Sun xVM VirtualBox®

Programming Guide and Reference

Version 2.0.6

http://www.virtualbox.org

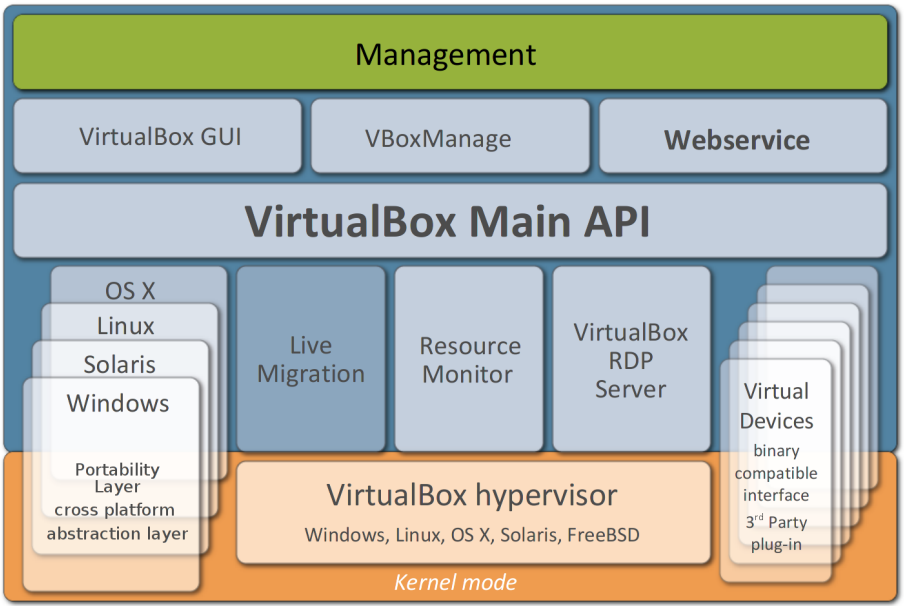
Contents

1 Introduction

VirtualBox comes with comprehensive support for third-party developers. The 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 visualized like in the picture below:



At the bottom of the stack (in orange) resides the hypervisor, which is the core of the virtualization engine that controls execution of the virtual machines and makes sure they do not conflict with whatever the host computer is doing otherwise.

On top of the hypervisor, additional internal modules provide extra functionality. For example, the RDP server that 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 here is the "VirtualBox 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 and available to anyone who wishes to control VirtualBox programmatically. We chose the name "Main API" to differentiate it from other parts of the program 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 so on. 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 for `VBoxHeadless`, which is not shown in the image.)

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

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

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

   Since the entire webservice API is publicy described in a web service description file (in WSDL format), you can write client programs that call the webservice 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 will be explained in detail in subsequent chapters of this book.

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

   a) For Java with JAX-WS as well as Python, the SDK contains easy-to-use classes that allow you to use the webservice in an object-oriented, straightforward manner. We shall refer to this as **"webservice client glue"**.

      The client glue for Java is described in chapter 2.1, *Using the client glue for JAX-WS*, page 18; the client glue for Python is described in chapter 2.2, *Using the client glue for Python*, page 19.

   b) Alternatively, you can use the webservice directly, without the client glue. We shall refer to this as the **"raw webservice"**.

      You will then have neither full object orientation nor type safety, since webservices 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 glue; 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 3, *Using the raw webservice with any language*, page 20.

2. Internally, for portability and easier maintenance, this Main API is expressed using the **Component Object Model (COM),** a 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, C# or JavaScript), 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 4, *The VirtualBox COM/XPCOM API*, page 29 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.

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

Webservice	**COM/XPCOM**
Pro: Easy to use with Java and Python with webservice client glue; extensive support even with other languages (C++, .NET, PHP, Perl and others)	**Con:** Requires compiled C++ code, high learning curve
Pro: Client can be on remote machine	**Con:** Client must be locally linked to VirtualBox code
Con: Significant overhead due to XML marshalling over the wire for each method call	**Pro:** Relatively high execution speed

In the following chapters, we will describe the different ways in which to program VirtualBox. We start with the method that is easiest to use

1.3 About webservices in general

Webservices 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, webservices use Internet standards such as HTTP and XML to communicate.[1]

[1] In some ways, webservices 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

In order to successfully use a webservice, a number of things are required – primarily, a webservice that is accepting connections, service descriptions, and then a client that connects to that webservice. 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 webservice (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 webservice and thus control a VirtualBox installation.

2. VirtualBox also comes with WSDL files that describe the services provided by the webservice. You can find these files in the `/sdk/webservice/` directory. These files are understood by the webservice toolkits that are shipped with most programming languages and enable you to easily access a webservice even if you don't use our client glue.

3. A client that connects to the webservice 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.

1.4 Running the webservice

The webservice 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 webservice 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.

to use between diverging platforms, webservices 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 webservice can be implemented. Many of the standards invented around XML are used one way or another. As a result, webservices 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 webservice. 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 webservice shields you from these details and is easy to use.

The vboxwebsrv program, which implements the webservice, 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 webservice 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 webservice 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 webservice supports the following command line options:

- -help (or -h): print a brief summary of command line options.
- -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 20. A webservice client that has logged on but makes no calls to the webservice will automatically be disconnected after the number of seconds specified here, as if it had called the IWebSessionManager::logoff() method provided by the webservice itself.

  It is normally vital that each webservice client call this method, as the webservice can accumulate large amounts of memory when running, especially if a webservice 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 webservice, or the webservice may eventually deny service.
- -check-interval (or -i): This specifies the interval in which the webservice checks for timed-out clients, in seconds, and defaults to 5. This normally does not need to be changed.

1.4.2 Authenticating at webservice logon

As opposed to the COM/XPCOM variant of the Main API, a client that wants to use the webservice must first log on by calling the IWebsessionManager::logon() API that is specific to the webservice. Logon is necessary for the webservice 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 webservice 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 authentication modules as laid out in section 9.3 of the VirtualBox User Manual; the webservice uses the same kind of modules as the VirtualBox RDP server.

By default, after installation, the webservice uses the VRDPAuth module that ships with VirtualBox. This module uses PAM on Linux hosts to authenticate users. Unless `vboxwebsrv` runs as root, authentication will fail because on most Linux distributions, the file `/etc/shadow`, which is used by PAM, is not readable.

1.4.3 Solaris host: starting webservice via SMF

On Solaris the VirtualBox webservice 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 in 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 refer to the Solaris documentation.

2 Starting out: the webservice client glue

To get you started quickly, we will run through one of the samples that are shipped with VirtualBox.

2.1 Using the client glue for JAX-WS

The VirtualBox SDK comes with precompiled support for JAX-WS bindings for both JDK1.5 and JDK1.6. The main difference is that JAX-WS in integrated into Java 6, thus no additional libraries are required when using Java 6.

As a common step before running any webservices examples, you need to start the VirtualBox webserver: `vboxwebsrv -t 1000`. The supplied parameter enables a watchdog for cleaning up unused object references after 1000 seconds. The default value of 5 seconds might cause your references to mysteriously disappear during debugging.

Change to the directory `bindings/webservice/java/jax-ws/samples` and examine the header of `Makefile` to see if the supplied variables (Java compiler, Java executable) and a few other details match your system settings.

2.1.1 Java 5 (JDK1.5.x)

As there's no out-of-the-box support of JAX-WS in Java 5, you need to download external JAX-WS implementation, for example from `https://jax-ws.dev.java.net/2.1.4/JAXWS2.1.4-20`
Then perform the installation (`java -jar JAXWS2.1.4-20080502.jar`).

Assuming you are in the `bindings/webservice/java/jax-ws/samples` directory, to start a simple client example just type `make run15` on a Linux or Solaris system. For Windows systems use commands similar to what is used in the Makefile. You can check the source code to see how typical tasks are performed and the Makefile to figure out compiler and runtime commandlines for using the VirtualBox API in your own project.

2.1.2 Java 6 (JDK1.6.x)

All you need to get the 1.6 examples working is a Java Development Kit (JDK) that ships with JAX-WS, the Java API for XML Web Services. JAX-WS is included with the Sun JDK Version 1.6 and above.

Type `make run16` on Linux or Solaris systems to compile an example program and start a simple demo enumerating VirtualBox VMs.

The current glue code has certain memory management related limitations, so when you no longer need an object - call its method `releaseRemote()` which frees appropriate managed reference. This limitatation may be reconsidered in a future version of the VirtualBox SDK.

2.2 Using the client glue for Python

VirtualBox comes with two flavors of the Python API: the webservices, discussed here, and the XPCOM API discussed in chapter 4.1, *Python XPCOM API*, page 29. The client code is mostly similar, except from the initialization part, so it's up to the application developer to choose the appropriate technology. The XPCOM API gives better performance without the SOAP overhead, enables certain features not possible via SOAP (e.g. callbacks) and does not require a webserver to be running. On the other hand, the XPCOM Python API requires a suitable Python XPCOM bridge for your Python installation (VirtualBox ships the most important ones for each platform) and it does not allow you to control remote instances of VirtualBox. Last but not least, Python is currently not supported on Windows hosts (you may use VBScript there).

The VirtualBox Python webservices glue code relies on the Python ZSI SOAP implementation (see `http://pywebsvcs.sourceforge.net/zsi.html`) thus it has to be installed in your Python distribution before trying the examples.

To get started change to the directory `bindings/webservice/python/samples` which contains an example of a simple interactive shell to control a VirtualBox instance. Just type `PYTHONPATH=../lib python ./vboxshell.py` or simply `make` to start the shell. See chapter 5, *The VirtualBox shell*, page 31 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 to write code targeting both local and remote cases (XPCOM and SOAP). The common part of the shell is the same – the only difference is how it interacts with the invocation layer.

3 Using the raw webservice with any language

The following examples show you how to use the raw webservice, without the client glue that was described in the preceding chapter.

3.1 Raw webservice example for Java and Ajax

Instead of Sun's JAX-WS, which ships with Java 1.6 and above, you can also use Ajax, an older webservice 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
   ```

   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 webservice offers, as described by the WSDL file.

This is the bit that makes using webservices 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 webservice, open a second terminal and change to the directory where the VirtualBox executables are located. Then type:

   ```
   ./vboxwebsrv
   ```

   The webservice now waits for connections and will run until you press Ctrl+C in this second terminal. (See chapter 1.4, *Running the webservice*, page 15 for details on how to run the webservice.)

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

   ```
   java clienttest
   ```

   The client sample will connect to the webservice (on localhost, but the code could be changed to connect remotely if the webservice 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).

3.2 Raw webservice example for Perl

We also ship a small sample for Perl. It uses the SOAP::Lite module to parse the VirtualBox WSDL file; you may need to install that module on your system first. Then perform the following steps:

1. Open a terminal and change to the `sdk/bindings/webservice/perl/samples/` directory.

2. In that directory, run Perl's `stubmaker.pl` tool on VirtualBox's `vboxwebService.wsdl` file (which you find in the parent directory, `sdk/bindings/webservice/`). Note that you need to specify an absolute path prefixed with "`file:///`", like so:

   ```
   stubmaker.pl file:///path/to/sdk/bindings/webservice/vboxweb.wsdl
   ```

   This can take a minute or, but needs to be done only once: the "stubmaker" tool parses the WSDL file and create a Perl module (`vboxService.pm`) that support the interfaces described in the WSDL file. (This module is then included by the Perl sample code via the "use vboxService" line).

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

   ```
   ./vboxwebsrv
   ```

   The webservice now waits for connections and will run until you press Ctrl+C in this second terminal. (See chapter 1.4, *Running the webservice*, page 15 for details on how to run the webservice.)

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

   ```
   perl clienttest.pl
   ```

and run the example in there while the webservice is running (start it as described in the previous chapters).

3.3 Programming considerations for the raw webservice

3.3.1 Fundamental conventions

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

- Webservices, as expressed by WSDL, are not object-oriented. Even worse, they are normally stateless (or, in webservices terminology, "loosely coupled"). Webservice operations are entirely procedural, and one cannot normally make assumptions of the state of a webservice between function calls. Most importantly, you cannot normally work on objects that are created by one method call.
- 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 webservice, this results in three fundamental conventions:

1. All **function names** in the VirtualBox webservice 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.
2. All calls to the VirtualBox webservice (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.)

   So, 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 webservice, this looks something like this (again, pseudo-code):

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

3. To make the webservice stateful, and objects persistent between method calls, the VirtualBox webservice introduces a **session manager** (by way of the `IWebsessionManager` interface), which manages object references. Any client wishing to interact with the webservice 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 webservices, **the VirtualBox webservice is both object-oriented and stateful.**

3.3.2 Example: A typical webservice client session

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

1. A client logs on to the webservice by calling `IWebsessionManager::logon()` with a valid user name and password. (The webservice can be configured to use various authentication methods, or to let anyone in, which obviously is not optimal for a production environment, as the webservice allows access to the entire VirtualBox API.)

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 16).

   For the new session, the webservice 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 webservice 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 `IWebsessionManager::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 webservice. For details, see below. As hinted above, `IWebsessionManager::logon()` is the *only* operation provided by the webservice 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 webservice. So, to retrieve the "version" attribute of this `IVirtualBox` object, the webservice client does this:

   ```
   string version;
   version = webservice->IVirtualBox_getVersion(oVirtualBox);

   print version;
   ```

   And it will print "2.0.6".

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

3.3.3 Managed object references

To a webservice 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 webservice 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 webservice 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 webservice 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 webservice 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 webserver, and this reference is returned to the webservice client.

Internally, in the webservice process, each managed object reference is simply a small data structure, containing a COM pointer to the "real" COM object, the session ID and the object ID. This structure is allocated on creation and stored efficiently in hashes, so that the webservice can look up the COM object quickly whenever a webservice client wishes to make a method call. The random session ID also ensures that one webservice 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 webservice 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 webservice clients properly clean up after the managed object references that are returned to them.

When a webservice 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".

3.3.4 Some more detail about webservice operation

3.3.4.1 SOAP messages

Whenever a client makes a call to a webservice, 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 webservice 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 webservice 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 webservice running on port 18083 of "localhost";
2. for the `SayHello()` function of the webservice, 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 webservice via HTTP and send that message;
4. wait for the webservice to send a response message;
5. decode that response message and put the return value of the remote procedure into the "result" variable.

3.3.4.2 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 webservice 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 webservice. For example, if a webservice 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 webservice supports and, for each operation, which parameters and types are needed with each request and reponse 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" webservices, 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 **??**, **??**, page **??**.

For details about how programming languages support webservices, 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 webservice.

2. For **Java,** there are several implementations – one of them being Axis by the Apache foundation, as described in chapter **??**, **??**, page **??**.

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 webservice client written in Perl is also shipped with VirtualBox.

3.3.5 Using the VirtualBox Main API documentation for webservice clients

The VirtualBox COM API is broken into many interfaces, which map to classes in C++. The complete API is documented in `VirtualBoxAPI.chm`, a help file in Windows Help format that can be viewed on both Windows and Linux (in the latter case, for example, with the `kchmviewer` program that ships with VirtualBox).

This file documents all interfaces, attributes and methods provided by the Main API. However, as indicated above, "interface", "attribute" and "method" are COM concepts, so the API documentation needs to be read with the following in mind:

1. Any method call becomes a **function call** in the webservice, with the object as the first parameter. So when the documentation says that the `IVirtualBox` interface supports the `createMachine()` method, the webservice 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 capizalized first letter. So, the "version" readonly attribute of the `IVirtualBox` interface can be retrieved by calling `IVirtualBox_getVersion()`.

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 webservice instead. As said above, managed object references should be released if the webservice client does not log off again immediately!

4. COM does not support **arrays**. As a result, the Main API needs to work around this limitation by using **"collections"** of objects wherever a list of things might be returned. For example, `IVirtualBox` has an attribute "machines", which is of the `IMachineCollection` type. However, arrays are supported in SOAP and

WSDL, so whenever you see a “Collection” of something in the API documentation, you can be sure that it is an array in your client programming language.

For example, the “machines” attribute of `IVirtualBox`, of which the API documentation says it is an `IMachineCollection`, will be an array of `IMachine` managed object references in your programming language.

Note: When an array of managed object references is returned by a function, you must release each managed object reference!

As was indicated above, the most central interface in the VirtualBox Main API is `IVirtualBox`. Virtual machines, which most programmers will also be interested in, are represented by the `IMachine` interface. Also, in order to do anything interesting with a virtual machine (such as changing its settings or starting it), one needs to create a session object first; this is of the `ISession` interface. It is recommended to read through the documentation of these three interfaces first to get a basic grip on the Main API.

4 The VirtualBox COM/XPCOM API

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

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.

4.1 Python XPCOM API

4.2 C++ COM API

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 only 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.

5 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. You can easily extend this shell with your own commands.

6 Classes (interfaces)

6.1 IAudioAdapter

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

6.1.1 Attributes

6.1.1.1 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.

6.1.1.2 audioController (read/write)

`AudioControllerType IAudioAdapter::audioController`

The audio hardware we emulate.

6.1.1.3 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.

6.2 IBIOSSettings

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

6.2.1 Attributes

6.2.1.1 logoFadeIn (read/write)

`boolean IBIOSSettings::logoFadeIn`

Fade in flag for BIOS logo animation.

6.2.1.2 logoFadeOut (read/write)

```
boolean IBIOSSettings::logoFadeOut
```

Fade out flag for BIOS logo animation.

6.2.1.3 logoDisplayTime (read/write)

```
unsigned long IBIOSSettings::logoDisplayTime
```

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

6.2.1.4 logoImagePath (read/write)

```
wstring IBIOSSettings::logoImagePath
```

Local file system path for external BIOS image.

6.2.1.5 bootMenuMode (read/write)

```
BIOSBootMenuMode IBIOSSettings::bootMenuMode
```

Mode of the BIOS boot device menu.

6.2.1.6 ACPIEnabled (read/write)

```
boolean IBIOSSettings::ACPIEnabled
```

ACPI support flag.

6.2.1.7 IOAPICEnabled (read/write)

```
boolean IBIOSSettings::IOAPICEnabled
```

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

6.2.1.8 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 other than it's not an absolute value but a relative one. Guest Additions time synchronization also honors this offset.

6.2.1.9 PXEDebugEnabled (read/write)

`boolean IBIOSSettings::PXEDebugEnabled`

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

6.2.1.10 IDEControllerType (read/write)

`IDEControllerType IBIOSSettings::IDEControllerType`

Type of the virtual IDE controller. Depending on this value, VirtualBox will provide different virtual IDE hardware devices to the guest.

6.3 IConsole

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

The console object that implements the IConsole interface is obtained from a session object after the session for the given machine has been opened using one of IVirtualBox::openSession, IVirtualBox::openRemoteSession or IVirtualBox::openExistingSession methods.

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

6.3.1 Attributes

6.3.1.1 machine (read-only)

`IMachine IConsole::machine`

Machine object this console is sessioned with.

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

6.3.1.2 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 this console is sessioned with. For the process that owns (executes) the VM, this is the preferable way of querying the VM state, because no IPC calls are made.

6.3.1.3 guest (read-only)

`IGuest IConsole::guest`

Note: This attribute is not supported in the webservice.

Guest object.

6.3.1.4 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.

6.3.1.5 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.

6.3.1.6 display (read-only)

`IDisplay IConsole::display`

Note: This attribute is not supported in the webservice.

Virtual display object.

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

6.3.1.7 debugger (read-only)

`IMachineDebugger IConsole::debugger`

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

Debugging interface.

6.3.1.8 USBDevices (read-only)

`IUSBDeviceCollection IConsole::USBDevices`

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

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

6.3.1.9 remoteUSBDevices (read-only)

`IHostUSBDeviceCollection IConsole::remoteUSBDevices`

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

6.3.1.10 sharedFolders (read-only)

`ISharedFolderCollection 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.

6.3.1.11 remoteDisplayInfo (read-only)

`IRemoteDisplayInfo IConsole::remoteDisplayInfo`

Interface that provides information on Remote Display (VRDP) connection.

6.3.2 adoptSavedState

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

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 full 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, hard disk configuration etc.). If there is a mismatch, the behavior of the virtual machine is undefined.

6.3.3 attachUSBDevice

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

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

6.3.4 createSharedFolder

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

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.

6.3.5 detachUSBDevice

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

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

After this method succeeds, the VirtualBox server reinitiates 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 reattachment.

See also: IUSBController::deviceFilters, USBDeviceState

6.3.6 discardCurrentSnapshotAndState

```
IProgress IConsole::discardCurrentSnapshotAndState()
```

This method is equivalent to doing discardSnapshot (currentSnapshot.id(), progress) followed by discardCurrentState().

As a result, the machine will be fully restored from the snapshot preceeding the current snapshot, while both the current snapshot and the current machine state will be discarded.

If the current snapshot is the first snapshot of the machine (i.e. it has the only snapshot), the current machine state will be discarded **before** discarding the snapshot. In other words, the machine will be restored from its last snapshot, before discarding it. This differs from performing a single discardSnapshot() call (note that no discardCurrentState() will be possible after it) to the effect that the latter will preserve the current state instead of discarding it.

Unless explicitly mentioned otherwise, all remarks and limitations of the above two methods also apply to this method.

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 discarded (as if discardSavedState() were called).

Note: This method is more efficient than calling two above methods separately: it requires less IPC calls and provides a single progress object.

6.3.7 discardCurrentState

