variable). In the SOAP case it's possible to create several VirtualBoxManager instances to communicate with multiple VirtualBox hosts.

```
import org.virtualbox_3_3.*;
....
VirtualBoxManager mgr = VirtualBoxManager.createInstance(null);
boolean ws = false; // or true, if we need the SOAP version
if (ws)

    String url = "http://myhost:18034";
    String user = "test";
    String passwd = "test";
    mgr.connect(url, user, passwd);

IVirtualBox vbox = mgr.getVBox();
System.out.println("VirtualBox version: " + vbox.getVersion() + "\n");
// get first VM name
String m =  vbox.getMachines().get(0).getName();
System.out.println("\nAttempting to start VM '" + m + "'");
// start it
mgr.startVm(m, null, 7000);

if (ws)
    mgr.disconnect();

mgr.cleanup();
```

For more a complete example, see `TestVBox.java`, shipped with the SDK.

11 License information

The sample code files shipped with the SDK are generally licensed liberally to make it easy for anyone to use this code for their own application code.

The Java files under `bindings/webservice/java/jax-ws/` (library files for the object-oriented web service) are, by contrast, licensed under the GNU Lesser General Public License (LGPL) V2.1.

See `sdk/bindings/webservice/java/jax-ws/src/COPYING.LIB` for the full text of the LGPL 2.1.

When in doubt, please refer to the individual source code files shipped with this SDK.

12 Main API change log

Generally, VirtualBox will maintain API compatibility within a major release; a major release occurs when the first or the second of the three version components of VirtualBox change (that is, in the x.y.z scheme, a major release is one where x or y change, but not when only z changes).

In other words, updates like those from 2.0.0 to 2.0.2 will not come with API breakages.

Migration between major releases most likely will lead to API breakage, so please make sure you updated code accordingly. The OOWS Java wrappers enforce that mechanism by putting VirtualBox classes into version-specific packages such as `org.virtualbox_2_2`. This approach allows for connecting to multiple VirtualBox versions simultaneously from the same Java application.

The following sections list incompatible changes that the Main API underwent since the original release of this SDK Reference with VirtualBox 2.0. A change is deemed "incompatible" only if it breaks existing client code (e.g. changes in method parameter lists, renamed or removed interfaces and similar). In other words, the list does not contain new interfaces, methods or attributes or other changes that do not affect existing client code.

12.1 Incompatible API changes with version 4.0

- A new Java glue layer replacing the previous OOWS JAX-WS bindings was introduced. The new library allows for uniform code targeting both local (COM/XPCOM) and remote (SOAP) transports. Now, instead of `IWebsessionManager`, the new class `VirtualBoxManager` must be used. See Java API chapter for details.

- The confusingly named and impractical session APIs were changed. In existing client code, the following changes need to be made:

  - Replace any `IVirtualBox::openSession(uuidMachine, ...)` API call with the machine's IMachine::lockMachine() call and a `LockType.Write` argument. The functionality is unchanged, but instead of "opening a direct session on a machine" all documentation now refers to "obtaining a write lock on a machine for the client session".

  - Similarly, replace any `IVirtualBox::openExistingSession(uuidMachine, ...)` call with the machine's IMachine::lockMachine() call and a `LockType.Shared` argument. Whereas it was previously impossible to connect a client session to a running VM process in a race-free manner, the new API will atomically either write-lock the machine for the current session or establish a remote link to an existing session. Existing client code which tried calling both `openSession()` and `openExistingSession()` can now use this one call instead.

  - Third, replace any `IVirtualBox::openRemoteSession(uuidMachine, ...)` call with the machine's IMachine::launchVMProcess() call. The functionality is unchanged.

  - The SessionState enum was adjusted accordingly: "Open" is now "Locked", "Closed" is now "Unlocked", "Closing" is now "Unlocking".

- Virtual machines created with VirtualBox 4.0 or later no longer register their media in the global media registry in the `VirtualBox.xml` file. Instead, such machines list all their

media in their own machine XML files. As a result, a number of media-related APIs had to be modified again.