```
IProgress IConsole::discardCurrentState()
```

This operation is similar to discardSnapshot() but affects the current machine state. This means that the state stored in the current snapshot will become a new current state, and all current settings of the machine and changes stored in differencing hard disks will be lost.

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

If the current snapshot of the machine is an online snapshot, the machine will go to the saved state, so that the next time it is powered on, the execution state will be restored from the current 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 discarded (as if IConsole::discardSavedState() were called).

6.3.8 discardSavedState

```
void IConsole::discardSavedState()
```

Discards (deletes) the saved state of the virtual machine 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 in the guest OS.

6.3.9 discardSnapshot

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

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

If the discarded 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 hard disks based on hard disks of the discarded snapshot will be also merged as described above.

If the discarded snapshot is the first one (the root snapshot) and it has exactly one child snapshot, this child snapshot will become the first snapshot after discarding. If there are no children at all (i.e. the first snapshot is the only snapshot of the machine), both the current and the first snapshot of the machine will be set to null. In all other cases, the first snapshot cannot be discarded.

You cannot discard the snapshot if it stores normal (non-differencing) hard disks that have differencing hard disks based on them. Snapshots of such kind can be discarded only when every normal hard disk has either no children at all or exactly one child. In the former case, the normal hard disk simply becomes unused (i.e. not attached to any VM). In the latter case, it receives all the changes strored in the child hard disk, and then it replaces the child hard disk in the configuration of the corresponding snapshot or machine.

Also, you cannot discard the snapshot if it stores hard disks (of any type) having differencing child hard disks that belong to other machines. Such snapshots can be only discarded after you discard all snapshots of other machines containing “foreign” child disks, or detach these “foreign” child disks from machines they are attached to.

One particular example of the snapshot storing normal hard disks is the first snapshot of a virtual machine that had normal hard disks attached when taking the snapshot. Be careful when discarding such snapshots because this implicitly commits changes (made since the snapshot being discarded has been taken) to normal hard disks (as described above), which may be not what you want.

The virtual machine is put to the Discarding state until the discard operation is completed.

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

Note: Child hard disks of all normal hard disks of the discarded snapshot must be accessible for this operation to succeed. In particular, this means that all virtual machines, whose hard disks are directly or indirectly based on the hard disks of discarded snapshot, must be powered off.

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

Note: Note that discarding the current snapshot will imlicitly call IMachine::saveSettings() to make all current machine settings permanent.

6.3.10 getDeviceActivity

```
DeviceActivity IConsole::getDeviceActivity(
            [in] DeviceTypetype)
```

Gets the current activity type of a given device or device group.

6.3.11 getPowerButtonHandled

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

Check if the last power button event was handled by guest.

6.3.12 pause

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

Pauses the virtual machine execution.

6.3.13 powerButton

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

Send the ACPI power button event to the guest.

6.3.14 powerDown

```
void IConsole::powerDown()
```

Stops the virtual machine execution. After this operation completes, the machine will go to the PoweredOff state.

@deprecated This method will be removed in VirtualBox 2.1 where the powerDownAsync() method will take its name. Do not use this method in the code.

6.3.15 powerDownAsync

```
IProgress IConsole::powerDownAsync()
```

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.

@warning This method will be renamed to “powerDown” in VirtualBox 2.1 where the original powerDown() method will be removed. You will need to rename “powerDownAsync” to “powerDown” in your sources to make them build with version 2.1.

6.3.16 powerUp

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

Starts the virtual machine execution using the current machine state (i.e. its current execution state, current settings and current hard disks).

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 MachineState::Saved state, it will continue its execution the point where the state has been saved.

> **Note:** Unless you are trying to write a new VirtualBox front-end that performs direct machine execution (like the VirtualBox or VBoxSDL front-ends), don't call IConsole::powerUp in a direct session opened by IVirtualBox::openSession and use this session only to change virtual machine settings. If you simply want to start virtual machine execution using one of the existing front-ends (for example the VirtualBox GUI or headless server), simply use IVirtualBox::openRemoteSession; these front-ends will power up the machine automatically for you.

See also: #saveState

6.3.17 registerCallback

```
void IConsole::registerCallback(
            [in] IConsoleCallback callback)
```

Registers a new console callback on this instance. The methods of the callback interface will be called by this instance when the appropriate event occurs.

6.3.18 removeSharedFolder

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

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

6.3.19 reset

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

Resets the virtual machine.

6.3.20 resume

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

Resumes the virtual machine execution.

6.3.21 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 hard disks. 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 drive, etc.). Together with the impossibility to change any VM settings when it is in the Saved state, this guarantees the 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

6.3.22 sleepButton

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

Send the ACPI sleep button event to the guest.

6.3.23 takeSnapshot

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

Saves the current execution state and all settings of the machine and creates differencing images for all normal (non-independent) hard disks.

This method can be called for a PoweredOff, Saved, Running or Paused virtual machine. When the machine is PoweredOff, an offline snapshot is created, in all other cases – an online snapshot.

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.

See also: ISnapshot, saveState

6.3.24 unregisterCallback

```
void IConsole::unregisterCallback(
            [in] IConsoleCallbackcallback)
```

Unregisters the console callback previously registered using registerCallback.

6.4 IConsoleCallback

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

6.4.1 onAdditionsStateChange

```
void IConsoleCallback::onAdditionsStateChange()
```

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

6.4.2 onCanShowWindow

```
boolean IConsoleCallback::onCanShowWindow()
```

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 in the @a canShow argument. This answer must remain valid at least until the next machine state change.

Note: This notification is not designed to be implemented by more than one callback at a time. If you have multiple IConsoleCallback instances registered on the given IConsole object, make sure you simply do nothing but return @c true and @c S_OK from all but one of them that actually manages console window activation.

6.4.3 onDVDDriveChange

```
void IConsoleCallback::onDVDDriveChange()
```

Notification when a property of the virtual DVD drive changes. Interested callees should use IDVDDrive methods to find out what has changed.

6.4.4 onFloppyDriveChange

```
void IConsoleCallback::onFloppyDriveChange()
```

Notification when a property of the virtual floppy drive changes. Interested callees should use IFloppyDrive methods to find out what has changed.

6.4.5 onKeyboardLedsChange

```
void IConsoleCallback::onKeyboardLedsChange(
            [in] boolean numLock,
            [in] boolean capsLock,
            [in] boolean scrollLock)
```

Notification when the guest OS executes the KBD_CMD_SET_LEDS command to alter the state of the keyboard LEDs.

6.4.6 onMouseCapabilityChange

```
void IConsoleCallback::onMouseCapabilityChange(
            [in] boolean supportsAbsolute,
            [in] boolean needsHostCursor)
```

Notification when the mouse capabilities reported by the guest have changed. The new capabilities are passed.

6.4.7 onMousePointerShapeChange

```
void IConsoleCallback::onMousePointerShapeChange(
            [in] boolean visible,
            [in] boolean alpha,
            [in] unsigned long xHot,
```

```
            [in] unsigned long yHot,
            [in] unsigned long width,
            [in] unsigned long height,
            [in] octet shape)
```

Notification when the guest mouse pointer shape has changed. The new shape data is given.

6.4.8 onNetworkAdapterChange

```
void IConsoleCallback::onNetworkAdapterChange(
            [in] INetworkAdapternetworkAdapter)
```

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.

6.4.9 onParallelPortChange

```
void IConsoleCallback::onParallelPortChange(
            [in] IParallelPortparallelPort)
```

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.

6.4.10 onRuntimeError

```
void IConsoleCallback::onRuntimeError(
            [in] boolean fatal,
            [in] wstring id,
            [in] wstring message)
```

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 @a 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 @a 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 probem 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 @a 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 identificators are known:

- "HostMemoryLow"
- "HostAudioNotResponding"
- "VDIStorageFull"

Note: This notification is not designed to be implemented by more than one callback at a time. If you have multiple IConsoleCallback instances registered on the given IConsole object, make sure you simply do nothing but return @c S_OK from all but one of them that does actual user notification and performs necessary actions.

6.4.11 onSerialPortChange

```
void IConsoleCallback::onSerialPortChange(
            [in] ISerialPortserialPort)
```

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.

6.4.12 onSharedFolderChange

```
void IConsoleCallback::onSharedFolderChange(
            [in] Scopescope)
```

Notification when a shared folder is added or removed. The @a 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.

6.4.13 onShowWindow

```
unsigned long long IConsoleCallback::onShowWindow()
```

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 @a 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.

> **Note:** This notification is not designed to be implemented by more than one callback at a time. If you have multiple IConsoleCallback instances registered on the given IConsole object, make sure you simply do nothing but return@c S_OK from all but one of them that actually manages console window activation.

6.4.14 onStateChange

```
void IConsoleCallback::onStateChange(
            [in] MachineStatestate)
```

Notification when the execution state of the machine has changed. The new state will be given.

6.4.15 onUSBControllerChange

```
void IConsoleCallback::onUSBControllerChange()
```

Notification when a property of the virtual USB controller changes. Interested callees should use IUSBController methods and attributes to find out what has changed.

6.4.16 onUSBDeviceStateChange

```
void IConsoleCallback::onUSBDeviceStateChange(
            [in] IUSBDevicedevice,
            [in] boolean attached,
            [in] IVirtualBoxErrorInfoerror)
```

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 @a error parameter is @c null, and the given device has been already added to (when @a attached is @c true) or removed from (when @a attached is @c false) the collection represented by IConsole::USBDevices. On failure, the collection doesn't change and the @a error perameter represents the error message describing the failure.

6.4.17 onVRDPServerChange

```
void IConsoleCallback::onVRDPServerChange()
```

Notification when a property of the VRDP server changes. Interested callees should use IVRDPServer methods and attributes to find out what has changed.

6.5 ICustomHardDisk

The ICustomHardDisk interface represents a specific type of IHardDisk that is supported through a third-party plugin.

This interface allows to add support for custom hard disk formats to VirtualBox.

Objects that support this interface also support the IHardDisk interface.

Hard disks using custom hard disk formats can be either opened using IVirtualBox::openHardDisk() or created from scratch using IVirtualBox::createHardDisk().

When a new hard disk object is created from scratch, an image file for it is not automatically created. To do it, you need to specify a valid location, and call createFixedImage() or createDynamicImage(). When it is done, the hard disk object can be registered by calling IVirtualBox::registerHardDisk() and then attached to virtual machines.

The description of the hard disk is stored in the VirtualBox configuration file, so it can be changed (at appropriate times) even when accessible returns `false`. However, the hard disk must not be attached to a running virtual machine.

6.5.1 Attributes

6.5.1.1 location (read/write)

```
wstring ICustomHardDisk::location
```

Location of this custom hard disk. For newly created hard disk objects, this value is `null`.

The format of the location string is plugin-dependent. In case if the plugin uses a regular file in the local file system to store hard disk data, then the location is a file path and the following rules apply:

- when assigning a new path, it must be absolute (full path) or relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to the image file.
- When reading this propery, a full path is always returned.

Note: This property cannot be changed when created returns `true`.

6.5.1.2 format (read-only)

```
wstring ICustomHardDisk::format
```

The plugin name of the image file.

6.5.1.3 created (read-only)

```
boolean ICustomHardDisk::created
```

Whether the virual disk image is created or not. For newly created hard disk objects or after a successful invocation of deleteImage(), this value is `false` until createFixed-Image() or createDynamicImage() is called.

6.5.2 createDynamicImage

```
IProgress ICustomHardDisk::createDynamicImage(
            [in] unsigned long long size)
```

Starts creating a dymically expanding hard disk image in the background. The previous image associated with this object, if any, must be deleted using deleteImage, otherwise the operation will fail.

> **Note:** After the returned progress object reports that the operation is complete, this hard disk object can be registered within this VirtualBox installation.

6.5.3 createFixedImage

```
IProgress ICustomHardDisk::createFixedImage(
            [in] unsigned long long size)
```

Starts creating a fixed-size hard disk image in the background. The previous image, if any, must be deleted using deleteImage, otherwise the operation will fail.

> **Note:** After the returned progress object reports that the operation is complete, this hard disk object can be registered within this VirtualBox installation.

6.5.4 deleteImage

```
void ICustomHardDisk::deleteImage()
```

Deletes the existing hard disk image. The hard disk must not be registered within this VirtualBox installation, otherwise the operation will fail.

> **Note:** After this operation succeeds, it will be impossible to register the hard disk until the image file is created again.

> **Note:** This operation is valid only for non-differencing hard disks, after they are unregistered using IVirtualBox::unregisterHardDisk().

6.6 IDVDDrive

The IDVDDrive interface represents the virtual CD/DVD drive of the virtual machine. Used in IMachine::DVDDrive.

6.6.1 Attributes

6.6.1.1 state (read-only)

```
DriveState IDVDDrive::state
```

Current drive state.

6.6.1.2 passthrough (read/write)

```
boolean IDVDDrive::passthrough
```

When a host drive is mounted and passthrough is enabled the guest will be able to directly send SCSI commands to the host drive. This enables the guest to use CD/DVD writers but is potentially dangerous.

6.6.2 captureHostDrive

```
void IDVDDrive::captureHostDrive(
            [in] IHostDVDDrive drive)
```

Captures the specified host drive.

6.6.3 getHostDrive

```
IHostDVDDrive IDVDDrive::getHostDrive()
```

Gets the currently mounted image ID.

6.6.4 getImage

```
IDVDImage IDVDDrive::getImage()
```

Gets the currently mounted image ID.

6.6.5 mountImage

```
void IDVDDrive::mountImage(
            [in] uuid imageId)
```

Mounts the specified image.

6.6.6 unmount

```
void IDVDDrive::unmount()
```

Unmounts the currently mounted image/device.

6.7 IDVDImage

The IDVDImage interface represents a file containing the image of the DVD or CD disk.

Image Accessibility

The accessible attribute of the image object defines the accessibility state of the image file. If the value of this attribute is `false` then some image attributes may contain invalid or outdated values (for example, the the image file size) until a new accessibility check is done that returns `true`.

> **Note:** Because of the possible slowness of the accessibility check, it is not implicitly performed upon the VirtualBox server startup (to prevent the application freeze). In partcular, this means that if you try to read image properties that depend on the accessibility state without first reading the value of the accessible attribute and ensuring it's value is `true`, you will get wrong (zero) values.

6.7.1 Attributes

6.7.1.1 id (read-only)

```
uuid IDVDImage::id
```

UUID of the CD/DVD image.

6.7.1.2 filePath (read-only)

```
wstring IDVDImage::filePath
```

Full file name of the CD/DVD image.

6.7.1.3 accessible (read-only)

```
boolean IDVDImage::accessible
```

Whether the CD/DVD image is currently accessible or not. The image, for example, can be unaccessible if it is placed on a network share that is not available by the time this property is read.

The accessibility check is performed automatically every time this attribute is read. You should keep it in mind that this check may be slow and can block the calling thread for a long time (for example, if the network share where the image is located is down).

The following attributes of the image object are considered to be invalid when this attribute is `false`:

- size

6.7.1.4 size (read-only)

```
unsigned long long IDVDImage::size
```

Size of the ISO image in bytes.

6.8 IDisplay

Note: This interface is not supported in the webservice.

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 sessoin's display on a remote computer.

6.8.1 Attributes

6.8.1.1 width (read-only)

```
unsigned long IDisplay::width
```

Current display width.

6.8.1.2 height (read-only)

```
unsigned long IDisplay::height
```

Current display height.

6.8.1.3 bitsPerPixel (read-only)

```
unsigned long IDisplay::bitsPerPixel
```

Current guest display color depth. Note that this may differ from IFramebuffer::bitsPerPixel.

6.8.2 drawToScreen

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

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

6.8.3 getFramebuffer

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

Queries the framebuffer for given screen.

6.8.4 invalidateAndUpdate

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

Does a full invalidation of the VM display and instructs the VM to update it.

6.8.5 lockFramebuffer

```
octet IDisplay::lockFramebuffer()
```

Requests access to the internal framebuffer.

6.8.6 registerExternalFramebuffer

```
void IDisplay::registerExternalFramebuffer(
            [in] IFramebufferframebuffer)
```

Registers an external framebuffer.

6.8.7 resizeCompleted

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

Signals that a framebuffer has completed the resize operation.

6.8.8 setFramebuffer

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

Sets the framebuffer for given screen.

6.8.9 setSeamlessMode

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

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

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

6.8.10 setVideoModeHint

```
void IDisplay::setVideoModeHint(
            [in] unsigned long width,
            [in] unsigned long height,
            [in] unsigned long bitsPerPixel,
            [in] unsigned long 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 @a width, @a height or @a 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 @a 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, @a display must be `0`.

6.8.11 setupInternalFramebuffer

```
void IDisplay::setupInternalFramebuffer(
            [in] unsigned long depth)
```

Prepares an internally managed framebuffer.

6.8.12 takeScreenShot

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

Takes a screen shot of the requested size and copies it to the 32-bpp buffer allocated by the caller.

6.8.13 unlockFramebuffer

```
void IDisplay::unlockFramebuffer()
```

Releases access to the internal framebuffer.

6.8.14 updateCompleted

```
void IDisplay::updateCompleted()
```

Signals that a framebuffer has completed the update operation.

6.9 IFloppyDrive

The IFloppyDrive interface represents the virtual floppy drive of the virtual machine. Used in IMachine::FloppyDrive.

6.9.1 Attributes

6.9.1.1 enabled (read/write)

```
boolean IFloppyDrive::enabled
```

Flag whether the floppy drive is enabled. If it is disabled, the floppy drive will not be reported to the guest.

6.9.1.2 state (read-only)

```
DriveState IFloppyDrive::state
```

Current drive state.

6.9.2 captureHostDrive

```
void IFloppyDrive::captureHostDrive(
           [in] IHostFloppyDrive drive)
```

Captures the specified host drive.

6.9.3 getHostDrive

```
IHostFloppyDrive IFloppyDrive::getHostDrive()
```

Gets the currently mounted image ID.

6.9.4 getImage

```
IFloppyImage IFloppyDrive::getImage()
```

Gets the currently mounted image ID.

6.9.5 mountImage

```
void IFloppyDrive::mountImage(
            [in] uuid imageId)
```

Mounts the specified image.

6.9.6 unmount

```
void IFloppyDrive::unmount()
```

Unmounts the currently mounted image/device.

6.10 IFloppyImage

The IFloppyImage interface represents a file containing the image of a floppy disk.

Image Accessibility

The accessible attribute of the image object defines the accessibility state of the image file. If the value of this attribute is `false` then some image attributes may contain invalid or outdated values (for example, the the image file size) until a new accessibility check is done that returns `true`.

> **Note:** Because of the possible slowness of the accessibility check, it is not implicitly performed upon the VirtualBox server startup (to prevent the application freeze). In partcular, this means that if you try to read image properties that depend on the accessibility state without first reading the value of the accessible attribute and ensuring it's value is `true`, you will get wrong (zero) values.

6.10.1 Attributes

6.10.1.1 id (read-only)

```
uuid IFloppyImage::id
```

UUID of the floppy image.

6.10.1.2 filePath (read-only)

`wstring IFloppyImage::filePath`

Full file name of the floppy image.

6.10.1.3 accessible (read-only)

`boolean IFloppyImage::accessible`

Whether the floppy image is currently accessible or not. The image, for example, can be unaccessible if it is placed on a network share that is not available by the time this property is read.

The accessibility check is performed automatically every time this attribute is read. You should keep it in mind that this check may be slow and can block the calling thread for a long time (for example, if the network share where the image is located is down).

The following attributes of the image object are considered to be invalid when this attribute is `false`:

- size

6.10.1.4 size (read-only)

`unsigned long IFloppyImage::size`

Size of the floppy image in bytes.

6.11 IFramebuffer

Note: This interface is not supported in the webservice.

6.11.1 Attributes

6.11.1.1 address (read-only)

`octet IFramebuffer::address`

Address of the start byte of the framebuffer.

6.11.1.2 width (read-only)

`unsigned long IFramebuffer::width`

Framebuffer width, in pixels.

6.11.1.3 height (read-only)

`unsigned long IFramebuffer::height`

Framebuffer height, in pixels.

6.11.1.4 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.

6.11.1.5 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.

6.11.1.6 pixelFormat (read-only)

`unsigned long IFramebuffer::pixelFormat`

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

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

6.11.1.7 usesGuestVRAM (read-only)

`boolean IFramebuffer::usesGuestVRAM`

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

6.11.1.8 heightReduction (read-only)

`unsigned long IFramebuffer::heightReduction`

Hint from the framebuffer 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.

6.11.1.9 overlay (read-only)

```
IFramebufferOverlay IFramebuffer::overlay
```

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

An alpha-blended overlay which is superposed over the framebuffer. 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.

6.11.2 copyScreenBits

```
boolean IFramebuffer::copyScreenBits(
            [in] unsigned long xDst,
            [in] unsigned long yDst,
            [in] unsigned long xSrc,
            [in] unsigned long ySrc,
            [in] unsigned long width,
            [in] unsigned long height)
```

Copies specified rectangle on the screen.

6.11.3 getVisibleRegion

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

Returns the visible region of this framebuffer.

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

If @a rectangles is not `NULL` but @a count is less than the required number of elements to store region data, the method will report a failure. If @a 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 @a countCopied.

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

6.11.4 lock

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

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

6.11.5 notifyUpdate

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

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

6.11.6 operationSupported

```
boolean IFramebuffer::operationSupported(
            [in] FramebufferAccelerationOperationoperation)
```

Returns whether the given acceleration operation is supported by the IFramebuffer implementation. If not, the display object will not attempt to call the corresponding IFramebuffer entry point. Even if an operation is indicated to supported, the IFramebuffer implementation always has the option to return non supported from the corresponding acceleration method in which case the operation will be performed by the display engine. This allows for reduced IFramebuffer implementation complexity where only common cases are handled.

6.11.7 requestResize

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

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 visualising 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 authomatically by the underlying code.

The @a pixelFormat parameter defines whether the direct mode is available or not. If @a pixelFormat is PixelFormat::Opaque then direct access to the guest VRAM buffer is not available – the @a VRAM, @a bitsPerPixel and @a 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, @a pixelFormat together with @a bitsPerPixel and @a bytesPerLine define the format of the video memory buffer pointed to by the @a VRAM parameter and the implementation is free to choose which mode to use. To indicate that this framebuffer 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 @a VRAM parameter of this method; otherwise it is assumed that the indirect strategy is chosen.

The @a width and @a 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 @a VRAM buffer contains enough space to represent the display mode of the given size. Note that this framebuffer's width and height attributes must return exactly the same values as passed to this method after the resize is completed (see below).

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

Note that if the direct mode is chosen, the bitsPerPixel, bytesPerLine and pixelFormat attributes of this framebuffer 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 PixelFormat::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 @c false in @a finished, then this lock is not released until IDisplay::resizeCompleted() is called.

6.11.8 setVisibleRegion

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

Suggests a new visible region to this framebuffer. 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.

6.11.9 solidFill

```
boolean IFramebuffer::solidFill(
            [in] unsigned long x,
            [in] unsigned long y,
            [in] unsigned long width,
            [in] unsigned long height,
            [in] unsigned long color)
```

Fills the specified rectangle on screen with a solid color.

6.11.10 unlock

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

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

6.11.11 videoModeSupported

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

Returns whether the framebuffer 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.

6.12 IFramebufferOverlay

Note: This interface is not supported in the webservice.

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.

6.12.1 Attributes

6.12.1.1 x (read-only)

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

X position of the overlay, relative to the framebuffer.

6.12.1.2 y (read-only)

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

Y position of the overlay, relative to the framebuffer.

6.12.1.3 visible (read/write)

```
boolean IFramebufferOverlay::visible
```

Whether the overlay is currently visible.

6.12.1.4 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.

6.12.2 move

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

Changes the overlay's position relative to the IFramebuffer.

6.13 IGuest

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

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.

6.13.1 Attributes

6.13.1.1 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.

6.13.1.2 additionsActive (read-only)

```
boolean IGuest::additionsActive
```

Flag whether the Guest Additions are installed and active in which case their version will be returned by the additionsVersion property.

6.13.1.3 additionsVersion (read-only)

```
wstring IGuest::additionsVersion
```

Version of the Guest Additions (3 decimal numbers separated by dots) or empty when the Additions are not installed. The Additions may also report a version but yet not be active as the version might be refused by VirtualBox (incompatible) or other failures occured.

6.13.1.4 supportsSeamless (read-only)

```
boolean IGuest::supportsSeamless
```

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

6.13.1.5 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.

6.13.1.6 memoryBalloonSize (read/write)

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

Guest system memory balloon size in megabytes.

6.13.1.7 statisticsUpdateInterval (read/write)

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

Interval to update guest statistics in seconds.

6.13.2 getStatistic

```
void IGuest::getStatistic(
            [in] unsigned long cpuId,
            [in] GuestStatisticTypestatistic,
            [out] unsigned long statVal)
```

Query specified guest statistics as reported by the VirtualBox Additions.

6.13.3 setCredentials

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

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.

6.14 IGuestOSType

Note: With the webservice, 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.

6.14.1 Attributes

6.14.1.1 id (read-only)

```
wstring IGuestOSType::id
```

Guest OS identifier string.

6.14.1.2 description (read-only)

```
wstring IGuestOSType::description
```

Human readable description of the guest OS.

6.14.1.3 recommendedRAM (read-only)

```
unsigned long IGuestOSType::recommendedRAM
```

Recommended RAM size in Megabytes.

6.14.1.4 recommendedVRAM (read-only)

```
unsigned long IGuestOSType::recommendedVRAM
```

Recommended video RAM size in Megabytes.

6.14.1.5 recommendedHDD (read-only)

```
unsigned long IGuestOSType::recommendedHDD
```

Recommended hard disk size in Megabytes.

6.15 IHardDisk

The IHardDisk interface represents a virtual hard disk drive used by virtual machines.

The virtual hard disk drive virtualizes the hard disk hardware and looks like a regular hard disk inside the virtual machine and the guest OS.

Storage Types

The storage type of the virtual hard disk determines where and how it stores its data (sectors). Currently, the following storage types are supported:

- *Virtual Disk Image (VDI)*, a regular file in the file system of the host OS (represented by the IVirtualDiskImage interface). This file has a special format optimized so that unused sectors of data occupy much less space than on a physical hard disk.
- *iSCSI Remote Disk*, a disk accessed via the Internet SCSI protocol over a TCP/IP network link (represented by the IISCSIHardDisk interface).
- *VMware VMDK image*, a regular file in the file system of the host OS (represented by the IVMDKImage interface). Note that the regular file may be just a descriptor referring to further files, so don't make assumptions about the OS representation of a VMDK image.
- *Custom HardDisk*, a disk accessed via a plugin which is loaded when the disk is accessed (represented by the ICustomHardDisk interface).
- *Virtual PC VHD Image*, a regular file in the file system of the host OS (represented by the IVHDImage interface).

The storage type of the particular hard disk object is indicated by the storageType property.

Each storage type is represented by its own interface (as shown above), that allows to query and set properties and perform operations specific to that storage type. When an IHardDisk object reports it uses some particular storage type, it also guaranteed to support the corresponding interface which you can query. And vice versa, every object that supports a storage-specific interface, also supports IHardDisk.

Virtual Hard Disk Types

The type of the virtual hard disk determines how it is attached to the virtual machine when you call IMachine::attachHardDisk() and what happens to it when a snapshot of the virtual machine is taken.

There are three types of virtual hard disks:

- *Normal*
- *Immutable*
- *Writethrough*

The virtual disk type is indicated by the type property. Each of the above types is described in detail further down.

There is also a forth, "hidden" virtual disk type: *Differencing*. It is "hidden" because you cannot directly create hard disks of this type – they are automatically created by VirtualBox when necessary.

Differencing Hard Disks

Unlike disks of other types (that are similar to real hard disks), the differencing hard disk does not store the full range of data sectors. Instead, it stores only a subset of sectors of some other disk that were changed since the differencing hard disk has been created. Thus, every differencing hard disk has a parent hard disk it is linked to, and represents the difference between the initial and the current hard disk state. A differencing hard disk can be linked to another differencing hard disk – this way, differencing hard disks can form a branch of changes. More over, a given virtual hard disk can have more than one differencing hard disk linked to it.

A disk the differencing hard disk is linked to (or, in other words, based on) is called a *parent* hard disk and is accessible through the parent property. Similarly, all existing differencing hard disks for a given parent hard disk are called its *child* hard disks (and accessible through the children property).

All differencing hard disks use Virtual Disk Image files to store changed sectors. They have the type property set to Normal, but can be easily distinguished from normal hard disks using the parent property: all differencing hard disks have a parent, while all normal hard disks don't.

When the virtual machine makes an attempt to read a sector that is missing in a differencing hard disk, its parent is accessed to resolve the sector in question. This process continues until the sector is found, or until the root hard disk is encountered, which always contains all sectors. As a consequence,

- The virtual hard disk geometry seen by the guest OS is always defined by the root hard disk.
- All hard disks on a branch, up to the root, must be accessible for a given differencing hard disk in order to let it function properly when the virtual machine is running.

Differencing hard disks can be implicitly created by VirtualBox in the following cases:

- When a hard disk is *indirectly* attached to the virtual machine using IMachine::attachHardDisk(). In this case, all disk writes performed by the guest OS will go to the created diffferencing hard disk, as opposed to the *direct* attachment, where all changes are written to the attached hard disk itself.

- When a snapshot of the virtual machine is taken. After that, disk writes to hard disks the differencing ones have been created for, will be directed to those differencing hard disks, to preserve the contents of the original disks.

Whether to create a differencing hard disk or not depends on the type of the hard disk attached to the virtual machine. This is explained below.

Note that in the current implementation, only the VirtualDiskImage storage type is used to represent differencing hard disks. In other words, all differencing hard disks are IVirtualDiskImage objects.

Normal Hard Disks

Normal hard disks are the most commonly used virtual hard disk. A normal hard disk is attached to the machine directly if it is not already attached to some other machine. Otherwise, an attempt to make an indirect attachment through a differencing hard disk will be made. This attempt will fail if the hard disk is attached to a virtual machine without snapshots (because it's impossible to create a differencing hard disk based on a hard disk that is subject to change).

When an indirect attachment takes place, in the simplest case, where the machine the hard disk is being attached to doesn't have snapshots, the differencing hard disk will be based on the normal hard disk being attached. Otherwise, the first (i.e. the most recent) descendant of the given normal hard disk found in the current snapshot branch (starting from the current snapshot and going up to the root) will be actually used as a base.

Note that when you detach an indirectly attached hard disk from the machine, the created differencing hard disk image is simply **deleted**, so **all changes are lost**. If you attach the same disk again, a clean differencing disk will be created based on the most recent child, as described above.

When taking a snapshot, the contents of all normal hard disks (and all differencing disks whose roots are normal hard disks) currently attached to the virtual machine is preserved by creating differencing hard disks based on them.

Immutable Hard Disks

Immutable hard disks can be used to provide a sort of read-only access. An immutable hard disk is always attached indirectly. The created differencing hard disk is automatically wiped out (recreated from scratch) every time you power off the virtual machine. Thus, the contents of the immutable disk remains unchanged between runs.

Detaching an immutable hard disk deletes the differencing disk created for it, with the same effect as in case with normal hard disks.

When taking a snapshot, the differencing part of the immutable hard disk is cloned (i.e. copied to a separate Virtual Disk Image file) without any changes. This is necessary to preserve the exact virtual machine state when you create an online snapshot.

Writethrough Hard Disks

Hard disks of this type are always attached directly. This means that every given writethrough hard disk can be attached only to one virtual machine at a time.

It is impossible to take a snapshot of a virtual machine with the writethrough hard disk attached, because taking a snapshot implies saving the execution state and preserving the contents of all hard disks, but writethrough hard disks cannot be preserved.

Preserving hard disk contents is necessary to ensure the guest OS stored in the snapshot will get the same hard disk state when restored, which is especially important when it has open file handles or when there are cached files and directories stored in memory.

Creating, Opening and Registering Hard Disks

Non-differencing hard disks are either created from scratch using IVirtualBox::createHardDisk() or opened from a VDI file using IVirtualBox::openVirtualDiskImage() (only for hard disks using the VirtualDiskImage storage type). Once a hard disk is created or opened, it needs to be registered using IVirtualBox::registerHardDisk() to make it available for attaching to virtual machines. See the documentation of individual interfaces for various storage types to get more information.

Differencing hard disks are never created explicitly and cannot be registered or unregistered; they are automatically registered upon creation and deregistered when deleted.

More about Indirect Hard Disk Attachments

Normally, when you attach a hard disk to the virtual machine, and then query the corresponding attachment using IMachine::getHardDisk() or IMachine::hardDiskAttachments you will get the same hard disk object, whose UUID you passed earlier to IMachine::attachHardDisk(). However, when an indirect attachment takes place, calling IMachine::getHardDisk() will return a differencing hard disk object, that is either based on the attached hard disk or on another differencing hard disk, the attached hard disk is eventually a root for (as described above). In both cases the returned hard disk object is the object the virtual machine actually uses to perform disk writes to.

Regardless of whether the attachment is direct or indirect, the machineId property of the attached disk will contain an UUID of the machine object IMachine::attachHardDisk() has been called on.

Note that both IMachine::attachHardDisk() and IMachine::detachHardDisk() are *lazy* operations. In particular, this means that when an indirect attachment is made, differencing hard disks are not created until machine settings are committed using IMachine::saveSettings(). Similarly, when a differencing hard disk is detached, it is not deleted until IMachine::saveSettings() is called. Calling IMachine::discardSettings() cancels all lazy attachments or detachments made since the last commit and effectively restores the previous set of hard disks.

Hard Disk Accessibility

The accessible attribute of the hard disk object defines the accessibility state of the respective hard disk storage (for example, the VDI file for IVirtualDiskImage objects). If the value of this attribute is `false` then some hard disk attributes may contain invalid or outdated values (for example, the virtual or the actual hard disk size) until a new accessibility check is done that returns `true` (see the attribute description for more details).

Note: Since checking the accessibility of a hard disk is a potentially very slow operation, it is not performed implicitly when the VirtualBox server process starts up to prevent the application from freezing. In particular, this means that if you try to read hard disk properties that depend on the accessibility state without first reading the value of the accessible attribute and ensuring its value is `true`, you will get wrong (zero) values.

6.15.1 Attributes

6.15.1.1 id (read-only)

```
uuid IHardDisk::id
```

UUID of the hard disk. For newly created hard disk objects, this value is a randomly generated UUID.

6.15.1.2 description (read/write)

```
wstring IHardDisk::description
```

Optional description of the hard disk. For a newly created hard disk, this value is `null`.

Note: For some storage types, reading this property is meaningless when accessible is `false`. Also, you cannot assign it a new value in this case.

6.15.1.3 storageType (read-only)

```
HardDiskStorageType IHardDisk::storageType
```

Storage type of this hard disk.
Storage type is defined when you open or create a new hard disk object.

6.15.1.4 location (read-only)

```
wstring IHardDisk::location
```

Storage location of this hard disk. The returned string serves for informational purposes only. To access detailed information about the storage, query the appropriate storage-specific interface.

6.15.1.5 type (read/write)

`HardDiskType IHardDisk::type`

Type (behavior) of this hard disk. For a newly created or opened hard disk, this value is HardDiskType::Normal.

> **Note:** In the current implementation, this property can be changed only on an unregistered hard disk object. This may be changed later.

6.15.1.6 parent (read-only)

`IHardDisk IHardDisk::parent`

Parent of this hard disk (a hard disk this one is directly based on).

Only differencing hard disks have parents, so a `null` object is returned for a hard disk of any other type.

6.15.1.7 children (read-only)

`IHardDiskCollection IHardDisk::children`

Children of this hard disk (all differencing hard disks for those this one is a parent). An empty collection is returned, if this hard disk doesn't have any children.

6.15.1.8 root (read-only)

`IHardDisk IHardDisk::root`

Root hard disk of this hard disk. If this hard disk is a differencing hard disk, its root hard disk is the first disk on the branch. For all other types of hard disks, this property returns the hard disk object itself (i.e. the same object you read this property on).

6.15.1.9 accessible (read-only)

`boolean IHardDisk::accessible`

Whether the hard disk storage is currently accessible or not. The storage, for example, can be unaccessible if it doesn't exist or if it is placed on a network resource that is not available by the time this attribute is read.

In the current implementation, the value of this property is also `false` if this hard disk is attached to a running virtual machine.

The accessibility check is performed automatically every time this attribute is read. You should keep it in mind that this check may be slow and can block the calling thread

for a long time (for example, if the network resourse where the hard disk storage is located is down).

The following attributes of the hard disk object are considered to be invalid when this attribute is `false`:

- size
- actualSize

Individual hard disk storage type interfaces may define additional attributes that depend on the accessibility state.

6.15.1.10 allAccessible (read-only)

```
boolean IHardDisk::allAccessible
```

Whether the whole hard disk branch, starting from this image and going through its ancestors up to the root, is accessible or not.

This property makes sense only for differencing hard disks. For all other types of hard disks it returns the same value as accessible.

6.15.1.11 lastAccessError (read-only)

```
wstring IHardDisk::lastAccessError
```

String describing the reason of inaccessibility of this hard disk after the last call to accessible that returned `false`. A `null` value of this property means that the last accessibility check returned `true`.

6.15.1.12 size (read-only)

```
unsigned long long IHardDisk::size
```

Logical size of this hard disk (in megabytes), as reported to the guest OS running inside the vurtual machine this disk is attached to. The logical size is defined when the hard disk is created.

> **Note:** Reading this property on a differencing hard disk will return the size of its root hard disk.

> **Note:** Reading this property is meaningless when accessible is `false`

6.15.1.13 actualSize (read-only)

`unsigned long long IHardDisk::actualSize`

Physical size of the storage used to store hard disk data (in bytes). This size is usually less than the logical size of the hard disk, depending on the storage type and on the size optimization method used for that storage.

> **Note:** Reading this property is meaningless when accessible is `false`

6.15.1.14 machineId (read-only)

`uuid IHardDisk::machineId`

UUID of the machine this hard disk is attached to (or a `null` UUID if it is not attached).

> **Note:** Immutable hard disks are never attached directly, so this attribute is always `null` in this case.

6.15.1.15 snapshotId (read-only)

`uuid IHardDisk::snapshotId`

UUID of the snapshot this hard disk is associated with (or `null` UUID if it is not associated with any snapshot).

> **Note:** This attribute is always `null` if machineId is `null`.

> **Note:** Writethrough hard disks are always attached directly and cannot be involved when taking snapshots, so this attribute is meaningless and therefore always `null`.

6.15.2 cloneToImage

```
IProgress IHardDisk::cloneToImage(
            [in] wstring filePath,
            [out] IVirtualDiskImageimage)
```

Starts creating a clone of this hard disk. The cloned hard disk will use the specified Virtual Disk Image file as a storage and will contain exactly the same sector data as the hard disk being cloned, except that a new UUID for the clone will be randomly generated.

The specified image file path can be absolute (full path) or relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to the image file.

It is an error to use the object returned in the @a image parameter until the returned @a progress object reports success.

> **Note:** In the current implementation, only non-differencing hard disks can be cloned.

6.16 IHardDiskAttachment

> **Note:** With the webservice, 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.

6.16.1 Attributes

6.16.1.1 hardDisk (read-only)

```
IHardDisk IHardDiskAttachment::hardDisk
```

Harddisk object this attachment is about.

6.16.1.2 bus (read-only)

```
StorageBus IHardDiskAttachment::bus
```

Disk controller ID of this attachment.

6.16.1.3 channel (read-only)

```
long IHardDiskAttachment::channel
```

Channel number of the attachment.

6.16.1.4 device (read-only)

```
long IHardDiskAttachment::device
```

Device slot number of the attachment.

6.17 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.

6.17.1 Attributes

6.17.1.1 DVDDrives (read-only)

```
IHostDVDDriveCollection IHost::DVDDrives
```

List of DVD drives available on the host.

6.17.1.2 floppyDrives (read-only)

```
IHostFloppyDriveCollection IHost::floppyDrives
```

List of floppy drives available on the host.

6.17.1.3 USBDevices (read-only)

```
IHostUSBDeviceCollection 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:** This method may set a @ref com_warnings "warning result code".

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

6.17.1.4 USBDeviceFilters (read-only)

```
IHostUSBDeviceFilterCollection 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: This method may set a @ref com_warnings “warning result code”.

Note: If USB functionality is not avaliable in the given edition of VirtualBox, this method will set the result code to @c E_NOTIMPL.

See also: IHostUSBDeviceFilter, USBDeviceState

6.17.1.5 networkInterfaces (read-only)

```
IHostNetworkInterfaceCollection IHost::networkInterfaces
```

List of host network interfaces currently defined on the host.

6.17.1.6 processorCount (read-only)

```
unsigned long IHost::processorCount
```

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

6.17.1.7 processorOnlineCount (read-only)

```
unsigned long IHost::processorOnlineCount
```

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

6.17.1.8 memorySize (read-only)

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

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

6.17.1.9 memoryAvailable (read-only)

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

Available system memory in the host system.

6.17.1.10 operatingSystem (read-only)

```
wstring IHost::operatingSystem
```

Name of the host system's operating system.

6.17.1.11 OSVersion (read-only)

```
wstring IHost::OSVersion
```

Host operating system's version string.

6.17.1.12 UTCTime (read-only)

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

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

6.17.2 createUSBDeviceFilter

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

Creates a new USB device filter. All attributes except the filter name are set to `null` (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

6.17.3 getProcessorDescription

```
wstring IHost::getProcessorDescription(
            [in] unsigned long cpuId)
```

Query the model string of a specified host CPU.

6.17.4 getProcessorSpeed

```
unsigned long IHost::getProcessorSpeed(
            [in] unsigned long cpuId)
```

Query the (approximate) maximum speed of a specified host CPU in Megahertz.

6.17.5 insertUSBDeviceFilter

```
void IHost::insertUSBDeviceFilter(
            [in] unsigned long position,
            [in] IHostUSBDeviceFilterfilter)
```

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 list, will return an error.

> **Note:** This method may set a @ref com_warnings “warning result code”.

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

See also: #USBDeviceFilters

6.17.6 removeUSBDeviceFilter

```
IHostUSBDeviceFilter IHost::removeUSBDeviceFilter(
            [in] unsigned long position)
```

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:** This method may set a @ref com_warnings “warning result code”.

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

See also: #USBDeviceFilters

6.18 IHostDVDDrive

The IHostDVDDrive interface represents the physical CD/DVD drive hardware on the host. Used indirectly in IHost::DVDDrives.

6.18.1 Attributes

6.18.1.1 name (read-only)

```
wstring IHostDVDDrive::name
```

Returns the platform-specific device identifier. On DOS-like platforms, it is a drive name (e.g. R:). On Unix-like platforms, it is a device name (e.g. /dev/hdc).

6.18.1.2 description (read-only)

```
wstring IHostDVDDrive::description
```

Returns a human readable description for the drive. This description usually contains the product and vendor name. A @c null string is returned if the description is not available.