  - Neither IVirtualBox::createHardDisk() nor IVirtualBox::openMedium() register media automatically any more.
  - IMachine::attachDevice() and IMachine::mountMedium() now take an IMedium object instead of a UUID as an argument. It is these two calls which add media to a registry now (either a machine registry for machines created with VirtualBox 4.0 or later or the global registry otherwise). As a consequence, if a medium is opened but never attached to a machine, it is no longer added to any registry any more.
  - To reduce code duplication, the APIs IVirtualBox::findHardDisk(), getHardDisk(), findDVDImage(), getDVDImage(), findFloppyImage() and getFloppyImage() have all been merged into IVirtualBox::findMedium(), and IVirtualBox::openHardDisk(), openDVDImage() and openFloppyImage() have all been merged into IVirtualBox::openMedium().
  - The rare use case of changing the UUID and parent UUID of a medium previously handled by `openHardDisk()` is now in a separate IMedium::setIDs method.
  - `ISystemProperties::get/setDefaultHardDiskFolder()` have been removed since disk images are now by default placed in each machine's folder.
  - The ISystemProperties::infoVDSize attribute replaces the `getMaxVDISize()` API call; this now uses bytes instead of megabytes.

- Machine management APIs were enhanced as follows:
  - IVirtualBox::createMachine() is no longer restricted to creating machines in the default "Machines" folder, but can now create machines at arbitrary locations. For this to work, the parameter list had to be changed.
  - The long-deprecated `IVirtualBox::createLegacyMachine()` API has been removed.
  - To reduce code duplication and for consistency with the aforementioned media APIs, `IVirtualBox::getMachine()` has been merged with IVirtualBox::findMachine(), and `IMachine::getSnapshot()` has been merged with IMachine::findSnapshot().
  - `IVirtualBox::unregisterMachine()` was replaced with IMachine::unregister() with additional functionality for cleaning up machine files.
  - `IConsole::forgetSavedState` has been renamed to IConsole::discardSavedState().

- All event callbacks APIs were replaced with a new, generic event mechanism that can be used both locally (COM, XPCOM) and remotely (web services). Also, the new mechanism is usable from scripting languages and a local Java. See events for details. The new concept will require changes to all clients that used event callbacks.

- `additionsActive()` was replaced with additionsRunLevel() and getAdditionsStatus() in order to support a more detailed status of the current Guest Additions loading/readiness state. IGuest::additionsVersion() no longer returns the Guest Additions interface version but the installed Guest Additions version and revision in form of `3.3.0r12345`.

- To address shared folders auto-mounting support, the following APIs were extended to require an additional `automount` parameter:
  - IVirtualBox::createSharedFolder()
  - IMachine::createSharedFolder()
  - IConsole::createSharedFolder()

  Also, a new property named `autoMount` was added to the ISharedFolder interface.

- The appliance (OVF) APIs were enhanced as follows:
  - IMachine::export() received an extra parameter `location`, which is used to decide for the disk naming.
  - IAppliance::write() received an extra parameter `manifest`, which can suppress creating the manifest file on export.
  - IVFSExplorer::entryList() received two extra parameters `sizes` and `modes`, which contains the sizes (in bytes) and the file access modes (in octal form) of the returned files.
- Support for remote desktop access to virtual machines has been cleaned up to allow third party implementations of the remote desktop server. This is called the VirtualBox Remote Desktop Extension (VRDE) and can be added to VirtualBox by installing the corresponding extension package; see the VirtualBox User Manual for details.

  The following API changes were made to support the VRDE interface:
  - `IVRDPServer` has been renamed to IVRDEServer.
  - `IRemoteDisplayInfo` has been renamed to IVRDEServerInfo.
  - IMachine::VRDEServer replaces `VRDPServer`.
  - IConsole::VRDEServerInfo replaces `RemoteDisplayInfo`.
  - ISystemProperties::VRDEAuthLibrary replaces `RemoteDisplayAuthLibrary`.
  - The following methods have been implemented in `IVRDEServer` to support generic VRDE properties:
    - IVRDEServer::setVRDEProperty
    - IVRDEServer::getVRDEProperty
    - IVRDEServer::VRDEProperties