6.18.1.3 udi (read-only)

```
wstring IHostDVDDrive::udi
```

Returns the unique device identifier for the drive. This attribute is reserved for future use instead of name. Currently it is not used and may return @c null on some platforms.

6.19 IHostFloppyDrive

The IHostFloppyDrive interface represents the physical floppy drive hardware on the host. Used indirectly in IHost::floppyDrives.

6.19.1 Attributes

6.19.1.1 name (read-only)

```
wstring IHostFloppyDrive::name
```

Returns the platform-specific device identifier. On DOS-like platforms, it is a drive name (e.g. A:). On Unix-like platforms, it is a device name (e.g. /dev/fd0).

6.19.1.2 description (read-only)

```
wstring IHostFloppyDrive::description
```

Returns a human readable description for the drive. This description usually contains the product and vendor name. A @c null string is returned if the description is not available.

6.19.1.3 udi (read-only)

```
wstring IHostFloppyDrive::udi
```

Returns the unique device identifier for the drive. This attribute is reserved for future use instead of name. Currently it is not used and may return @c null on some platforms.

6.20 IHostNetworkInterface

6.20.1 Attributes

6.20.1.1 name (read-only)

```
wstring IHostNetworkInterface::name
```

Returns the host network interface name.

6.20.1.2 id (read-only)

```
uuid IHostNetworkInterface::id
```

Returns the interface UUID.

6.21 IHostUSBDevice

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 courrent state of the USB device.

See also: IHost::USBDevices, IHost::USBDeviceFilters

6.21.1 Attributes

6.21.1.1 state (read-only)

```
USBDeviceState IHostUSBDevice::state
```

Current state of the device.

6.22 IHostUSBDeviceFilter

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 remote attribute is ignored by this type of filters, because it makes sense only for machine USB filters.

See also: IHost::USBDeviceFilters

6.22.1 Attributes

6.22.1.1 action (read/write)

`USBDeviceFilterAction IHostUSBDeviceFilter::action`

Action performed by the host when an attached USB device matches this filter.

6.23 IISCSIHardDisk

THe IISCSIHardDisk interface represents a specific type of IHardDisk that uses iSCSI.

The IISCSIHardDisk interface represents virtual hard disks that use the Internet SCSI (iSCSI) protocol to store hard disk data on remote machines.

Objects that support this interface also support the IHardDisk interface.

iSCSI hard disks can be created using IVirtualBox::createHardDisk(). When a new hard disk object is created, all its properties are uninitialized. After you assign some meaningful values to them, the hard disk object can be registered by calling IVirtualBox::registerHardDisk() and then attached to virtual machines.

The description of the iSCSI hard disk is stored in the VirtualBox configuration file, so it can be changed (at appropriate times) even when accessible returns `false`. However, the hard disk must not be attached to a running virtual machine.

> **Note:** In the current imlementation, the type of all iSCSI hard disks is Writethrough and cannot be changed.

6.23.1 Attributes

6.23.1.1 server (read/write)

`wstring IISCSIHardDisk::server`

iSCSI Server name (either a host name or an IP address). For newly created hard disk objects, this value is `null`.

6.23.1.2 port (read/write)

```
unsigned short IISCSIHardDisk::port
```

iSCSI Server port. For newly created hard disk objects, this value is `0`, which means the default port.

6.23.1.3 target (read/write)

```
wstring IISCSIHardDisk::target
```

iSCSI target name. For newly created hard disk objects, this value is `null`.

6.23.1.4 lun (read/write)

```
unsigned long long IISCSIHardDisk::lun
```

Logical unit number for this iSCSI disk. For newly created hard disk objects, this value is `0`.

6.23.1.5 userName (read/write)

```
wstring IISCSIHardDisk::userName
```

User name for accessing this iSCSI disk. For newly created hard disk objects, this value is `null`.

6.23.1.6 password (read/write)

```
wstring IISCSIHardDisk::password
```

User password for accessing this iSCSI disk. For newly created hard disk objects, this value is `null`.

6.24 IInternalMachineControl

Note: This interface is not supported in the webservice.

6.24.1 adoptSavedState

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

Gets called by IConsole::adoptSavedState.

6.24.2 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.

6.24.3 beginSavingState

```
void IInternalMachineControl::beginSavingState(
            [in] IProgressprogress,
            [out] wstring stateFilePath)
```

Called by the VM process to inform the server it wants to save the current state and stop the VM execution.

6.24.4 beginTakingSnapshot

```
void IInternalMachineControl::beginTakingSnapshot(
            [in] IConsoleinitiator,
            [in] wstring name,
            [in] wstring description,
            [in] IProgressprogress,
            [out] wstring stateFilePath,
            [out] IProgressserverProgress)
```

Called by the VM process to inform the server it wants to take a snapshot.

6.24.5 captureUSBDevice

```
void IInternalMachineControl::captureUSBDevice(
            [in] uuid id)
```

Requests a capture of the given host USB device. When the request is completed, the VM process will get a IInternalSessionControl::onUSBDeviceAttach notification.

6.24.6 detachAllUSBDevices

```
void IInternalMachineControl::detachAllUSBDevices(
            [in] boolean 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.

6.24.7 detachUSBDevice

```
void IInternalMachineControl::detachUSBDevice(
            [in] uuid id,
            [in] boolean 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.

6.24.8 discardCurrentSnapshotAndState

```
IProgress IInternalMachineControl::discardCurrentSnapshotAndState(
            [in] IConsoleinitiator,
            [out] MachineStatemachineState)
```

Gets called by IConsole::discardCurrentSnapshotAndState.

6.24.9 discardCurrentState

```
IProgress IInternalMachineControl::discardCurrentState(
            [in] IConsoleinitiator,
            [out] MachineStatemachineState)
```

Gets called by IConsole::discardCurrentState.

6.24.10 discardSnapshot

```
IProgress IInternalMachineControl::discardSnapshot(
            [in] IConsoleinitiator,
            [in] uuid id,
            [out] MachineStatemachineState)
```

Gets called by IConsole::discardSnapshot.

6.24.11 endSavingState

```
void IInternalMachineControl::endSavingState(
            [in] boolean success)
```

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.

6.24.12 endTakingSnapshot

```
void IInternalMachineControl::endTakingSnapshot(
            [in] boolean success)
```

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.

6.24.13 getIPCId

```
wstring IInternalMachineControl::getIPCId()
```

6.24.14 onSessionEnd

```
IProgress IInternalMachineControl::onSessionEnd(
            [in] ISessionsession)
```

Triggered by the given session object when the session is about to close normally.

6.24.15 pullGuestProperties

```
void IInternalMachineControl::pullGuestProperties(
            [out] wstring name[],
            [out] wstring value[],
            [out] unsigned long long timestamp[],
            [out] wstring flags[])
```

Get the list of the guest properties matching a set of patterns along with their values, timestamps and flags and give responsibility for managing properties to the console.

6.24.16 pushGuestProperties

```
void IInternalMachineControl::pushGuestProperties(
            [in] wstring name[],
            [in] wstring value[],
            [in] unsigned long long timestamp[],
            [in] wstring flags[])
```

Set the list of the guest properties matching a set of patterns along with their values, timestamps and flags and return responsibility for managing properties to IMachine.

6.24.17 runUSBDeviceFilters

```
void IInternalMachineControl::runUSBDeviceFilters(
            [in] IUSBDevicedevice,
            [out] boolean matched,
            [out] unsigned long 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).

6.24.18 updateState

```
void IInternalMachineControl::updateState(
            [in] MachineStatestate)
```

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.

6.25 IInternalSessionControl

Note: This interface is not supported in the webservice.

6.25.1 accessGuestProperty

```
void IInternalSessionControl::accessGuestProperty(
            [in] wstring name,
            [in] wstring value,
            [in] wstring flags,
            [in] boolean isSetter,
            [out] wstring retValue,
            [out] unsigned long long retTimestamp,
            [out] wstring retFlags)
```

Called by IMachine::getGuestProperty() and by IMachine::setGuestProperty() in order to read and modify guest properties.

6.25.2 assignMachine

```
void IInternalSessionControl::assignMachine(
            [in] IMachinemachine)
```

Assigns the machine object associated with this direct-type session or informs the session that it will be a remote one (if machine = NULL).

6.25.3 assignRemoteMachine

```
void IInternalSessionControl::assignRemoteMachine(
            [in] IMachinemachine,
            [in] IConsoleconsole)
```

Assigns the machine and the (remote) console object associated with this remote-type session.

6.25.4 enumerateGuestProperties

```
void IInternalSessionControl::enumerateGuestProperties(
            [in] wstring patterns,
            [out] wstring key[],
            [out] wstring value[],
            [out] unsigned long long timestamp[],
            [out] wstring flags[])
```

Return a list of the guest properties matching a set of patterns along with their values, timestamps and flags.

6.25.5 getPID

```
unsigned long IInternalSessionControl::getPID()
```

PID of the process that has created this Session object.

6.25.6 getRemoteConsole

```
IConsole IInternalSessionControl::getRemoteConsole()
```

Returns the console object suitable for remote control.

6.25.7 onDVDDriveChange

```
void IInternalSessionControl::onDVDDriveChange()
```

Triggered when settings of the DVD drive object of the associated virtual machine have changed.

6.25.8 onFloppyDriveChange

```
void IInternalSessionControl::onFloppyDriveChange()
```

Triggered when settings of the floppy drive object of the associated virtual machine have changed.

6.25.9 onNetworkAdapterChange

```
void IInternalSessionControl::onNetworkAdapterChange(
            [in] INetworkAdapternetworkAdapter)
```

Triggered when settings of a network adapter of the associated virtual machine have changed.

6.25.10 onParallelPortChange

```
void IInternalSessionControl::onParallelPortChange(
            [in] IParallelPortparallelPort)
```

Triggered when settings of a parallel port of the associated virtual machine have changed.

6.25.11 onSerialPortChange

```
void IInternalSessionControl::onSerialPortChange(
            [in] ISerialPortserialPort)
```

Triggered when settions of a serial port of the associated virtual machine have changed.

6.25.12 onSharedFolderChange

```
void IInternalSessionControl::onSharedFolderChange(
            [in] boolean 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.

6.25.13 onShowWindow

```
void IInternalSessionControl::onShowWindow(
            [in] boolean check,
            [out] boolean canShow,
            [out] unsigned long long winId)
```

Called by IMachine::canShowConsoleWindow() and by IMachine::showConsoleWindow() in order to notify console callbacks IConsoleCallback::onCanShowWindow() and IConsoleCallback::onShowWindow().

6.25.14 onUSBControllerChange

```
void IInternalSessionControl::onUSBControllerChange()
```

Triggered when settings of the USB controller object of the associated virtual machine have changed.

6.25.15 onUSBDeviceAttach

```
void IInternalSessionControl::onUSBDeviceAttach(
            [in] IUSBDevicedevice,
            [in] IVirtualBoxErrorInfoerror,
            [in] unsigned long 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 @c null @a error object means success, otherwise it describes a failure.

6.25.16 onUSBDeviceDetach

```
void IInternalSessionControl::onUSBDeviceDetach(
            [in] uuid id,
            [in] IVirtualBoxErrorInfoerror)
```

Triggered when a request to release the USB device (as a result of machine termination or direct call to IConsole::detachUSBDevice) has completed. A @c null @a error object means success, otherwise it

6.25.17 onVRDPServerChange

```
void IInternalSessionControl::onVRDPServerChange()
```

Triggered when settings of the VRDP server object of the associated virtual machine have changed.

6.25.18 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.

6.25.19 updateMachineState

```
void IInternalSessionControl::updateMachineState(
            [in] MachineStateaMachineState)
```

Updates the machine state in the VM process. Must be called only in certain cases (see the method implementation).

6.26 IKeyboard

The IKeyboard interface represents the virtual machine's keyboard. Used in IConsole::keyboard.

Through this interface, the virtual machine's virtual keyboard can be controlled. One can send keystrokes to the virtual machine and send the Ctrl-Alt-Del sequence to it.

6.26.1 putCAD

```
void IKeyboard::putCAD()
```

Sends the Ctrl-Alt-Del sequence to the keyboard.

6.26.2 putScancode

```
void IKeyboard::putScancode(
            [in] long scancode)
```

Sends a scancode to the keyboard.

6.26.3 putScancodes

```
unsigned long IKeyboard::putScancodes(
            [in] long scancodes[])
```

Sends an array of scancode to the keyboard.

6.27 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 explicitely 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 IVirtualBox::openSession, IVirtualBox::openRemoteSession or IVirtualBox::openExistingSession methdods. After the session has been successfully opened, 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 the IMachine interface 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 IConsole interface. Refer to the IConsole interface description to get more information about this topic.

See also: ISession, IConsole

6.27.1 Attributes

6.27.1.1 parent (read-only)

```
IVirtualBox IMachine::parent
```

Associated parent obect.

6.27.1.2 accessible (read-only)

```
boolean IMachine::accessible
```

Whether this virtual machine is currently accessible or not.

The machine is considered to be inaccessible when:

- It is a registered virtual machine, and
- Its settings file is inaccessible (for example, it is located on a network share that is not accessible during VirtualBox startup, or becomes inaccessible later, or if the settings file can be read but is invalid).

Otherwise, the value of this property is always `true`.

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.

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 IVirtualBox::unregisterMachine 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.

6.27.1.3 accessError (read-only)

`IVirtualBoxErrorInfo IMachine::accessError`

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

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 unaccessible). Otherwise, a null IVirtualBoxErrorInfo object will be returned.

6.27.1.4 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.

Note: For "legacy" machines created using the IVirtualBox::createLegacyMachine() call, the above naming limitations do not apply because the machine name does not affect the settings file name. The settings file name remains the same as it was specified during machine creation and never changes.

6.27.1.5 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).

6.27.1.6 id (read-only)

`uuid IMachine::id`

UUID of the virtual machine.

6.27.1.7 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.

6.27.1.8 memorySize (read/write)

`unsigned long IMachine::memorySize`

System memory size in megabytes.

6.27.1.9 memoryBalloonSize (read/write)

`unsigned long IMachine::memoryBalloonSize`

Initial memory balloon size in megabytes.

6.27.1.10 statisticsUpdateInterval (read/write)

`unsigned long IMachine::statisticsUpdateInterval`

Initial interval to update guest statistics in seconds.

6.27.1.11 VRAMSize (read/write)

`unsigned long IMachine::VRAMSize`

Video memory size in megabytes.

6.27.1.12 MonitorCount (read/write)

`unsigned long IMachine::MonitorCount`

Number of virtual monitors.

> **Note:** Only effective on Windows XP and later guests with Guest Additions installed.

6.27.1.13 BIOSSettings (read-only)

`IBIOSSettings IMachine::BIOSSettings`

Object containing all BIOS settings.

6.27.1.14 HWVirtExEnabled (read/write)

`TSBool IMachine::HWVirtExEnabled`

This setting determines whether VirtualBox will try to make use of the host CPU's hardware virtualization extensions such as Intel VT-x and AMD-V. Note that in case such extensions are not available, they will not be used.

6.27.1.15 HWVirtExNestedPagingEnabled (read/write)

`boolean IMachine::HWVirtExNestedPagingEnabled`

This setting determines whether VirtualBox will try to make use of the nested paging extension of Intel VT-x and AMD-V. Note that in case such extensions are not available, they will not be used.

6.27.1.16 PAEEnabled (read/write)

`boolean IMachine::PAEEnabled`

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.

6.27.1.17 snapshotFolder (read/write)

`wstring IMachine::snapshotFolder`

Full path to the directory used to store snapshot data (difrerencing hard disks 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` 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.

6.27.1.18 VRDPServer (read-only)

`IVRDPServer IMachine::VRDPServer`

VRDP server object.

6.27.1.19 hardDiskAttachments (read-only)

`IHardDiskAttachmentCollection IMachine::hardDiskAttachments`

Collection of hard disks attached to the machine.

6.27.1.20 DVDDrive (read-only)

`IDVDDrive IMachine::DVDDrive`

Associated DVD drive object.

6.27.1.21 FloppyDrive (read-only)

`IFloppyDrive IMachine::FloppyDrive`

Associated floppy drive object.

6.27.1.22 USBController (read-only)

`IUSBController IMachine::USBController`

Associated USB controller object.

Note: This method may set a @ref com_warnings “warning result code”.

Note: If USB functionality is not avaliable in the given edition of VirtualBox, this method will set the result code to @c E_NOTIMPL.

6.27.1.23 audioAdapter (read-only)

`IAudioAdapter IMachine::audioAdapter`

Associated audio adapter, always present.

6.27.1.24 SATAController (read-only)

```
ISATAController IMachine::SATAController
```

Associated SATA controller object.

6.27.1.25 settingsFilePath (read-only)

```
wstring IMachine::settingsFilePath
```

Full name of the file containing machine settings data.

6.27.1.26 settingsFileVersion (read-only)

```
wstring IMachine::settingsFileVersion
```

Current version of the format of the settings file of this machine (settingsFilePath).
The version string has the following format:

```
x.y-platform
```

where `x` and `y` are the major and the minor format versions, and `platform` is the platform identifier.

The current version usually matches the value of the IVirtualBox::settingsFormatVersion attribute unless the settings file was created by an older version of VirtualBox and there was a change of the settings file format since then.

Note that VirtualBox automatically converts settings files from older versions to the most recent version when reading them (usually at VirtualBox startup) but it doesn't save the changes back until you call a method that implicitly saves settings (such as setExtraData()) or call saveSettings() explicitly. Therefore, if the value of this attribute differs from the value of IVirtualBox::settingsFormatVersion, then it means that the settings file was converted but the result of the conversion is not yet saved to disk.

The above feature may be used by interactive front-ends to inform users about the settings file format change and offer them to explicitly save all converted settings files (the global and VM-specific ones), optionally create bacup copies of the old settings files before saving, etc.

See also: IVirtualBox::settingsFormatVersion, saveSettingsWithBackup()

6.27.1.27 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 IVirtualBox::unregisterMachine. For all other cases, the settigs 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).

6.27.1.28 sessionState (read-only)

```
SessionState IMachine::sessionState
```

Current session state for this machine.

6.27.1.29 sessionType (read-only)

```
wstring IMachine::sessionType
```

Type of the session. If sessionState is SessionSpawning or SessionOpen, this attribute contains the same value as passed to the IVirtualBox::openRemoteSession() method in the @a type parameter. If the session was opened directly using IVirtualBox::openSession(), or if sessionState is SessionClosed, the value of this attribute is @c null.

6.27.1.30 sessionPid (read-only)

```
unsigned long IMachine::sessionPid
```

Identifier of the session process. This attribute contains the platform-dependent identifier of the process that has opened a direct session for this machine using the IVirtualBox::openSession() call. The returned value is only valid if sessionState is SessionOpen or SessionClosing (i.e. a session is currently open or being closed) by the time this property is read.

6.27.1.31 state (read-only)

```
MachineState IMachine::state
```

Current execution state of this machine.

6.27.1.32 lastStateChange (read-only)

`long long IMachine::lastStateChange`

Time stamp of the last execution state change, in milliseconds since 1970-01-01 UTC.

6.27.1.33 stateFilePath (read-only)

`wstring IMachine::stateFilePath`

Full path to the file that stores the execution state of the machine when it is in the MachineState::Saved state.

> **Note:** When the machine is not in the Saved state, this attribute `null`.

6.27.1.34 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 (upto `VBox.log.3` in the current version).

6.27.1.35 currentSnapshot (read-only)

`ISnapshot IMachine::currentSnapshot`

Current snapshot of this machine.

> **Note:** A `null` object is returned if the machine doesn't have snapshots.

See also: ISnapshot

6.27.1.36 snapshotCount (read-only)

`unsigned long IMachine::snapshotCount`

Number of snapshots taken on this machine. Zero means the machine doesn't have any snapshots.

6.27.1.37 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 right after one of the following calls are made:

- IConsole::discardCurrentState or IConsole::discardCurrentSnapshotAndState
- IConsole::takeSnapshot (issued on a powered off or saved machine, for which settingsModified returns `false`)
- IMachine::setCurrentSnapshot

The current state remains identical until one of the following happens:

- settings of the machine are changed
- the saved state is discarded
- the current snapshot is discarded
- an attempt to execute the machine is made

Note: For machines that don't have snapshots, this property is always `false`.

6.27.1.38 sharedFolders (read-only)

`ISharedFolderCollection 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.

6.27.1.39 clipboardMode (read/write)

`ClipboardMode IMachine::clipboardMode`

Synchronization mode between the host OS clipboard and the guest OS clipboard.

6.27.2 attachHardDisk

```
void IMachine::attachHardDisk(
            [in] uuid id,
            [in] StorageBusbus,
            [in] long channel,
            [in] long device)
```

Attaches a virtual hard disk identified by the given UUID to the given device slot of the given channel on the given bus. The specified device slot must not have another disk attached and the given hard disk must not be already attached to this machine.

See IHardDisk for detailed information about attaching hard disks.

> **Note:** You cannot attach a hard disk to a running machine. Also, you cannot attach a hard disk to a newly created machine until it is registered.

> **Note:** Attaching a hard disk to a machine creates a *lazy* attachment. In particular, no differeincing images are actually created until saveSettings() is called to commit all changed settings.

6.27.3 canShowConsoleWindow

```
boolean IMachine::canShowConsoleWindow()
```

Returns @c 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.

6.27.4 createSharedFolder

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

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.

6.27.5 deleteSettings

```
void IMachine::deleteSettings()
```

Deletes the settings file of this machine from disk. The machine must not be registered in order for this operation to succeed.

> **Note:** settingsModified will return TRUE after this method successfully returns.

> **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 IVirtualBox::unregisterMachine.

> **Note:** The deleted machine settings file can be restored (saved again) by calling saveSettings().

6.27.6 detachHardDisk

```
void IMachine::detachHardDisk(
            [in] StorageBusbus,
            [in] long channel,
            [in] long device)
```

Detaches the hard disk drive attached to the given device slot of the given controller. See IHardDisk for detailed information about attaching hard disks.

> **Note:** You cannot detach a hard disk from a running machine.

> **Note:** Detaching a hard disk from a machine creates a *lazy* detachment. In particular, if the detached hard disk is a differencing hard disk, it is not actually deleted until saveSettings() is called to commit all changed settings. Keep in mind, that doing saveSettings() will **physically delete** all detached differencing hard disks, so be careful.

6.27.7 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 IVirtualBox::unregisterMachine.

6.27.8 enumerateGuestProperties

```
void IMachine::enumerateGuestProperties(
            [in] wstring patterns,
            [out] wstring name[],
            [out] wstring value[],
            [out] unsigned long long timestamp[],
            [out] wstring flags[])
```

Return a list of the guest properties matching a set of patterns along with their values, timestamps and flags.

6.27.9 findSnapshot

```
ISnapshot IMachine::findSnapshot(
            [in] wstring name)
```

Returns a snapshot of this machine with the given name.

6.27.10 getBootOrder

```
DeviceType IMachine::getBootOrder(
            [in] unsigned long order)
```

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 DeviceType::Null is returned.

@todo getHardDiskBootOrder(), getNetworkBootOrder()

6.27.11 getExtraData

```
wstring IMachine::getExtraData(
            [in] wstring key)
```

Returns associated machine-specific extra data.

If the reuqested data @a key does not exist, this function will succeed and return @c NULL in the @a value argument.

6.27.12 getGuestProperty

```
void IMachine::getGuestProperty(
            [in] wstring name,
            [out] wstring value,
            [out] unsigned long long timestamp,
            [out] wstring flags)
```

Reads an entry from the machine's guest property store.

6.27.13 getGuestPropertyTimestamp

```
unsigned long long IMachine::getGuestPropertyTimestamp(
            [in] wstring property)
```

Reads a property timestamp from the machine's guest property store.

6.27.14 getGuestPropertyValue

```
wstring IMachine::getGuestPropertyValue(
            [in] wstring property)
```

Reads a value from the machine's guest property store.

6.27.15 getHardDisk

```
IHardDisk IMachine::getHardDisk(
            [in] StorageBusbus,
            [in] long channel,
            [in] long device)
```

Returns the hard disk attached to the given controller under the specified device number.

6.27.16 getNetworkAdapter

```
INetworkAdapter IMachine::getNetworkAdapter(
            [in] unsigned long slot)
```

Returns the network adapter associated with the given slot. Slots are numbered sequentially, starting with zero. The total number of adapters per every machine is defined by the ISystemProperties::networkAdapterCount property, so the maximum slot number is one less than that property's value.

6.27.17 getNextExtraDataKey

```
void IMachine::getNextExtraDataKey(
            [in] wstring key,
            [out] wstring nextKey,
            [out] wstring nextValue)
```

Returns the machine-specific extra data key name following the supplied key.

An error is returned if the supplied @a key does not exist. @c NULL is returned in @a nextKey if the supplied key is the last key. When supplying @c NULL for the @a key, the first key item is returned in @a nextKey (if there is any). @a nextValue is an optional parameter and if supplied, the next key's value is returned in it.

6.27.18 getParallelPort

```
IParallelPort IMachine::getParallelPort(
            [in] unsigned long slot)
```

Returns the parallel port associated with the given slot. Slots are numbered sequentially, starting with zero. The total number of parallel ports per every machine is defined by the ISystemProperties::parallelPortCount property, so the maximum slot number is one less than that property's value.

6.27.19 getSerialPort

```
ISerialPort IMachine::getSerialPort(
            [in] unsigned long slot)
```

Returns the serial port associated with the given slot. Slots are numbered sequentially, starting with zero. The total number of serial ports per every machine is defined by the ISystemProperties::serialPortCount property, so the maximum slot number is one less than that property's value.

6.27.20 getSnapshot

```
ISnapshot IMachine::getSnapshot(
            [in] uuid id)
```

Returns a snapshot of this machine with the given UUID. A `null` UUID can be used to obtain the first snapshot taken on this machine. This is useful if you want to traverse the whole tree of snapshots starting from the root.

6.27.21 removeSharedFolder

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

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

6.27.22 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 IVirtualBoxCallback::onMachineDataChange() 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 IVirtualBox::unregisterMachine.

6.27.23 saveSettingsWithBackup

```
wstring IMachine::saveSettingsWithBackup()
```

Creates a backup copy of the machine settings file (settingsFilePath) in case of auto-conversion, and then calls saveSettings().

Note that the backup copy is created **only** if the settings file auto-conversion took place (see settingsFileVersion for details). Otherwise, this call is fully equivalent to saveSettings() and no backup copying is done.

The backup copy is created in the same directory where the original settings file is located. It is given the following file name:

```
original.xml.x.y-platform.bak
```

where `original.xml` is the original settings file name (excluding path), and `x.y-platform` is the version of the old format of the settings file (before auto-conversion).

If the given backup file already exists, this method will try to add the `.N` suffix to the backup file name (where `N` counts from 0 to 9) and copy it again until it succeeds. If all suffixes are occupied, or if any other copy error occurs, this method will return a failure.

If the copy operation succeeds, the @a bakFileName return argument will receive a full path to the created backup file (for informational purposes). Note that this will happen even if the subsequent saveSettings() call performed by this method after the copy operation, fails.

> **Note:** The VirtualBox API never calls this method. It is intended purely for the purposes of creating backup copies of the settings files by front-ends before saving the results of the automatically performed settings conversion to disk.

See also: settingsFileVersion

6.27.24 setBootOrder

```
void IMachine::setBootOrder(
            [in] unsigned long position,
            [in] DeviceTypedevice)
```

Puts the given device to the specified position in the boot order.

To indicate that no device is associated with the given position, DeviceType::Null should be used.

@todo setHardDiskBootOrder(), setNetworkBootOrder()

6.27.25 setCurrentSnapshot

```
void IMachine::setCurrentSnapshot(
            [in] uuid id)
```

Sets the current snapshot of this machine.

Note: In the current implementation, this operation is not implemented.

6.27.26 setExtraData

```
void IMachine::setExtraData(
            [in] wstring key,
            [in] wstring value)
```

Sets associated machine-specific extra data.
If you pass @c NULL as a key @a vaule, the given @a key will be deleted.

Note: Before performing the actual data change, this method will ask all registered callbacks using the IVirtualBoxCallback::onExtraDataCanChange() notification for a permission. If one of the callbacks refuses the new value, the change will not be performed.

Note: On success, the IVirtualBoxCallback::onExtraDataChange() notification is called to inform all registered callbacks 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.

6.27.27 setGuestProperty

```
void IMachine::setGuestProperty(
            [in] wstring property,
            [in] wstring value,
            [in] wstring flags)
```

Sets, changes or deletes an entry in the machine's guest property store.

6.27.28 setGuestPropertyValue

```
void IMachine::setGuestPropertyValue(
            [in] wstring property,
            [in] wstring value)
```

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.

6.27.29 showConsoleWindow

```
unsigned 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.

6.28 IMachineDebugger

Note: This interface is not supported in the webservice.

6.28.1 Attributes

6.28.1.1 singlestep (read/write)

```
boolean IMachineDebugger::singlestep
```

Switch for enabling singlestepping.

6.28.1.2 recompileUser (read/write)

```
boolean IMachineDebugger::recompileUser
```

Switch for forcing code recompilation for user mode code.

6.28.1.3 recompileSupervisor (read/write)

```
boolean IMachineDebugger::recompileSupervisor
```

Switch for forcing code recompilation for supervisor mode code.

6.28.1.4 PATMEnabled (read/write)

```
boolean IMachineDebugger::PATMEnabled
```

Switch for enabling and disabling the PATM component.

6.28.1.5 CSAMEnabled (read/write)

`boolean IMachineDebugger::CSAMEnabled`

Switch for enabling and disabling the CSAM component.

6.28.1.6 logEnabled (read/write)

`boolean IMachineDebugger::logEnabled`

Switch for enabling and disabling logging.

6.28.1.7 HWVirtExEnabled (read-only)

`boolean IMachineDebugger::HWVirtExEnabled`

Flag indicating whether the VM is currently making use of CPU hardware virtualization extensions.

6.28.1.8 HWVirtExNestedPagingEnabled (read-only)

`boolean IMachineDebugger::HWVirtExNestedPagingEnabled`

Flag indicating whether the VM is currently making use of the nested paging CPU hardware virtualization extension.

6.28.1.9 PAEEnabled (read-only)

`boolean IMachineDebugger::PAEEnabled`

Flag indicating whether the VM is currently making use of the Physical Address Extension CPU feature.

6.28.1.10 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%.

6.28.1.11 VM (read-only)

`unsigned long long IMachineDebugger::VM`

Gets the VM handle. This is only for internal use while we carve the details of this interface.

6.28.2 dumpStats

```
void IMachineDebugger::dumpStats(
            [in] wstring pattern)
```

Dumps VM statistics.

6.28.3 getStats

```
void IMachineDebugger::getStats(
            [in] wstring pattern,
            [in] boolean withDescriptions,
            [out] wstring stats)
```

Get the VM statistics in a XMLish format.

6.28.4 resetStats

```
void IMachineDebugger::resetStats(
            [in] wstring pattern)
```

Reset VM statistics.

6.29 IManagedObjectRef

Note: This interface is supported in the webservice 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 IManagedObjectRef::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.

6.29.1 getInterfaceName

```
wstring IManagedObjectRef::getInterfaceName()
```

Returns the name of the interface that this managed object represents, for example, "IMachine", as a string.

6.29.2 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.

6.30 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.

6.30.1 Attributes

6.30.1.1 absoluteSupported (read-only)

```
boolean IMouse::absoluteSupported
```

Whether the guest OS supports absolute mouse pointer positioning or not.

> **Note:** VirtualBox Guest Tools need to be installed to the guest OS in order to enable absolute mouse positioning support. You can use the IConsole-Callback::onMouseCapabilityChange callback to be instantly informed about changes of this attribute during virtual machine execution.

See also: putMouseEventAbsolute

6.30.2 putMouseEvent

```
void IMouse::putMouseEvent(
            [in] long dx,
            [in] long dy,
            [in] long dz,
            [in] long buttonState)
```

Initiates a mouse event using relative pointer movements along x and y axis.

6.30.3 putMouseEventAbsolute

```
void IMouse::putMouseEventAbsolute(
            [in] long x,
            [in] long y,
            [in] long dz,
            [in] long buttonState)
```

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

6.31 INetworkAdapter

6.31.1 Attributes

6.31.1.1 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.

6.31.1.2 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.

6.31.1.3 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.

6.31.1.4 MACAddress (read/write)

```
wstring INetworkAdapter::MACAddress
```

Ethernet MAC address of the adapter, 12 hexadecimal characters. When setting it to NULL, VirtualBox will generate a unique MAC address.

6.31.1.5 attachmentType (read-only)

```
NetworkAttachmentType INetworkAdapter::attachmentType
```

6.31.1.6 hostInterface (read/write)

```
wstring INetworkAdapter::hostInterface
```

Name of the Host Network Interface that is currently in use. NULL will be returned if no device has been allocated. On Linux, setting this refers to a permanent TAP device. However, a file descriptor has precedence over the interface name on Linux. Note that when VBox allocates a TAP device, this property will not be set, i.e. the interface name would have to be determined using the file descriptor and /proc/self/fd.

6.31.1.7 internalNetwork (read/write)

```
wstring INetworkAdapter::internalNetwork
```

Name of the internal network the VM is attached to.

6.31.1.8 NATNetwork (read/write)

```
wstring INetworkAdapter::NATNetwork
```

Name of the NAT network the VM is attached to.

6.31.1.9 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.

6.31.1.10 lineSpeed (read/write)

```
unsigned long INetworkAdapter::lineSpeed
```

Line speed reported by custom drivers, in units of 1 kbps.

6.31.1.11 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.

6.31.1.12 traceFile (read/write)

```
wstring INetworkAdapter::traceFile
```

Filename where a network trace will be stored. If not set, VBox-pid.pcap will be used.

6.31.2 attachToHostInterface

```
void INetworkAdapter::attachToHostInterface()
```

Attach the network adapter to a host interface. On Linux, the TAP setup application will be executed if configured and unless a device name and/or file descriptor has been set, a new TAP interface will be created.

6.31.3 attachToInternalNetwork

```
void INetworkAdapter::attachToInternalNetwork()
```

Attach the network adapter to an internal network.

6.31.4 attachToNAT

```
void INetworkAdapter::attachToNAT()
```

Attach the network adapter to the Network Address Translation (NAT) interface.

6.31.5 detach

```
void INetworkAdapter::detach()
```

Detach the network adapter

6.32 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

6.32.1 Attributes

6.32.1.1 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.

6.32.1.2 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.

6.32.1.3 IOBase (read/write)

```
unsigned long IParallelPort::IOBase
```

Base I/O address of the parallel port.

6.32.1.4 IRQ (read/write)

```
unsigned long IParallelPort::IRQ
```

IRQ number of the parallel port.

6.32.1.5 path (read/write)

```
wstring IParallelPort::path
```

Host parallel device name. If this parallel port is enabled, setting a @c null or an empty string as this attribute's value will result into an error.

6.33 IPerformanceCollector

The IPerformanceCollector interface represents a service that collects and stores performance metrics data.

Performance metrics are associated with objects like IHost and IMachine. Each object has a distinct set of performance metrics. The set can be obtained with IPerformanceCollector::getMetrics.

Metric data are collected at the specified intervals and are retained internally. The interval and the number of samples retained can be set with IPerformanceCollector::setMetrics.

Metrics are organized hierarchically, each level separated by slash (/). General scheme for metric name is "Category/Metric[/SubMetric][:aggregation]". For example CPU/Load/User:avg metric name stands for: 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

"Category/Metric" together form 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 IPerformanceCollector::queryMetricsData. 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 IPerfromanceCollector 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 IPerformanceCollector::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 IPerformanceCollector::queryMetricsData. The data that have been collected so far are returned. Note that the values are still retained internally and data collection continues.

6.33.1 Attributes

6.33.1.1 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. IPerformanceCollector::getMetrics can be used to get the list of supported metrics for a particular object.

6.33.2 disableMetrics

```
void IPerformanceCollector::disableMetrics(
            [in] wstring metricNames[],
            [in] $unknown objects[])
```

Turns off collecting specified base metrics.

> **Note:** @c Null or empty metric name array means all metrics. @c 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.

6.33.3 enableMetrics

```
void IPerformanceCollector::enableMetrics(
            [in] wstring metricNames[],
            [in] $unknown objects[])
```

Turns on collecting specified base metrics.

> **Note:** @c Null or empty metric name array means all metrics. @c 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.

6.33.4 getMetrics

```
IPerformanceMetric IPerformanceCollector::getMetrics(
            [in] wstring metricNames[],
            [in] $unknown objects[])
```

Returns parameters of specified metrics for a set of objects.

> **Note:** @c Null metrics array means all metrics. @c Null object array means all existing objects.

6.33.5 queryMetricsData

```
long IPerformanceCollector::queryMetricsData(
            [in] wstring metricNames[],
            [in] $unknown objects[],
            [out] wstring returnMetricNames[],
            [out] $unknown returnObjects[],
            [out] unsigned long returnDataIndices[],
            [out] unsigned long returnDataLengths[])
```

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 sub-array is indicated by `returnDataIndices` and `returnDataLengths`. The first value for metric `metricNames[i]` is at `returnData[returnIndices[i]]`.

> **Note:** @c Null or empty metric name array means all metrics. @c 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 @c 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 @c interval*count period to avoid confusion. This way a completely new set of data values will be provided by each query.

6.33.6 setupMetrics

```
void IPerformanceCollector::setupMetrics(
            [in] wstring metricNames[],
            [in] $unknown objects[],
            [in] unsigned long period,
            [in] unsigned long count)
```

Sets parameters of specified base metrics for a set of objects. Returns an array of IPerformanceMetric describing the metrics have been affected.

> **Note:** @c Null or empty metric name array means all metrics. @c 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.

6.34 IPerformanceMetric

The IPerformanceMetric interface represents parameters of the given performance metric.

6.34.1 Attributes

6.34.1.1 metricName (read-only)

```
wstring IPerformanceMetric::metricName
```

Name of the metric.

6.34.1.2 object (read-only)

```
$unknown IPerformanceMetric::object
```

Object this metric belongs to.

6.34.1.3 description (read-only)

```
wstring IPerformanceMetric::description
```

Textual description of the metric.

6.34.1.4 period (read-only)

```
unsigned long IPerformanceMetric::period
```

Time interval between samples, measured in seconds.

6.34.1.5 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.

6.34.1.6 unit (read-only)

```
wstring IPerformanceMetric::unit
```

Unit of measurement.

6.34.1.7 minimumValue (read-only)

```
long IPerformanceMetric::minimumValue
```

Minimum possible value of this metric.

6.34.1.8 maximumValue (read-only)

```
long IPerformanceMetric::maximumValue
```

Maximum possible value of this metric.

6.35 IProgress

The IProgress interface represents a task progress object that allows to wait for the completion of some asynchronous task.

The task consists of one or more operations that run sequentially, one after one. There is an individual percent of completion of the current operation and the percent of completion of the task as a whole. Similarly, you can wait for the completion of a particular operation or for the completion of the whole task.

Every operation is identified by a number (starting from 0) and has a separate description.

6.35.1 Attributes

6.35.1.1 id (read-only)

```
uuid IProgress::id
```

ID of the task.

6.35.1.2 description (read-only)

```
wstring IProgress::description
```

Description of the task.

6.35.1.3 initiator (read-only)

```
$unknown IProgress::initiator
```

Initiator of the task.

6.35.1.4 cancelable (read-only)

```
boolean IProgress::cancelable
```

Whether the task can be interrupted.

6.35.1.5 percent (read-only)

```
long IProgress::percent
```

Current task progress value in percent. This value depends on how many operations are already complete.

6.35.1.6 completed (read-only)

```
boolean IProgress::completed
```

Whether the task has been completed.

6.35.1.7 canceled (read-only)

```
boolean IProgress::canceled
```

Whether the task has been canceled.

6.35.1.8 resultCode (read-only)

```
result IProgress::resultCode
```

Result code of the progress task. Valid only if completed is true.

6.35.1.9 errorInfo (read-only)

```
IVirtualBoxErrorInfo IProgress::errorInfo
```

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

Extended information about the unsuccessful result of the progress operation. May be NULL when no extended information is available. Valid only if completed is true and resultCode indicates a failure.

6.35.1.10 operationCount (read-only)

```
unsigned long IProgress::operationCount
```

Number of operations this task is divided into. Every task consists of at least one operation.

6.35.1.11 operation (read-only)

```
unsigned long IProgress::operation
```

Number of the operation being currently executed.

6.35.1.12 operationDescription (read-only)

```
wstring IProgress::operationDescription
```

Description of the operation being currently executed.

6.35.1.13 operationPercent (read-only)

```
long IProgress::operationPercent
```

Current operation progress value in percent.

6.35.2 cancel

```
void IProgress::cancel()
```

Cancels the task.

Note: If cancelable is `false`, then this method will fail.

6.35.3 waitForCompletion

```
void IProgress::waitForCompletion(
            [in] long timeout)
```

Waits until the task is done (including all operations) with a given timeout.

6.35.4 waitForOperationCompletion

```
void IProgress::waitForOperationCompletion(
            [in] unsigned long operation,
            [in] long timeout)
```

Waits until the given operation is done with a given timeout.

6.36 IRemoteDisplayInfo

Note: With the webservice, 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 display (VRDP) capabilities and status. This is used in the IConsole::remoteDisplayInfo attribute.

6.36.1 Attributes

6.36.1.1 active (read-only)

```
boolean IRemoteDisplayInfo::active
```

Whether the remote display connection is active.

6.36.1.2 numberOfClients (read-only)

```
unsigned long IRemoteDisplayInfo::numberOfClients
```

How many times a client connected.

6.36.1.3 beginTime (read-only)

```
long long IRemoteDisplayInfo::beginTime
```

When the last connection was established, in milliseconds since 1970-01-01 UTC.

6.36.1.4 endTime (read-only)

```
long long IRemoteDisplayInfo::endTime
```

When the last connection was terminated or the current time, if connection is still active, in milliseconds since 1970-01-01 UTC.

6.36.1.5 bytesSent (read-only)

```
unsigned long long IRemoteDisplayInfo::bytesSent
```

How many bytes were sent in last or current, if still active, connection.

6.36.1.6 bytesSentTotal (read-only)

```
unsigned long long IRemoteDisplayInfo::bytesSentTotal
```

How many bytes were sent in all connections.

6.36.1.7 bytesReceived (read-only)

```
unsigned long long IRemoteDisplayInfo::bytesReceived
```

How many bytes were received in last or current, if still active, connection.

6.36.1.8 bytesReceivedTotal (read-only)

```
unsigned long long IRemoteDisplayInfo::bytesReceivedTotal
```

How many bytes were received in all connections.