    A few implementation-specific attributes of the old `IVRDPServer` interface have been removed and replaced with properties:
    - `IVRDPServer::Ports` has been replaced with the `"TCP/Ports"` property. The property value is a string, which contains a comma-separated list of ports or ranges of ports. Use a dash between two port numbers to specify a range. Example: `"5000,5010-5012"`
    - `IVRDPServer::NetAddress` has been replaced with the `"TCP/Address"` property. The property value is an IP address string. Example: `"127.0.0.1"`
    - `IVRDPServer::VideoChannel` has been replaced with the `"VideoChannel/Enabled"` property. The property value is either `"true"` or `"false"`
    - `IVRDPServer::VideoChannelQuality` has been replaced with the `"VideoChannel/Quality"` property. The property value is a string which contain a decimal number in range 10..100. Invalid values are ignored and the quality is set to the default value 75. Example: `"50"`
- The VirtualBox external authentication module interface has been updated and made more generic. Because of that, `VRDPAuthType` enumeration has been renamed to AuthType.

12.2 Incompatible API changes with version 3.2

- The following interfaces were renamed for consistency:
  - IMachine::getCpuProperty() is now IMachine::getCPUProperty();

  - IMachine::setCpuProperty() is now IMachine::setCPUProperty();
  - IMachine::getCpuIdLeaf() is now IMachine::getCPUIDLeaf();
  - IMachine::setCpuIdLeaf() is now IMachine::setCPUIDLeaf();
  - IMachine::removeCpuIdLeaf() is now IMachine::removeCPUIDLeaf();
  - IMachine::removeAllCpuIdLeafs() is now IMachine::removeAllCPUIDLeaves();
  - the CpuPropertyType enum is now CPUPropertyType.
  - IVirtualBoxCallback::onSnapshotDiscarded() is now IVirtualBoxCallback::onSnapshotDeleted.

- When creating a VM configuration with IVirtualBox::createMachine) it is now possible to ignore existing configuration files which would previously have caused a failure. For this the `override` parameter was added.
- Deleting snapshots via IConsole::deleteSnapshot() is now possible while the associated VM is running in almost all cases. The API is unchanged, but client code that verifies machine states to determine whether snapshots can be deleted may need to be adjusted.
- The IoBackendType enumeration was replaced with a boolean flag (see IStorageController::useHostIOCache).
- To address multi-monitor support, the following APIs were extended to require an additional `screenId` parameter:
  - IMachine::querySavedThumbnailSize()
  - IMachine::readSavedThumbnailToArray()
  - IMachine::querySavedScreenshotPNGSize()
  - IMachine::readSavedScreenshotPNGToArray()
- The `shape` parameter of IConsoleCallback::onMousePointerShapeChange was changed from a implementation-specific pointer to a safearray, enabling scripting languages to process pointer shapes.

12.3 Incompatible API changes with version 3.1

- Due to the new flexibility in medium attachments that was introduced with version 3.1 (in particular, full flexibility with attaching CD/DVD drives to arbitrary controllers), we seized the opportunity to rework all interfaces dealing with storage media to make the API more flexible as well as logical. The IStorageController, IMedium, IMediumAttachment and, IMachine interfaces were affected the most. Existing code using them to configure storage and media needs to be carefully checked.

  All media (hard disks, floppies and CDs/DVDs) are now uniformly handled through the IMedium interface. The device-specific interfaces (`IHardDisk`, `IDVDImage`, `IHostDVDDrive`, `IFloppyImage` and `IHostFloppyDrive`) have been merged into IMedium; CD/DVD and floppy media no longer need special treatment. The device type of a medium determines in which context it can be used. Some functionality was moved to the other storage-related interfaces.

  `IMachine::attachHardDisk` and similar methods have been renamed and generalized to deal with any type of drive and medium. IMachine::attachDevice() is the API method for adding any drive to a storage controller. The floppy and DVD/CD drives are no longer handled specially, and that means you can have more than one of them. As before, drives can only be changed while the VM is powered off. Mounting (or unmounting) removable media at runtime is possible with IMachine::mountMedium().

  Newly created virtual machines have no storage controllers associated with them. Even the IDE Controller needs to be created explicitly. The floppy controller is now visible as

a separate controller, with a new storage bus type. For each storage bus type you can query the device types which can be attached, so that it is not necessary to hardcode any attachment rules.