6.36.1.9 user (read-only)

```
wstring IRemoteDisplayInfo::user
```

Login user name supplied by the client.

6.36.1.10 domain (read-only)

```
wstring IRemoteDisplayInfo::domain
```

Login domain name supplied by the client.

6.36.1.11 clientName (read-only)

```
wstring IRemoteDisplayInfo::clientName
```

The client name supplied by the client.

6.36.1.12 clientIP (read-only)

```
wstring IRemoteDisplayInfo::clientIP
```

The IP address of the client.

6.36.1.13 clientVersion (read-only)

```
unsigned long IRemoteDisplayInfo::clientVersion
```

The client software version number.

6.36.1.14 encryptionStyle (read-only)

```
unsigned long IRemoteDisplayInfo::encryptionStyle
```

Public key exchange method used when connection was established. Values: 0 - RDP4 public key exchange scheme. 1 - X509 sertificates were sent to client.

6.37 ISATAController

6.37.1 Attributes

6.37.1.1 enabled (read/write)

```
boolean ISATAController::enabled
```

Flag whether the SATA controller is present in the guest system. If disabled, the virtual guest hardware will not contain any SATA controller. Can only be changed when the VM is powered off.

6.37.1.2 portCount (read/write)

```
unsigned long ISATAController::portCount
```

The number of usable ports on the sata controller. It ranges from 1 to 30.

6.37.2 GetIDEEmulationPort

```
long ISATAController::GetIDEEmulationPort(
            [in] long devicePosition)
```

Gets the corresponding port number which is emulated as an IDE device.

6.37.3 SetIDEEmulationPort

```
void ISATAController::SetIDEEmulationPort(
            [in] long devicePosition,
            [in] long portNumber)
```

Sets the port number which is emulated as an IDE device.

6.38 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

6.38.1 Attributes

6.38.1.1 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.

6.38.1.2 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.

6.38.1.3 IOBase (read/write)

```
unsigned long ISerialPort::IOBase
```

Base I/O address of the serial port.

6.38.1.4 IRQ (read/write)

```
unsigned long ISerialPort::IRQ
```

IRQ number of the serial port.

6.38.1.5 hostMode (read/write)

```
PortMode ISerialPort::hostMode
```

How is this port connected to the host.

6.38.1.6 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.

6.38.1.7 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. In either of the above cases, setting a @c null or an empty string as the attribute's value will result into an error. Otherwise, the value of this property is ignored.

6.39 ISession

The ISession interface represents a serialization primitive for virtual machines.

With VirtualBox, every time one wishes to manipulate a virtual machine (e.g. change its settings or start execution), a session object is required. Such an object must be passed to one of the session methods that open the given session, which then initiates the machine manipulation.

A session serves several purposes: it identifies to the inter-process VirtualBox code which process is currently working with the virtual machine, and it ensures that there are no incompatible requests from several processes for the same virtual machine. Session objects can therefore be thought of as mutex semaphores that lock virtual machines to prevent conflicting accesses from several processes.

How sessions objects are used 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 one session object automatically when IWebsessionManager::logon is called. A managed object reference to that session object can be retrieved by calling IWebsessionManager::getSessionObject. This session object reference can then be used to open sessions.

Sessions are mainly used in two variations:

- To start a virtual machine in a separate process, one would call IVirtualBox::openRemoteSession, which requires a session object as its first parameter. This session then identifies the caller and lets him control the started machine (for example, pause machine execution or power it down) as well as be notified about machine execution state changes.
- To alter machine settings, or to start machine execution within the current process, one needs to open a direct session for the machine first by calling IVirtualBox::openSession. While a direct session is open within one process, no any

other process may open another direct session for the same machine. This prevents the machine from being changed by other processes while it is running or while the machine is being configured.

One also can attach to an existing direct session alreay opened by another process (for example, in order to send a control request to the virtual machine such as the pause or the reset request). This is done by calling IVirtualBox::openExistingSession.

Note: Unless you are trying to write a new VirtualBox front-end that performs direct machine execution (like the VirtualBox or VBoxSDL front-ends), don't call IConsole::powerUp in a direct session opened by IVirtualBox::openSession and use this session only to change virtual machine settings. If you simply want to start virtual machine execution using one of the existing front-ends (for example the VirtualBox GUI or headless server), simply use IVirtualBox::openRemoteSession; these front-ends will power up the machine automatically for you.

6.39.1 Attributes

6.39.1.1 state (read-only)

`SessionState ISession::state`

Current state of this session.

6.39.1.2 type (read-only)

`SessionType ISession::type`

Type of this session. The value of this attribute is valid only if the session is currently open (i.e. its #state is SessionType::SessionOpen), otherwise an error will be returned.

6.39.1.3 machine (read-only)

`IMachine ISession::machine`

Machine object associated with this session.

6.39.1.4 console (read-only)

`IConsole ISession::console`

Console object associated with this session.

6.39.2 close

```
void ISession::close()
```

Closes a session that was previously opened.

It is recommended that every time an "open session" method (such as IVirtualBox::openRemoteSession or IVirtualBox::openSession) has been called to manipulate a virtual machine, the caller invoke ISession::close() when it's done doing so. Since sessions are serialization primitives much like ordinary mutexes, they are best used the same way: for each "open" call, there should be a matching "close" call, even when errors occur.

Otherwise, if a direct session for a machine opened with IVirtualBox::openSession() is not explicitly closed when the application terminates, the state of the machine will be set to MachineState::Aborted on the server.

Generally, it is recommended to close all open sessions explicitly before terminating the application (no matter what is the reason of the termination).

> **Note:** Do not expect the session state (ISession::state to return to "Closed" immediately after you invoke ISession::close(), particularly if you have started a remote session to execute the VM in a new process. The session state will automatically return to "Closed" once the VM is no longer executing, which can of course take a very long time.

6.40 ISharedFolder

> **Note:** With the webservice, 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 definitioin 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 cretaing 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 prevoious (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 vesion of the product.

6.40.1 Attributes

6.40.1.1 name (read-only)

```
wstring ISharedFolder::name
```

Logical name of the shared folder.

6.40.1.2 hostPath (read-only)

```
wstring ISharedFolder::hostPath
```

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

6.40.1.3 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 unaccessible if it is placed on the network share that is not available by the time this property is read.

6.40.1.4 writable (read-only)

`boolean ISharedFolder::writable`

Whether the folder defined by the host path is writable or not.

6.41 ISnapshot

The ISnapshot interface represents a snapshot of the virtual machine.

The *snapshot* stores all the information about a virtual machine necessary to bring it to exactly the same state as it was at the time of taking the snapshot. The snapshot includes:

- all settings of the virtual machine (i.e. its hardware configuration: RAM size, attached hard disks, etc.)
- the execution state of the virtual machine (memory contents, CPU state, etc.).

Snapshots can be *offline* (taken when the VM is powered off) or *online* (taken when the VM is running). The execution state of the offline snapshot is called a *zero execution state* (it doesn't actually contain any information about memory contents or the CPU state, assuming that all hardware is just powered off).

Snapshot branches

Snapshots can be chained. Chained snapshots form a branch where every next snapshot is based on the previous one. This chaining is mostly related to hard disk branching (see IHardDisk description). This means that every time a new snapshot is created, a new differencing hard disk is implicitly created for all normal hard disks attached to the given virtual machine. This allows to fully restore hard disk contents when the machine is later reverted to a particular snapshot.

In the current implelemtation, multiple snapshot branches within one virtual machine are not allowed. Every machine has a signle branch, and IConsole::takeSnapshot() operation adds a new snapshot to the top of that branch.

Existings snapshots can be discarded using IConsole::discardSnapshot().

Current snapshot

Every virtual machine has a current snapshot, identified by IMachine::currentSnapshot. This snapshot is used as a base for the *current machine state* (see below), to the effect that all normal hard disks of the machine and its execution state are based on this snapshot.

In the current implementation, the current snapshot is always the last taken snapshot (i.e. the head snapshot on the branch) and it cannot be changed.

The current snapshot is `null` if the machine doesn't have snapshots at all; in this case the current machine state is just current settings of this machine plus its current execution state.

Current machine state

The current machine state is what represened by IMachine instances got directly from IVirtualBox using getMachine(), findMachine(), etc. (as opposed to instances

returned by ISnapshot::machine). This state is always used when the machine is powered on.

The current machine state also includes the current execution state. If the machine is being currently executed (IMachine::state is MachineState::Running and above), its execution state is just what's happening now. If it is powered off (MachineState::PoweredOff or MachineState::Aborted), it has a zero execution state. If the machine is saved (MachineState::Saved), its execution state is what saved in the execution state file (IMachine::stateFilePath).

If the machine is in the saved state, then, next time it is powered on, its execution state will be fully restored from the saved state file and the execution will continue from the point where the state was saved.

Similarly to snapshots, the current machine state can be discarded using IConsole::discardCurrentState().

Taking and discarding snapshots

The table below briefly explains the meaning of every snapshot operation:

OperationMeaningRemarksIConsole::takeSnapshot()Save the current state of the virtual machine, including all settings, contents of normal hard disks and the current modifications to immutable hard disks (for online snapshots)The current state is not changed (the machine will continue execution if it is being executed when the snapshot is taken)IConsole::discardSnapshot()Forget the state of the virtual machine stored in the snapshot: dismiss all saved settings and delete the saved execution state (for online snapshots)Other snapshots (including child snapshots, if any) and the current state are not directly affectedIConsole::discardCurrentState()Restore the current state of the virtual machine from the state stored in the current snapshot, including all settings and hard disk contentsThe current state of the machine existed prior to this operation is lostIConsole::discardCurrentSnapshotAndState()Completely revert the virtual machine to the state it was in before the current snapshot has been takenThe current state, as well as the current snapshot, are lost

6.41.1 Attributes

6.41.1.1 id (read-only)

```
uuid ISnapshot::id
```

UUID of the snapshot.

6.41.1.2 name (read/write)

```
wstring ISnapshot::name
```

Short name of the snapshot.

6.41.1.3 description (read/write)

```
wstring ISnapshot::description
```

Optional description of the snapshot.

6.41.1.4 timeStamp (read-only)

`long long ISnapshot::timeStamp`

Time stamp of the snapshot, in milliseconds since 1970-01-01 UTC.

6.41.1.5 online (read-only)

`boolean ISnapshot::online`

`true` if this snapshot is an online snapshot and `false` otherwise.

> **Note:** 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 `null`.

6.41.1.6 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.

6.41.1.7 parent (read-only)

`ISnapshot ISnapshot::parent`

Parent snapshot (a snapshot this one is based on).

> **Note:** It's not an error to read this attribute on a snapshot that doesn't have a parent – a null object will be returned to indicate this.

6.41.1.8 children (read-only)

`ISnapshotCollection ISnapshot::children`

Child snapshots (all snapshots having this one as a parent).

> **Note:** In the current implementation, there can be only one child snapshot, or no children at all, meaning this is the last (head) snapshot.

6.42 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.

6.42.1 Attributes

6.42.1.1 minGuestRAM (read-only)

`unsigned long ISystemProperties::minGuestRAM`

Minium guest system memory in Megabytes.

6.42.1.2 maxGuestRAM (read-only)

`unsigned long ISystemProperties::maxGuestRAM`

Maximum guest system memory in Megabytes.

6.42.1.3 minGuestVRAM (read-only)

`unsigned long ISystemProperties::minGuestVRAM`

Minimum guest video memory in Megabytes.

6.42.1.4 maxGuestVRAM (read-only)

`unsigned long ISystemProperties::maxGuestVRAM`

Maximum guest video memory in Megabytes.

6.42.1.5 maxVDISize (read-only)

`unsigned long long ISystemProperties::maxVDISize`

Maximum size of a virtual disk image in Megabytes.

6.42.1.6 networkAdapterCount (read-only)

`unsigned long ISystemProperties::networkAdapterCount`

Number of network adapters associated with every IMachine instance.

6.42.1.7 serialPortCount (read-only)

`unsigned long ISystemProperties::serialPortCount`

Number of serial ports associated with every IMachine instance.

6.42.1.8 parallelPortCount (read-only)

`unsigned long ISystemProperties::parallelPortCount`

Number of parallel ports associated with every IMachine instance.

6.42.1.9 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()

6.42.1.10 defaultVDIFolder (read/write)

`wstring ISystemProperties::defaultVDIFolder`

Full path to the default directory used to create new or open existing virtual disk images when an image file name contains no path.
The initial value of this property is <VirtualBox_home>`/VDI`.

Note: Setting this property to `null` will restore the initial value.

Note: When settings this property, the specified path can be absolute (full path) or relative to the VirtualBox home directory. When reading this property, a full path is always returned.

Note: The specified path may not exist, it will be created when necessary.

See also: IVirtualBox::createHardDisk(), IVirtualBox::openVirtualDiskImage()

6.42.1.11 defaultMachineFolder (read/write)

`wstring ISystemProperties::defaultMachineFolder`

Full path to the default directory used to create new or open existing machines when a settings file name contains no path.

The initial value of this property is <VirtualBox_home>`/Machines`.

> **Note:** Setting this property to `null` will restore the initial value.

> **Note:** When settings this property, the specified path can be absolute (full path) or relative to the VirtualBox home directory. When reading this property, a full path is always returned.

> **Note:** The specified path may not exist, it will be created when necessary.

See also: IVirtualBox::createMachine(), IVirtualBox::openMachine()

6.42.1.12 remoteDisplayAuthLibrary (read/write)

`wstring ISystemProperties::remoteDisplayAuthLibrary`

Library that provides authentication for VRDP 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 `VRDPAuth`. 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` will restore the initial value.

6.42.1.13 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 ISystemProperties::remoteDisplayAuthLibrary, 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 `VRDPAuth`, meaning that the webservice will use the same authentication library that is used by default for VBoxVRDP (again, see ISystemProperties::remoteDisplayAuthLibrary). The format and calling convetion of authentication libraries is the same for the webservice as it is for VBoxVRDP.

6.42.1.14 HWVirtExEnabled (read/write)

`boolean ISystemProperties::HWVirtExEnabled`

This specifies the default value for hardware virtualization extensions. If enabled, virtual machines will make use of hardware virtualization extensions such as Intel VT-x and AMD-V by default. This value can be overridden by each VM using their IMachine::HWVirtExEnabled property.

6.42.1.15 LogHistoryCount (read/write)

`unsigned long ISystemProperties::LogHistoryCount`

This value specifies how many old release log files are kept.

6.43 IUSBController

6.43.1 Attributes

6.43.1.1 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.

6.43.1.2 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.

6.43.1.3 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.

6.43.1.4 deviceFilters (read-only)

```
IUSBDeviceFilterCollection 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

6.43.2 createDeviceFilter

```
IUSBDeviceFilter IUSBController::createDeviceFilter(
            [in] wstring name)
```

Creates a new USB device filter. All attributes except the filter name are set to `null` (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

6.43.3 insertDeviceFilter

```
void IUSBController::insertDeviceFilter(
            [in] unsigned long position,
            [in] IUSBDeviceFilterfilter)
```

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 inster a filter that is already in the collection, will return an error.

See also: #deviceFilters

6.43.4 removeDeviceFilter

```
IUSBDeviceFilter IUSBController::removeDeviceFilter(
            [in] unsigned long position)
```

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

6.44 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.

6.44.1 Attributes

6.44.1.1 id (read-only)

```
uuid IUSBDevice::id
```

Unique USB device ID. This ID is built from #vendorId, #productId, #revision and #serialNumber.

6.44.1.2 vendorId (read-only)

```
unsigned short IUSBDevice::vendorId
```

Vendor ID.

6.44.1.3 productId (read-only)

```
unsigned short IUSBDevice::productId
```

Product ID.

6.44.1.4 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.

6.44.1.5 manufacturer (read-only)

```
wstring IUSBDevice::manufacturer
```

Manufacturer string.

6.44.1.6 product (read-only)

```
wstring IUSBDevice::product
```

Product string.

6.44.1.7 serialNumber (read-only)

```
wstring IUSBDevice::serialNumber
```

Serial number string.

6.44.1.8 address (read-only)

```
wstring IUSBDevice::address
```

Host specific address of the device.

6.44.1.9 port (read-only)

```
unsigned short IUSBDevice::port
```

Host USB port number the device is physically coonected to.

6.44.1.10 version (read-only)

```
unsigned short IUSBDevice::version
```

The major USB version of the device - 1 or 2.

6.44.1.11 portVersion (read-only)

```
unsigned short IUSBDevice::portVersion
```

The major USB version of the host USB port the device is physically coonected to - 1 or 2. For devices not connected to anything this will have the same value as the version attribute.

6.44.1.12 remote (read-only)

```
boolean IUSBDevice::remote
```

Whether the device is physically connected to a remote VRDP client or to a local host machine.

6.45 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 ommitted before a dash (`-`), the minimum possible integer is assumed; if `n` is ommitted after a dash, the maximum possible integer is assummed.

- *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 does'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

6.45.1 Attributes

6.45.1.1 name (read/write)

`wstring IUSBDeviceFilter::name`

Visible name for this filter. This name is used to visually distungish one filter from another, so it can neither be `null` nor an empty string.

6.45.1.2 active (read/write)

`boolean IUSBDeviceFilter::active`

Whether this filter active or has been temporarily disabled.

6.45.1.3 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).

6.45.1.4 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).

6.45.1.5 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 zeroes). 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`.

6.45.1.6 manufacturer (read/write)

`wstring IUSBDeviceFilter::manufacturer`

Manufacturer filter.

6.45.1.7 product (read/write)

`wstring IUSBDeviceFilter::product`

Product filter.

6.45.1.8 serialNumber (read/write)

`wstring IUSBDeviceFilter::serialNumber`

Serial number filter.

6.45.1.9 port (read/write)

`wstring IUSBDeviceFilter::port`

Host USB port filter.

6.45.1.10 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.

6.45.1.11 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 bitmask 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.

6.46 IVHDImage

The IVHDImage interface represents virtual hard disks that use Virtual PC Virtual Machine Disk image files to store hard disk data.

Hard disks using VHD images can be either opened using IVirtualBox::openHardDisk() or created from scratch using IVirtualBox::createHardDisk().

Objects that support this interface also support the IHardDisk interface.

When a new hard disk object is created from scatch, an image file for it is not automatically created. To do it, you need to specify a valid file path, and call createFixedImage() or createDynamicImage(). When it is done, the hard disk object can be registered by calling IVirtualBox::registerHardDisk() and then attached to virtual machines.

The description of the VHD hard disk is stored in the VirtualBox configuration file, so it can be changed (at appropriate times) even when accessible returns `false`. However, the hard disk must not be attached to a running virtual machine.

> **Note:** In the current imlementation, the type of all VHD hard disks is Writethrough and cannot be changed.

6.46.1 Attributes

6.46.1.1 filePath (read/write)

`wstring IVHDImage::filePath`

Full file name of the VHD image of this hard disk. For newly created hard disk objects, this value is `null`.

When assigning a new path, it can be absolute (full path) or relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to the image file.

When reading this propery, a full path is always returned.

Note: This property cannot be changed when created returns `true`. In this case, the specified file name can be absolute (full path) or relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to the image file.

6.46.1.2 created (read-only)

```
boolean IVHDImage::created
```

Whether the virual disk image is created or not. For newly created hard disk objects or after a successful invocation of deleteImage(), this value is `false` until createFixedImage() or createDynamicImage() is called.

6.46.2 createDynamicImage

```
IProgress IVHDImage::createDynamicImage(
            [in] unsigned long long size)
```

Starts creating a dymically expanding hard disk image in the background. The previous image associated with this object, if any, must be deleted using deleteImage, otherwise the operation will fail.

Note: After the returned progress object reports that the operation is complete, this hard disk object can be registered within this VirtualBox installation.

6.46.3 createFixedImage

```
IProgress IVHDImage::createFixedImage(
            [in] unsigned long long size)
```

Starts creating a fixed-size hard disk image in the background. The previous image, if any, must be deleted using deleteImage, otherwise the operation will fail.

Note: After the returned progress object reports that the operation is complete, this hard disk object can be registered within this VirtualBox installation.

6.46.4 deleteImage

```
void IVHDImage::deleteImage()
```

Deletes the existing hard disk image. The hard disk must not be registered within this VirtualBox installation, otherwise the operation will fail.

> **Note:** After this operation succeeds, it will be impossible to register the hard disk until the image file is created again.

> **Note:** This operation is valid only for non-differencing hard disks, after they are unregistered using IVirtualBox::unregisterHardDisk().

6.47 IVMDKImage

The IVMDKImage interface represents a specific type of IHardDisk that uses VMDK image files.

The Virtual Machine Disk (VMDK) format is the industry standard format for virtual hard disk image files, which VirtualBox supports besides its own native VDI format.

Objects that support this interface also support the IHardDisk interface.

Hard disks using VMDK images can be either opened using IVirtualBox::openHardDisk() or created from scratch using IVirtualBox::createHardDisk().

When a new hard disk object is created from scratch, an image file for it is not automatically created. To do it, you need to specify a valid file path, and call createFixedImage() or createDynamicImage(). When it is done, the hard disk object can be registered by calling IVirtualBox::registerHardDisk() and then attached to virtual machines.

The description of the VMDK hard disk is stored in the VirtualBox configuration file, so it can be changed (at appropriate times) even when accessible returns `false`. However, the hard disk must not be attached to a running virtual machine.

> **Note:** In the current imlementation, the type of all VMDK hard disks is Writethrough and cannot be changed.

6.47.1 Attributes

6.47.1.1 filePath (read/write)

```
wstring IVMDKImage::filePath
```

Full file name of the VMDK image of this hard disk. For newly created hard disk objects, this value is `null`.

When assigning a new path, it can be absolute (full path) or relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to the image file.

When reading this propery, a full path is always returned.

> **Note:** This property cannot be changed when created returns `true`.

6.47.1.2 created (read-only)

```
boolean IVMDKImage::created
```

Whether the virual disk image is created or not. For newly created hard disk objects or after a successful invocation of deleteImage(), this value is `false` until createFixedImage() or createDynamicImage() is called.

6.47.2 createDynamicImage

```
IProgress IVMDKImage::createDynamicImage(
            [in] unsigned long long size)
```

Starts creating a dymically expanding hard disk image in the background. The previous image associated with this object, if any, must be deleted using deleteImage, otherwise the operation will fail.

> **Note:** After the returned progress object reports that the operation is complete, this hard disk object can be registered within this VirtualBox installation.

6.47.3 createFixedImage

```
IProgress IVMDKImage::createFixedImage(
            [in] unsigned long long size)
```

Starts creating a fixed-size hard disk image in the background. The previous image, if any, must be deleted using deleteImage, otherwise the operation will fail.

> **Note:** After the returned progress object reports that the operation is complete, this hard disk object can be registered within this VirtualBox installation.

6.47.4 deleteImage

```
void IVMDKImage::deleteImage()
```

Deletes the existing hard disk image. The hard disk must not be registered within this VirtualBox installation, otherwise the operation will fail.

> **Note:** After this operation succeeds, it will be impossible to register the hard disk until the image file is created again.

> **Note:** This operation is valid only for non-differencing hard disks, after they are unregistered using IVirtualBox::unregisterHardDisk().

6.48 IVRDPServer

6.48.1 Attributes

6.48.1.1 enabled (read/write)

```
boolean IVRDPServer::enabled
```

VRDP server status.

6.48.1.2 port (read/write)

```
unsigned long IVRDPServer::port
```

VRDP server port number.

> **Note:** Setting the value of this property to `0` will reset the port number to the default value which is currently `3389`. Reading this property will always return a real port number, even after it has been set to `0` (in which case the default port is returned).

6.48.1.3 netAddress (read/write)

```
wstring IVRDPServer::netAddress
```

VRDP server address.

6.48.1.4 authType (read/write)

```
VRDPAuthType IVRDPServer::authType
```

VRDP authentication method.

6.48.1.5 authTimeout (read/write)

```
unsigned long IVRDPServer::authTimeout
```

Timeout for guest authentication. Milliseconds.

6.48.1.6 allowMultiConnection (read/write)

```
boolean IVRDPServer::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.