This required matching changes e.g. in the callback interfaces (the medium specific change notification was replaced by a generic medium change notification) and removing associated enums (e.g. `DriveState`). In many places the incorrect use of the plural form "media" was replaced by "medium", to improve consistency.

- Reading the IMedium::state attribute no longer automatically performs an accessibility check; a new method IMedium::refreshState() does this. The attribute only returns the state any more.

- There were substantial changes related to snapshots, triggered by the "branched snapshots" functionality introduced with version 3.1. IConsole::discardSnapshot was renamed to IConsole::deleteSnapshot(). IConsole::discardCurrentState and IConsole::discardCurrentSnapshotAndState were removed; corresponding new functionality is in IConsole::restoreSnapshot(). Also, when IConsole::takeSnapshot() is called on a running virtual machine, a live snapshot will be created. The old behavior was to temporarily pause the virtual machine while creating an online snapshot.

- The `IVRDPServer`, `IRemoteDisplayInfo` and `IConsoleCallback` interfaces were changed to reflect VRDP server ability to bind to one of available ports from a list of ports.

  The `IVRDPServer::port` attribute has been replaced with `IVRDPServer::ports`, which is a comma-separated list of ports or ranges of ports.

  An `IRemoteDisplayInfo::port` attribute has been added for querying the actual port VRDP server listens on.

  An IConsoleCallback::onRemoteDisplayInfoChange() notification callback has been added.

- The parameter lists for the following functions were modified:

  - IHost::removeHostOnlyNetworkInterface()
  - IHost::removeUSBDeviceFilter()

- In the OOWS bindings for JAX-WS, the behavior of structures changed: for one, we implemented natural structures field access so you can just call a "get" method to obtain a field. Secondly, setters in structures were disabled as they have no expected effect and were at best misleading.

12.4 Incompatible API changes with version 3.0

- In the object-oriented web service bindings for JAX-WS, proper inheritance has been introduced for some classes, so explicit casting is no longer needed to call methods from a parent class. In particular, IHardDisk and other classes now properly derive from IMedium.

- All object identifiers (machines, snapshots, disks, etc) switched from GUIDs to strings (now still having string representation of GUIDs inside). As a result, no particular internal structure can be assumed for object identifiers; instead, they should be treated as opaque unique handles. This change mostly affects Java and C++ programs; for other languages, GUIDs are transparently converted to strings.

- The uses of NULL strings have been changed greatly. All out parameters now use empty strings to signal a null value. For in parameters both the old NULL and empty string is allowed. This change was necessary to support more client bindings, especially using the

webservice API. Many of them either have no special NULL value or have trouble dealing with it correctly in the respective library code.

- Accidentally, the `TSBool` interface still appeared in 3.0.0, and was removed in 3.0.2. This is an SDK bug, do not use the SDK for VirtualBox 3.0.0 for developing clients.
- The type of IVirtualBoxErrorInfo::resultCode changed from `result` to `long`.
- The parameter list of IVirtualBox::openHardDisk was changed.
- The method IConsole::discardSavedState was renamed to IConsole::forgetSavedState, and a parameter was added.
- The method IConsole::powerDownAsync was renamed to IConsole::powerDown, and the previous method with that name was deleted. So effectively a parameter was added.
- In the IFramebuffer interface, the following were removed:
  - the `operationSupported` attribute;

    (as a result, the `FramebufferAccelerationOperation` enum was no longer needed and removed as well);
  - the `solidFill()` method;
  - the `copyScreenBits()` method.
- In the IDisplay interface, the following were removed:
  - the `setupInternalFramebuffer()` method;
  - the `lockFramebuffer()` method;
  - the `unlockFramebuffer()` method;
  - the `registerExternalFramebuffer()` method.

12.5 Incompatible API changes with version 2.2

- Added explicit version number into JAX-WS Java package names, such as `org.virtualbox_2_2`, allowing connect to multiple VirtualBox clients from single Java application.
- The interfaces having a “2” suffix attached to them with version 2.1 were renamed again to have that suffix removed. This time around, this change involves only the name, there are no functional differences.

  As a result, IDVDImage2 is now IDVDImage; IHardDisk2 is now IHardDisk; IHardDisk2Attachment is now IHardDiskAttachment.

  Consequentially, all related methods and attributes that had a “2” suffix have been renamed; for example, IMachine::attachHardDisk2 now becomes IMachine::attachHardDisk().
- IVirtualBox::openHardDisk has an extra parameter for opening a disk read/write or read-only.
- The remaining collections were replaced by more performant safe-arrays. This affects the following collections:
  - IGuestOSTypeCollection
  - IHostDVDDriveCollection
  - IHostFloppyDriveCollection
  - IHostUSBDeviceCollection

  - IHostUSBDeviceFilterCollection
  - IProgressCollection
  - ISharedFolderCollection
  - ISnapshotCollection
  - IUSBDeviceCollection
  - IUSBDeviceFilterCollection

- Since "Host Interface Networking" was renamed to "bridged networking" and host-only networking was introduced, all associated interfaces needed renaming as well. In detail:
  - The HostNetworkInterfaceType enum has been renamed to HostNetworkInterfaceMediumType
  - The IHostNetworkInterface::type attribute has been renamed to IHostNetworkInterface::mediumType
  - INetworkAdapter::attachToHostInterface() has been renamed to INetworkAdapter::attachToBridgedInterfa
  - In the IHost interface, createHostNetworkInterface() has been renamed to createHostOnlyNetworkInterface()
  - Similarly, removeHostNetworkInterface() has been renamed to removeHostOnlyNetworkInterface()

12.6 Incompatible API changes with version 2.1

- With VirtualBox 2.1, error codes were added to many error infos that give the caller a machine-readable (numeric) feedback in addition to the error string that has always been available. This is an ongoing process, and future versions of this SDK reference will document the error codes for each method call.

- The hard disk and other media interfaces were completely redesigned. This was necessary to account for the support of VMDK, VHD and other image types; since backwards compatibility had to be broken anyway, we seized the moment to redesign the interfaces in a more logical way.
  - Previously, the old IHardDisk interface had several derivatives called IVirtualDiskImage, IVMDKImage, IVHDImage, IISCSIHardDisk and ICustomHardDisk for the various disk formats supported by VirtualBox. The new IHardDisk2 interface that comes with version 2.1 now supports all hard disk image formats itself.
  - IHardDiskFormat is a new interface to describe the available back-ends for hard disk images (e.g. VDI, VMDK, VHD or iSCSI). The IHardDisk2::format attribute can be used to find out the back-end that is in use for a particular hard disk image. ISystemProperties::hardDiskFormats[] contains a list of all back-ends supported by the system. ISystemProperties::defaultHardDiskFormat contains the default system format.
  - In addition, the new IMedium interface is a generic interface for hard disk, DVD and floppy images that contains the attributes and methods shared between them. It can be considered a parent class of the more specific interfaces for those images, which are now IHardDisk2, IDVDImage2 and IFloppyImage2.

    In each case, the "2" versions of these interfaces replace the earlier versions that did not have the "2" suffix. Previously, the IDVDImage and IFloppyImage interfaces were entirely unrelated to IHardDisk.
  - As a result, all parts of the API that previously referenced IHardDisk, IDVDImage or IFloppyImage or any of the old subclasses are gone and will have replacements that use IHardDisk2, IDVDImage2 and IFloppyImage2; see, for example, IMachine::attachHardDisk2.

  - In particular, the IVirtualBox::hardDisks2 array replaces the earlier IVirtualBox::hardDisks collection.

- IGuestOSType was extended to group operating systems into families and for 64-bit support.
- The IHostNetworkInterface interface was completely rewritten to account for the changes in how Host Interface Networking is now implemented in VirtualBox 2.1.
- The IVirtualBox::machines2[] array replaces the former IVirtualBox::machines collection.
- Added IHost::getProcessorFeature() and ProcessorFeature enumeration.
- The parameter list for IVirtualBox::createMachine() was modified.
- Added IMachine::pushGuestProperty.
- New attributes in IMachine: accelerate3DEnabled, HWVirtExVPIDEnabled, guestPropertyNotificationPatterns, CPUCount.
- Added IConsole::powerUpPaused() and IConsole::getGuestEnteredACPIMode().
- Removed ResourceUsage enumeration.