6.48.1.7 reuseSingleConnection (read/write)

```
boolean IVRDPServer::reuseSingleConnection
```

Flag whether the existing connection must be dropped and a new connection must be established by the VRDP server, when a new client connects in single connection mode.

6.49 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 IVirtualBox::machines attribute.

6.49.1 Attributes

6.49.1.1 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.

6.49.1.2 revision (read-only)

```
unsigned long IVirtualBox::revision
```

The internal build revision number of the product.

6.49.1.3 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.

6.49.1.4 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).

6.49.1.5 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`.

6.49.1.6 settingsFileVersion (read-only)

```
wstring IVirtualBox::settingsFileVersion
```

Current version of the format of the global VirtualBox settings file (`VirtualBox.xml`).

The version string has the following format:

```
x.y-platform
```

where x and y are the major and the minor format versions, and platform is the platform identifier.

The current version usually matches the value of the settingsFormatVersion attribute unless the settings file was created by an older version of VirtualBox and there was a change of the settings file format since then.

Note that VirtualBox automatically converts settings files from older versions to the most recent version when reading them (usually at VirtualBox startup) but it doesn't save the changes back until you call a method that implicitly saves settings (such as setExtraData()) or call saveSettings() explicitly. Therefore, if the value of this attribute differs from the value of settingsFormatVersion, then it means that the settings file was converted but the result of the conversion is not yet saved to disk.

The above feature may be used by interactive front-ends to inform users about the settings file format change and offer them to explicitly save all converted settings files (the global and VM-specific ones), optionally create bacup copies of the old settings files before saving, etc.

See also: settingsFormatVersion, saveSettingsWithBackup()

6.49.1.7 settingsFormatVersion (read-only)

```
wstring IVirtualBox::settingsFormatVersion
```

Most recent version of the settings file format.

The version string has the following format:

```
x.y-platform
```

where x and y are the major and the minor format versions, and platform is the platform identifier.

VirtualBox uses this version of the format when saving settings files (either as a result of method calls that require to save settings or as a result of an explicit call to saveSettings()).

See also: settingsFileVersion

6.49.1.8 host (read-only)

```
IHost IVirtualBox::host
```

Associated host object.

6.49.1.9 systemProperties (read-only)

```
ISystemProperties IVirtualBox::systemProperties
```

Associated system information object.

6.49.1.10 machines (read-only)

```
IMachineCollection IVirtualBox::machines
```

Collection of machine objects registered within this VirtualBox instance.

6.49.1.11 machines2 (read-only)

```
IMachine IVirtualBox::machines2[]
```

Array of machine objects registered within this VirtualBox instance.

6.49.1.12 hardDisks (read-only)

```
IHardDiskCollection IVirtualBox::hardDisks
```

Collection of hard disk objects registered within this VirtualBox instance.

This collection contains only "top-level" (basic or independent) hard disk images, but not differencing ones. All differencing images of the given top-level image (i.e. all its children) can be enumerated using IHardDisk::children.

6.49.1.13 DVDImages (read-only)

```
IDVDImageCollection IVirtualBox::DVDImages
```

6.49.1.14 FloppyImages (read-only)

```
IFloppyImageCollection IVirtualBox::FloppyImages
```

6.49.1.15 progressOperations (read-only)

```
IProgressCollection IVirtualBox::progressOperations
```

6.49.1.16 guestOSTypes (read-only)

```
IGuestOSTypeCollection IVirtualBox::guestOSTypes
```

6.49.1.17 sharedFolders (read-only)

```
ISharedFolderCollection 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.

6.49.1.18 performanceCollector (read-only)

```
IPerformanceCollector IVirtualBox::performanceCollector
```

Associated performance collector object.

6.49.2 createHardDisk

```
IHardDisk IVirtualBox::createHardDisk(
            [in] HardDiskStorageTypestorageType)
```

Creates a new unregistered hard disk that will use the given storage type.

Most properties of the created hard disk object are uninitialized. Valid values must be assigned to them (and probalby some actions performed) to make the actual usage of this hard disk (register, attach to a virtual machine, etc.). See the description of IHardDisk and descriptions of storage type specific interfaces for more information.

> **Note:** For hard disks using the VirtualDiskImage storage type, an image file is not actually created until you call IVirtualDiskImage::createDynamicImage() or IVirtualDiskImage::createFixedImage().

6.49.3 createLegacyMachine

```
IMachine IVirtualBox::createLegacyMachine(
            [in] wstring settingsFile,
            [in] wstring name,
            [in] uuid id)
```

Creates a new virtual machine in “legacy” mode, using the specified settings file to store machine settings.

As opposed to machines created by createMachine(), the settings file of the machine created in “legacy” mode is not automatically renamed when the machine name is changed – it will always remain the same as specified in this method call.

The specified settings file name can be absolute (full path) or relative to the VirtualBox home directory. If the file name doesn’t contain an extension, the default extension (.xml) will be appended.

Optionally the UUID of the machine can be predefined. If this is not desired (i.e. a new UUID should be generated), pass just an empty or null UUID.

Note that the configuration of the newly created machine is not saved to disk (and therefore no settings file is created) until IMachine::saveSettings() is called. If the specified settings file already exists, IMachine::saveSettings() will return an error.

You should also specify a valid name for the machine. See the IMachine::name property description for more details about the machine name.

The created machine remains unregistered until you call registerMachine().

@deprecated This method may be removed later. It is better to use IVirtualBox::createMachine().

> **Note:** There is no way to change the name of the settings file of the created machine.

6.49.4 createMachine

```
IMachine IVirtualBox::createMachine(
            [in] wstring baseFolder,
            [in] wstring name,
            [in] uuid id)
```

Creates a new virtual machine.

The new machine will have "empty" default settings and will not yet be registered. The typical sequence to create a virtual machine is therefore something like this:

1. Call this method (IVirtualBox::createMachine) to have a new machine created. The machine object returned is "mutable", i.e. automatically locked for the current session, as if openSession had been called on it.
2. Assign meaningful settings to the new machine by calling the respective methods.
3. Call IMachine::saveSettings to have the settings written to the machine's XML settings file. The configuration of the newly created machine will not be saved to disk (and the settings subfolder and file, as described below, will not be created) until this method is called.
4. Call registerMachine to have the machine show up in the list of machines registered with VirtualBox.

Every machine has a *settings file* that is used to store the machine configuration. This file is stored in the directory called *machine settings subfolder*. Unless specified otherwise, both the subfolder and the settings file will have a name that corresponds to the name of the virtual machine. You can specify where to create the machine settings subfolder using the @a baseFolder argument. The base folder can be absolute (full path) or relative to the VirtualBox home directory.

If a null or empty string is given as the base folder (which is recommended), the default machine settings folder will be used as the base folder to create the machine settings subfolder and file. In any case, the full path to the settings file will look like:

```
<base_folder>/<machine_name>/<machine_name>.xml
```

Optionally the UUID of the machine can be predefined. If this is not desired (i.e. a new UUID should be generated), pass just an empty or null UUID.

You should also specify a valid name for the machine. See the IMachine::name property description for more details about the machine name.

The created machine remains unregistered until you call registerMachine().

> **Note:** There is no way to change the name of the settings file or subfolder of the created machine directly.

6.49.5 createSharedFolder

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

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.

6.49.6 findDVDImage

```
IDVDImage IVirtualBox::findDVDImage(
            [in] wstring filePath)
```

Returns a registered CD/DVD image with the given image file.

> **Note:** On host systems with case sensitive filesystems, a case sensitive comparison is performed, otherwise the case of symbols in the file path is ignored.

6.49.7 findFloppyImage

```
IFloppyImage IVirtualBox::findFloppyImage(
            [in] wstring filePath)
```

Returns a registered floppy image with the given image file.

> **Note:** On host systems with case sensitive filesystems, a case sensitive comparison is performed, otherwise the case of symbols in the file path is ignored.

6.49.8 findHardDisk

```
IHardDisk IVirtualBox::findHardDisk(
            [in] wstring location)
```

Returns a registered hard disk that uses the given location to store data. The search is done by comparing the value of the @a location argument to the IHardDisk::location attribute of each registered hard disk.

For locations repesented by file paths (such as VDI and VMDK images), the specified location can be either an absolute file path or a path relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to construct the absolute image file name to search for. Note that on host systems with case sensitive filesystems, a case sensitive comparison is performed, otherwise the case of symbols in the file path is ignored.

6.49.9 findMachine

```
IMachine IVirtualBox::findMachine(
            [in] wstring name)
```

Attempts to find a virtual machine given its name. To look up a machine by UUID, use IVirtualBox::getMachine instead.

6.49.10 findVirtualDiskImage

```
IVirtualDiskImage IVirtualBox::findVirtualDiskImage(
            [in] wstring filePath)
```

Returns a registered hard disk that uses the given image file.
@deprecated Use IVirtualBox::findHardDisk() instead.

> **Note:** The specified file path can be absolute (full path) or relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to the image file.

> **Note:** On host systems with case sensitive filesystems, a case sensitive comparison is performed, otherwise the case of symbols in the file path is ignored.

6.49.11 getDVDImage

```
IDVDImage IVirtualBox::getDVDImage(
            [in] uuid id)
```

Returns a registered CD/DVD image with the given UUID.

6.49.12 getDVDImageUsage

```
wstring IVirtualBox::getDVDImageUsage(
            [in] uuid id,
            [in] ResourceUsageusage)
```

Returns the list of of UUIDs of all virtual machines that use the given CD/DVD image.

6.49.13 getExtraData

```
wstring IVirtualBox::getExtraData(
            [in] wstring key)
```

Returns associated global extra data.

If the requested data @a key does not exist, this function will succeed and return @c NULL in the @a value argument.

6.49.14 getFloppyImage

```
IFloppyImage IVirtualBox::getFloppyImage(
            [in] uuid id)
```

Returns a registered floppy image with the given UUID.

6.49.15 getFloppyImageUsage

```
wstring IVirtualBox::getFloppyImageUsage(
            [in] uuid id,
            [in] ResourceUsageusage)
```

Returns the list of of UUIDs of all virtual machines that use the given floppy image.

6.49.16 getGuestOSType

```
IGuestOSType IVirtualBox::getGuestOSType(
            [in] wstring id)
```

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 IVirtualBox::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.

6.49.17 getHardDisk

```
IHardDisk IVirtualBox::getHardDisk(
            [in] uuid id)
```

Returns the registered hard disk with the given UUID.

6.49.18 getMachine

```
IMachine IVirtualBox::getMachine(
            [in] uuid id)
```

Attempts to find a virtual machine given its UUID. To look up a machine by name, use IVirtualBox::findMachine instead.

6.49.19 getNextExtraDataKey

```
void IVirtualBox::getNextExtraDataKey(
            [in] wstring key,
            [out] wstring nextKey,
            [out] wstring nextValue)
```

Returns the global extra data key name following the supplied key.

An error is returned if the supplied @a key does not exist. @c NULL is returned in @a nextKey if the supplied key is the last key. When supplying @c NULL for the @a key, the first key item is returned in @a nextKey (if there is any). @a nextValue is an optional parameter and if supplied, the next key's value is returned in it.

6.49.20 openDVDImage

```
IDVDImage IVirtualBox::openDVDImage(
            [in] wstring filePath,
            [in] uuid id)
```

Opens the CD/DVD image contained in the specified file of the supported format and assigns it the given UUID. The opened image remains unregistered until registerDVDImage() is called.

6.49.21 openExistingSession

```
void IVirtualBox::openExistingSession(
            [in] ISessionsession,
            [in] uuid machineId)
```

Opens a new remote session with the virtual machine for which a direct session is already open.

The remote session provides some level of control over the VM execution (using the IConsole interface) to the caller; however, within the remote session context, not all VM settings are available for modification.

As opposed to openRemoteSession(), the number of remote sessions opened this way is not limited by the API

> **Note:** It is an error to open a remote session with the machine that doesn't have an open direct session.

See also: openRemoteSession

6.49.22 openFloppyImage

```
IFloppyImage IVirtualBox::openFloppyImage(
            [in] wstring filePath,
            [in] uuid id)
```

Opens a floppy image contained in the specified file of the supported format and assigns it the given UUID. The opened image remains unregistered until registerFloppyImage() is called.

6.49.23 openHardDisk

```
IHardDisk IVirtualBox::openHardDisk(
            [in] wstring location)
```

Opens a hard disk from an existing location.

This method tries to guess the hard disk storage type from the format of the location string and from the contents of the resource the location points to. Currently, a *file path* is the only supported format for the location string which must point to either a VDI file or to a VMDK file. On success, an IHardDisk object will be returned that also implements the corresponding interface (IVirtualDiskImage or IVMDKImage, respectively). The IHardDisk::storageType property may also be used to determine the storage type of the returned object (instead of trying to query one of these interfaces).

> **Note:** The specified file path can be absolute (full path) or relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to the image file.

The opened hard disk remains unregistered until registerHardDisk() is called.

6.49.24 openMachine

```
IMachine IVirtualBox::openMachine(
            [in] wstring settingsFile)
```

Opens a virtual machine from the existing settings file. The opened machine remains unregistered until you call registerMachine().

The specified settings file name can be absolute (full path) or relative to the VirtualBox home directory. This file must exist and must be a valid machine settings file whose contents will be used to construct the machine object.

@deprecated Will be removed soon.

6.49.25 openRemoteSession

```
IProgress IVirtualBox::openRemoteSession(
            [in] ISessionsession,
            [in] uuid machineId,
            [in] wstring type,
            [in] wstring environment)
```

Spawns a new process that executes a virtual machine (called a "remote session").

Opening a remote session causes the VirtualBox server to start a new process that opens a direct session with the given VM. As a result, the VM is locked by that direct session in the new process, preventing conflicting changes from other processes. Since sessions act as locks that such prevent conflicting changes, one cannot open a remote session for a VM that already has another open session (direct or remote), or is currently in the process of opening one (see IMachine::sessionState).

While the remote session still provides some level of control over the VM execution to the caller (using the IConsole interface), not all VM settings are available for modification within the remote session context.

This operation 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 is returned to allow the caller to wait for this asynchronous operation to be completed. Until then, the remote session object remains in the closed state, and accessing the machine or its console through it is invalid. It is recommended to use IProgress::waitForCompletion or similar calls to wait for completion.

As with all ISession objects, it is recommended to call ISession::close on the local session object once openRemoteSession() has been called. However, the session's state (see ISession::state) will not return to "Closed" until the remote session has also closed (i.e. until the VM is no longer running). In that case, however, the state of the session will automatically change back to "Closed".

Currently supported session types (values of the @a type argument) are:

- `gui`: VirtualBox Qt GUI session
- `vrdp`: VirtualBox VRDP Server session

The @a 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 @c null, the server environment is inherited by the started process as is.

See also: openExistingSession

6.49.26 openSession

```
void IVirtualBox::openSession(
            [in] ISessionsession,
            [in] uuid machineId)
```

Opens a new direct session with the given virtual machine.

A direct session acts as a local lock on the given VM. There can be only one direct session open at a time for every virtual machine, protecting the VM from being manipulated by conflicting actions from different processes. Only after a direct session has been opened, one can change all VM settings and execute the VM in the process space of the session object.

Sessions therefore can be compared to mutex semaphores that lock a given VM for modification and execution. See ISession for details.

> **Note:** Unless you are writing a new VM frontend, you will not want to execute a VM in the current process. To spawn a new process that executes a VM, use IVirtualBox::openRemoteSession instead.

Upon successful return, the session object can be used to get access to the machine and to the VM console.

In VirtualBox terminology, the machine becomes "mutable" after a session has been opened. Note that the "mutable" machine object, on which you may invoke IMachine methods to change its settings, will be a different object from the immutable IMachine objects returned by various IVirtualBox methods. To obtain a mutable IMachine object (upon which you can invoke settings methods), use the ISession::machine attribute.

One must always call ISession::close to release the lock on the machine, or the machine's state will eventually be set to "Aborted".

In other words, to change settings on a machine, the following sequence is typically performed:

1. Call this method (openSession) to have a machine locked for the current session.
2. Obtain a mutable IMachine object from ISession::machine.
3. Change the settings of the machine.

4. Call IMachine::saveSettings.
5. Close the session by calling ISession::close.

6.49.27 openVirtualDiskImage

```
IVirtualDiskImage IVirtualBox::openVirtualDiskImage(
            [in] wstring filePath)
```

Opens a hard disk from an existing Virtual Disk Image file. The opened hard disk remains unregistered until registerHardDisk() is called.
@deprecated Use IVirtualBox::openHardDisk() instead.

> **Note:** Opening differencing images is not supported.

> **Note:** The specified file path can be absolute (full path) or relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to the image file.

6.49.28 registerCallback

```
void IVirtualBox::registerCallback(
            [in] IVirtualBoxCallback callback)
```

Registers a new global VirtualBox callback. The methods of the given callback object will be called by VirtualBox when an appropriate event occurs.

6.49.29 registerDVDImage

```
void IVirtualBox::registerDVDImage(
            [in] IDVDImage image)
```

Registers a CD/DVD image within this VirtualBox installation. The image must not be registered and must not be associated with the same image file as any of the already registered images, otherwise the registration will fail.

6.49.30 registerFloppyImage

```
void IVirtualBox::registerFloppyImage(
            [in] IFloppyImage image)
```

Registers a floppy image within this VirtualBox installation. The image must not be registered and must not be associated with the same image file as any of the already registered images, otherwise the registration will fail.

6.49.31 registerHardDisk

```
void IVirtualBox::registerHardDisk(
            [in] IHardDiskhardDisk)
```

Registers the given hard disk within this VirtualBox installation. The hard disk must not be registered, must be IHardDisk::accessible and must not be a differencing hard disk, otherwise the registration will fail.

6.49.32 registerMachine

```
void IVirtualBox::registerMachine(
            [in] IMachinemachine)
```

Registers the machine previously created using createMachine() or opened using openMachine() within this VirtualBox installation. After successful method invocation, the IVirtualBoxCallback::onMachineRegistered signal is sent to all registered callbacks.

> **Note:** This method implicitly calls IMachine::saveSettings to save all current machine settings before registering it.

6.49.33 removeSharedFolder

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

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

6.49.34 saveSettings

```
void IVirtualBox::saveSettings()
```

Saves the global settings to the global settings file (settingsFilePath).

This method is only useful for explicitly saving the global settings file after it has been auto-converted from the old format to the most recent format (see settingsFileVersion for details). Normally, the global settings file is implicitly saved when a global setting is changed.

6.49.35 saveSettingsWithBackup

```
wstring IVirtualBox::saveSettingsWithBackup()
```

Creates a backup copy of the global settings file (settingsFilePath) in case of auto-conversion, and then calls saveSettings().

Note that the backup copy is created **only** if the settings file auto-conversion took place (see settingsFileVersion for details). Otherwise, this call is fully equivalent to saveSettings() and no backup copying is done.

The backup copy is created in the same directory where the original settings file is located. It is given the following file name:

```
original.xml.x.y-platform.bak
```

where `original.xml` is the original settings file name (excluding path), and `x.y-platform` is the version of the old format of the settings file (before auto-conversion).

If the given backup file already exists, this method will try to add the `.N` suffix to the backup file name (where `N` counts from 0 to 9) and copy it again until it succeeds. If all suffixes are occupied, or if any other copy error occurs, this method will return a failure.

If the copy operation succeeds, the @a bakFileName return argument will receive a full path to the created backup file (for informational purposes). Note that this will happen even if the subsequent saveSettings() call performed by this method after the copy operation, fails.

> **Note:** The VirtualBox API never calls this method. It is intended purely for the purposes of creating backup copies of the settings files by front-ends before saving the results of the automatically performed settings conversion to disk.

See also: settingsFileVersion

6.49.36 setExtraData

```
void IVirtualBox::setExtraData(
            [in] wstring key,
            [in] wstring value)
```

Sets associated global extra data.

If you pass @c NULL as a key @a value, the given @a key will be deleted.

> **Note:** Before performing the actual data change, this method will ask all registered callbacks using the IVirtualBoxCallback::onExtraDataCanChange() notification for a permission. If one of the callbacks refuses the new value, the change will not be performed.

Note: On success, the IVirtualBoxCallback::onExtraDataChange() notification is called to inform all registered callbacks about a successful data change.

6.49.37 unregisterCallback

```
void IVirtualBox::unregisterCallback(
            [in] IVirtualBoxCallback callback)
```

Unregisters the previously registered global VirtualBox callback.

6.49.38 unregisterDVDImage

```
IDVDImage IVirtualBox::unregisterDVDImage(
            [in] uuid id)
```

Unregisters the CD/DVD image previously registered using registerDVDImage().

Note: The specified image must not be mounted to any of the existing virtual machines.

6.49.39 unregisterFloppyImage

```
IFloppyImage IVirtualBox::unregisterFloppyImage(
            [in] uuid id)
```

Unregisters the floppy image previously registered using registerFloppyImage().

Note: The specified image must not be mounted to any of the existing virtual machines.

6.49.40 unregisterHardDisk

```
IHardDisk IVirtualBox::unregisterHardDisk(
            [in] uuid id)
```

Unregisters a hard disk previously registered using registerHardDisk().

Note: The specified hard disk must not be attached to any of the existing virtual machines and must not have children (differencing) hard disks.

6.49.41 unregisterMachine

```
IMachine IVirtualBox::unregisterMachine(
            [in] uuid id)
```

Unregisters the machine previously registered using registerMachine. After successful method invocation, the IVirtualBoxCallback::onMachineRegistered signal is sent to all registered callbacks.

> **Note:** The specified machine must not be in the Saved state, have an open (or a spawning) direct session associated with it, have snapshots or have hard disks attached.

> **Note:** This method implicitly calls IMachine::saveSettings to save all current machine settings before unregistering it.

> **Note:** If the given machine is inaccessible (see IMachine::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.

6.49.42 waitForPropertyChange

```
void IVirtualBox::waitForPropertyChange(
            [in] wstring what,
            [in] unsigned long timeout,
            [out] wstring changed,
            [out] wstring values)
```

Blocks the caller until any of the properties represented by the @a what argument changes the value or until the given timeout interval expires.

The @a what argument is a comma separated list of propertiy masks that describe properties the caller is interested in. The property mask is a string in the following format:

```
[[group.]subgroup.]name
```

where @c name is the property name and @c group, @c subgroup are zero or or more property group specifiers. Each element (group or name) in the property mask may be either a latin string or an asterisk symbol (@c "*") which is used to match any

string for the given element. A property mask that doesn't contain asterisk symbols represents a single fully qualified property name.

Groups in the fully qualified property name go from more generic (the left-most part) to more specific (the right-most part). The first element is usually a name of the object the property belongs to. The second element may be either a property name, or a child object name, or an index if the preceeding element names an object which is one of many objects of the same type. This way, property names form a hierarchy of properties. Here are some examples of property names:

`VirtualBox.version`IVirtualBox::version property`Machine.<UUID>.name`IMachine::name property of the machine with the given UUID

Most property names directly correspond to the properties of objects (components) provided by the VirtualBox library and may be used to track changes to these properties. However, there may be pseudo-property names that don't correspond to any existing object's property directly, as well as there may be object properties that don't have a corresponding property name that is understood by this method, and therefore changes to such properties cannot be tracked. See individual object's property descrcriptions to get a fully qualified property name that can be used with this method (if any).

There is a special property mask @c "*" (i.e. a string consisting of a single asterisk symbol) that can be used to match all properties. Below are more examples of property masks:

`VirtualBox.*`Track all properties of the VirtualBox object`Machine.*.name`Track changes to the IMachine::name property of all registered virtual machines

6.50 IVirtualBoxCallback

Note: This interface is not supported in the webservice.

6.50.1 onExtraDataCanChange

```
boolean IVirtualBoxCallback::onExtraDataCanChange(
            [in] uuid machineId,
            [in] wstring key,
            [in] wstring value,
            [out] wstring error)
```

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.

6.50.2 onExtraDataChange

```
void IVirtualBoxCallback::onExtraDataChange(
            [in] uuid machineId,
```

```
            [in] wstring key,
            [in] wstring value)
```

Notification when machine specific or global extra data has changed.

6.50.3 onGuestPropertyChange

```
void IVirtualBoxCallback::onGuestPropertyChange(
            [in] uuid machineId,
            [in] wstring name,
            [in] wstring value,
            [in] wstring flags)
```

Notification when a guest property has changed.

6.50.4 onMachineDataChange

```
void IVirtualBoxCallback::onMachineDataChange(
            [in] uuid machineId)
```

Any of the settings of the given machine has changed.

6.50.5 onMachineRegistered

```
void IVirtualBoxCallback::onMachineRegistered(
            [in] uuid machineId,
            [in] boolean registered)
```

The given machine was registered or unregistered within this VirtualBox installation.

6.50.6 onMachineStateChange

```
void IVirtualBoxCallback::onMachineStateChange(
            [in] uuid machineId,
            [in] MachineStatestate)
```

The execution state of the given machine has changed. See also: IMachine::state

6.50.7 onMediaRegistered

```
void IVirtualBoxCallback::onMediaRegistered(
            [in] uuid mediaId,
            [in] DeviceTypemediaType,
            [in] boolean registered)
```

The given media was registered or unregistered within this VirtualBox installation.
The @a mediaType parameter describes what type of media the specified @a mediaId refers to. Possible values are:

- DeviceType::HardDisk: the media is a hard disk that, if registered, can be obtained using the IVirtualBox::getHardDisk call.
- DeviceType::DVD: the media is a CD/DVD image that, if registered, can be obtained using the IVirtualBox::getDVDImage call.
- DeviceType::Floppy: the media is a Floppy image that, if registered, can be obtained using the IVirtualBox::getFloppyImage call.

Note that if this is a deregistration notification, there is no way to access the object representing the unregistered media. It is supposed that the application will do required cleanup based on the @a mediaId value.

6.50.8 onSessionStateChange

```
void IVirtualBoxCallback::onSessionStateChange(
            [in] uuid machineId,
            [in] SessionStatestate)
```

The state of the session for the given machine was changed. See also: IMachine::sessionState

6.50.9 onSnapshotChange

```
void IVirtualBoxCallback::onSnapshotChange(
            [in] uuid machineId,
            [in] uuid snapshotId)
```

Snapshot properties (name and/or description) have been changed. See also: ISnapshot

6.50.10 onSnapshotDiscarded

```
void IVirtualBoxCallback::onSnapshotDiscarded(
            [in] uuid machineId,
            [in] uuid snapshotId)
```

Snapshot of the given machine has been discarded.

> **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

6.50.11 onSnapshotTaken

```
void IVirtualBoxCallback::onSnapshotTaken(
            [in] uuid machineId,
            [in] uuid snapshotId)
```

A new snapshot of the machine has been taken. See also: ISnapshot

6.51 IVirtualBoxErrorInfo

Note: This interface is not supported in the webservice.

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 retrievefd 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 @a next attribute is the previous error and so on, up to the first error (which is the last in the chain).

6.51.1 Attributes

6.51.1.1 resultCode (read-only)

```
result 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.

Note: In MS COM, there is no equivalent. In XPCOM, it is the same as nsIException::result.

6.51.1.2 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. In XPCOM, there is no equivalent.

6.51.1.3 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.

6.51.1.4 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.

6.51.1.5 next (read-only)

`IVirtualBoxErrorInfo IVirtualBoxErrorInfo::next`

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

Next error object if there is any, or @c null otherwise.

> **Note:** In MS COM, there is no equivalent. In XPCOM, it is the same as nsIException::inner.

6.52 IVirtualDiskImage

The IVirtualDiskImage interface represent a specific type of IHardDisk that uses VDI image files.

The Virtual Disk Image (VDI) format is VirtualBox's native format for hard disk containers.

Objects that support this interface also support the IHardDisk interface.

Hard disks using virtual disk images can be either opened using IVirtualBox::openHardDisk() or created from scratch using IVirtualBox::createHardDisk().

When a new hard disk object is created from scratch, an image file for it is not automatically created. To do it, you need to specify a valid file path, and call createFixedImage() or createDynamicImage(). When it is done, the hard disk object can be registered by calling IVirtualBox::registerHardDisk() and then attached to virtual machines.

The description of the Virtual Disk Image is stored in the image file. For this reason, changing the value of this property requires the hard disk to be accessible. The description of a registered hard disk can be changed only if a virtual machine using it is not running.

6.52.1 Attributes

6.52.1.1 filePath (read/write)

```
wstring IVirtualDiskImage::filePath
```

Full file name of the virtual disk image of this hard disk. For newly created hard disk objects, this value is `null`.

When assigning a new path, it can be absolute (full path) or relative to the VirtualBox home directory. If only a file name without any path is given, the default VDI folder will be used as a path to the image file.

When reading this propery, a full path is always returned.

> **Note:** This property cannot be changed when created returns `true`.

6.52.1.2 created (read-only)

```
boolean IVirtualDiskImage::created
```

Whether the virual disk image is created or not. For newly created hard disk objects or after a successful invocation of deleteImage(), this value is `false` until createFixedImage() or createDynamicImage() is called.

6.52.2 createDynamicImage

```
IProgress IVirtualDiskImage::createDynamicImage(
            [in] unsigned long long size)
```

Starts creating a dymically expanding hard disk image in the background. The previous image associated with this object, if any, must be deleted using deleteImage, otherwise the operation will fail.

> **Note:** After the returned progress object reports that the operation is complete, this hard disk object can be registered within this VirtualBox installation.

6.52.3 createFixedImage

```
IProgress IVirtualDiskImage::createFixedImage(
            [in] unsigned long long size)
```

Starts creating a fixed-size hard disk image in the background. The previous image, if any, must be deleted using deleteImage, otherwise the operation will fail.

> **Note:** After the returned progress object reports that the operation is complete, this hard disk object can be registered within this VirtualBox installation.

6.52.4 deleteImage

```
void IVirtualDiskImage::deleteImage()
```

Deletes the existing hard disk image. The hard disk must not be registered within this VirtualBox installation, otherwise the operation will fail.

> **Note:** After this operation succeeds, it will be impossible to register the hard disk until the image file is created again.

> **Note:** This operation is valid only for non-differencing hard disks, after they are unregistered using IVirtualBox::unregisterHardDisk().

6.53 IWebsessionManager

Note: This interface is supported in the webservice only, not in COM/XPCOM.

Websession manager. This provides essential services to webservice clients.

6.53.1 getSessionObject

```
ISession IWebsessionManager::getSessionObject(
            [in] IVirtualBoxrefIVirtualBox)
```

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

6.53.2 logoff

```
void IWebsessionManager::logoff(
            [in] IVirtualBoxrefIVirtualBox)
```

Logs off the client who has previously logged on with IWebsessionManager::logoff and destroys all resources associated with the session (most importantly, all managed objects created in the server while the session was active).

6.53.3 logon

```
IVirtualBox IWebsessionManager::logon(
            [in] wstring username,
            [in] wstring 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.

7 Enumerations (enums)

7.1 AudioControllerType

Virtual audio controller type.

AC97

SB16

7.2 AudioDriverType

Host audio driver type.

Null `null` value. Also means “dummy audio driver”.

WinMM

OSS

ALSA

DirectSound

CoreAudio

MMPM

Pulse

SolAudio

7.3 BIOSBootMenuMode

BIOS boot menu mode.

Disabled

MenuOnly

MessageAndMenu

7.4 ClipboardMode

Host-Guest clipboard interchange mode.

Disabled

HostToGuest

GuestToHost

Bidirectional

7.5 DeviceActivity

Device activity for IConsole::getDeviceActivity.

Null

Idle

Reading

Writing

7.6 DeviceType

Device type.

Null `null` value which may also mean “no device”.

> **Note:** This value is 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.

7.7 DriveState

Null `null` value. Never used by the API.

NotMounted

ImageMounted

HostDriveCaptured

7.8 FramebufferAccelerationOperation

Framebuffer acceleration operation.

SolidFillAcceleration

ScreenCopyAcceleration

7.9 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 informantion 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.

7.10 GuestStatisticType

Statistics type for IGuest::getStatistic.

CPULoad_Idle Idle CPU load (0-100%) for last interval.

CPULoad_Kernel Kernel CPU load (0-100%) for last interval.

CPULoad_User User CPU load (0-100%) for last interval.

Threads Total number of threads in the system.

Processes Total number of processes in the system.

Handles Total number of handles in the system.

MemoryLoad Memory load (0-100%).

PhysMemTotal Total physical memory in megabytes.

PhysMemAvailable Free physical memory in megabytes.

PhysMemBalloon Ballooned physical memory in megabytes.

MemCommitTotal Total amount of memory in the committed state in megabytes.

MemKernelTotal Total amount of memory used by the guest OS's kernel in megabytes.

MemKernelPaged Total amount of paged memory used by the guest OS's kernel in megabytes.

MemKernelNonpaged Total amount of nonpaged memory used by the guest OS's kernel in megabytes.

MemSystemCache Total amount of memory used by the guest OS's system cache in megabytes.

PageFileSize Pagefile size in megabytes.

SampleNumber Statistics sample number

MaxVal

7.11 HardDiskStorageType

Virtual hard disk storage type. See also: IHardDisk

VirtualDiskImage Virtual Disk Image, VDI (a regular file in the file system of the host OS, see IVirtualDiskImage)

ISCSIHardDisk iSCSI Remote Disk (a disk accessed via the Internet SCSI protocol over a TCP/IP network, see IISCSIHardDisk)

VMDKImage VMware Virtual Machine Disk image (a regular file in the file system of the host OS, see IVMDKImage)

CustomHardDisk Disk formats supported through plugins (see ICustomHardDisk)

VHDImage Virtual PC Virtual Machine Disk image (a regular file in the file system of the host OS, see IVHDImage)

7.12 HardDiskType

Virtual hard disk type. See also: IHardDisk

Normal Normal hard disk (attached directly or indirectly, preserved when taking snapshots).

Immutable Immutable hard disk (attached indirectly, changes are wiped out after powering off the virtual machine).

Writethrough Write through hard disk (attached directly, ignored when taking snapshots).

7.13 IDEControllerType

IDE controller type.

Null `null` value. Never used by the API.

PIIX3

PIIX4

7.14 MachineState

Virtual machine execution state. This enumeration represents possible values of the IMachine::state attribute.

Null `null` value. Never used by the API.

PoweredOff The machine is not running.

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.

> **Note:** No any machine settings can be altered when the machine is in this state.

Aborted A process that run the machine has abnormally terminated. Other than that, this value is equivalent to #PoweredOff.

Running The machine is currently being executed.

> **Note:** This value can be used in comparison expressions: all state values below it describe a virtual machine that is not currently being executed (i.e., it is completely out of action).

Paused The execution of the machine has been paused.

> **Note:** This value can be used in comparison expressions: all state values above it represent unstable states of the running virtual machine. Unless explicitly stated otherwise, no machine settings can be altered when it is in one of the unstable states.

Stuck The execution of the machine has reached the "Guru Meditation" condition. This condition indicates an internal VMM 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).

Starting The machine is being started after powering it on from a zero execution state.

Stopping The machine is being normally stopped (after explicitly powering it off, or after the guest OS has initiated a shutdown sequence).

Saving The machine is saving its execution state to a file as a result of calling IConsole::saveState or an online snapshot of the machine is being taken using IConsole::takeSnapshot.

Restoring The execution state of the machine is being restored from a file after powering it on from a saved execution state.

Discarding A snapshot of the machine is being discarded after calling IConsole::discardSnapshot or its current state is being discarded after IConsole::discardCurrentState.

7.15 MouseButtonState

Mouse button state.

LeftButton

RightButton

MiddleButton

WheelUp

WheelDown

MouseStateMask

7.16 NetworkAdapterType

Network adapter type.

Null `null` value. Never used by the API.

Am79C970A

Am79C973

I82540EM

I82543GC

7.17 NetworkAttachmentType

Network attachment type.

Null `null` value. Also means “not attached”.

NAT

HostInterface

Internal

7.18 PortMode

The PortMode enumeration represents possible communicaton 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.

7.19 ResourceUsage

Usage type constants for IVirtualBox::getDVDImageUsage and IVirtualBox::getFloppyImageUsage.

Null `null` value. Never used by the API.

Permanent Scopes the VMs that use the resource permanently (the information about this usage is stored in the VM settings file).

Temporary Scopes the VMs that are temporarily using the resource (the information about the usage is not yet saved in the VM settings file). Temporary usage can take place only in the context of an open session.

All Combines Permanent and Temporary.

7.20 Scope

Scope of the operation.

A generic enumeration used in various methods to define the action or argument scope.

Global

Machine

Session

7.21 SessionState

Session state. This enumeration represents possible values of IMachine::sessionState and ISession::state attributes. Idividual value descriptions contain the appropriate meaning for every case.

Null `null` value. Never used by the API.

Closed The machine has no open sessions (IMachine::sessionState); the session is closed (ISession::state)

Open The machine has an open direct session (IMachine::sessionState); the session is open (ISession::state)

Spawning A new (direct) session is being opened for the machine as a result of IVirtualBox::openRemoteSession() call (IMachine::sessionState); the session is currently being opened as a result of IVirtualBox::openRemoteSession() call (ISession::state)

Closing The direct session is being closed (IMachine::sessionState); the session is being closed (ISession::state)

7.22 SessionType

Session type. This enumeration represents possible values of the ISession::type attribute.

Null `null` value. Never used by the API.

Direct Direct session (opened by IVirtualBox::openSession())

Remote Remote session (opened by IVirtualBox::openRemoteSession())

Existing Existing session (opened by IVirtualBox::openExistingSession())

7.23 StorageBus

Interface bus type for storage devices.

Null `null` value. Never used by the API.

IDE

SATA

7.24 TSBool

Boolean variable having a third state, default.

False

True

Default

7.25 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.

7.26 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 ot to #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 #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 #Unavailable, #Busy or #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 #Busy, #Available or #Held.

> **Note:** Due to differences in USB stack implementations in Linux and Win32, states #Busy and #Available 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 #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.

7.27 VRDPAuthType

VRDP authentication type.

Null `null` value. Also means "no authentication".

External

Guest

8 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.

8.1 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.

8.2 Protocol Specification

The HGCM protocol definitions are contained in the `VBox/VBoxGuest.h`

8.2.1 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 (hexademical 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.

8.2.2 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 the HGCM how to look for the requested service:

Name (hexademical 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 hardcoded 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.

8.2.3 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).

8.2.4 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
VMMDevHGCM-ParmType_32bit (1)	A 32 bit value.
VMMDevHGCM-ParmType_64bit (2)	A 64 bit value.
VMMDevHGCM-Parm-Type_PhysAddr (3)	A 32 bit size followed by a 32 bit or 64 bit guest physical address.
VMMDevHGCM-ParmType_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.
VMMDevHGCM-Parm-Type_LinAddr_In (5)	Same as VMMDevHGCMParmType_LinAddr but the buffer is used only for host to guest data.
VMMDevHGCM-Parm-Type_LinAddr_Out (6)	Same as VMMDevHGCMParmType_LinAddr but the buffer is used only for guest to host data.
VMMDevHGCM-Parm-Type_LinAddr_Locked (7)	Same as VMMDevHGCMParmType_LinAddr but the buffer is already locked by the guest.
VMMDevHGCM-Parm-Type_LinAddr_Locked_In (1)	Same as VMMDevHGCMParmType_LinAddr_In but the buffer is already locked by the guest.
VMMDevHGCM-Parm-Type_LinAddr_Locked_Out (1)	Same as VMMDevHGCMParmType_LinAddr_Out but the buffer is already locked by the guest.

The

8.2.5 Cancel

This request cancels a call request (VMMDevHGCMCancel):

Name	Description
header	The generic HGCM request header with type equal to VMMDevReq_HGCMCancel (64).

8.3 Guest Software Interface

The guest HGCM clients can call HGCM services from both drivers and applications.

8.3.1 The Guest Driver Interface

The driver interface is implemented in the VirtualBox guest additions driver (VBoxGuest), which works with the 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 (VBOX_SUCCESS (rc))
    {
        rc = data.result;
    }

    if (VBOX_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. */

    /* Initalize 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 (VBOX_SUCCESS (rc))
{
    rc = data.callInfo.result;
    *pcbBuffer = data.cb.u.value32; /* This is returned by the HGCM service. */
}
```

8.3.2 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.

8.4 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.