pyvbox Documentation

Release 1.2.0 vbox 5.1.1

Michael Dorman

Jan 14, 2018

Contents

CHAPTER 1

Introduction

What's in pyvbox:

- A complete implementation of the VirtualBox Main API
- Create a VirtualBox instance and seamlessly explore the potential of VirtualBox's amazing Main API
- Pythonic functions and names.
- Introspection, documentation strings, getters and setters, and more...

Project documentation at pythonhosted.org.

Project hosting provided by github.com.

[mjdorma+pyvbox@gmail.com]

1.1 Install

Simply run the following:

```
> python setup.py install
```

or PyPi:

```
> pip install pyvbox
```

1.2 Getting started

Exploring the library:

```
> ipython
In [1]: import virtualbox

In [2]: virtualbox?

In [3]: virtualbox.VirtualBox?

In [4]: virtualbox.library.IMachine?

In [5]: virtualbox.library.MachineState?

In [6]: virtualbox.library.MachineState.teleported?
```

Listing machines:

```
> ipython
In [1]: import virtualbox

In [2]: vbox = virtualbox.VirtualBox()

In [3]: print("VM(s):\n + %s" % "\n + ".join([vm.name for vm in vbox.machines]))
VM(s):
 + filestore
 + xpsp3
 + win7
 + win8
 + test_vm
```

Launch machine, take a screen shot, stop machine:

```
> ipython
In [1]: import virtualbox

In [2]: vbox = virtualbox.VirtualBox()

In [3]: session = virtualbox.Session()

In [4]: vm = vbox.find_machine('test_vm')

In [5]: progress = vm.launch_vm_process(session, 'gui', '')

In [6]: h, w, _, _, _, _ = session.console.display.get_screen_resolution(0)

In [7]: png = session.console.display.take_screen_shot_to_array(0, h, w, virtualbox.
↪library.BitmapFormat.png)

In [8]: with open('screenshot.png', 'wb') as f:
  ....:     f.write(png)

In [9]: print(session.state)
Locked

In [10]: session.state
Out[10]: SessionState(2)

In [11]: session.state >= 2
Out[11]: True
```

```
In [12]: session.console.power_down()
```

Write text into a window on a running machine:

```
> ipython
In [1]: import virtualbox

In [2]: vbox = virtualbox.VirtualBox()

In [3]: vm = vbox.find_machine('test_vm')

In [4]: session = vm.create_session()

In [5]: session.console.keyboard.put_keys("Q: 'You want control?'\nA: 'Yes, but just␣
↪a tad...'")
```

Execute a command in the guest:

```
> ipython
In [1]: import virtualbox

In [2]: vbox = virtualbox.VirtualBox()

In [3]: vm = vbox.find_machine('test_vm')

In [4]: session = vm.create_session()

In [5]: gs = session.console.guest.create_session('Michael Dorman', 'password')

In [6]: process, stdout, stderr = gs.execute('C:\\Windows\\System32\\cmd.exe', ['/C',
↪'tasklist'])

In [7]: print stdout

Image Name                     PID Session Name        Session#    Mem Usage
========================= ====== ================ ======== ============
System Idle Process              0 Console                    0         28 K
System                           4 Console                    0        236 K
smss.exe                       532 Console                    0        432 K
csrss.exe                      596 Console                    0      3,440 K
winlogon.exe                   620 Console                    0      2,380 K
services.exe                   664 Console                    0      3,780 K
lsass.exe                      676 Console                    0      6,276 K
VBoxService.exe                856 Console                    0      3,972 K
svchost.exe                    900 Console                    0      4,908 K
svchost.exe                   1016 Console                    0      4,264 K
svchost.exe                   1144 Console                    0     18,344 K
svchost.exe                   1268 Console                    0      2,992 K
svchost.exe                   1372 Console                    0      3,948 K
spoolsv.exe                   1468 Console                    0      4,712 K
svchost.exe                   2000 Console                    0      3,856 K
wuauclt.exe                    400 Console                    0      7,176 K
alg.exe                       1092 Console                    0      3,656 K
wscntfy.exe                   1532 Console                    0      2,396 K
explorer.exe                  1728 Console                    0     14,796 K
wmiprvse.exe                  1832 Console                    0      7,096 K
VBoxTray.exe                  1940 Console                    0      3,196 K
ctfmon.exe                    1948 Console                    0      3,292 K
```

```
cmd.exe                      1284 Console                    0      2,576 K
tasklist.exe                  124 Console                    0      4,584 K
```

Using context to manage opened sessions and locks:

```
> ipython
In [1]: import virtualbox

In [2]: vbox = virtualbox.VirtualBox()

In [3]: vm = vbox.find_machine('test_vm')

In [4]: with vm.create_session() as session:
   ...:     with session.console.guest.create_session('Michael Dorman', 'password')␣
↪as gs:
   ...:         print(gs.directory_exists("C:\\Windows"))
   ...:
True
```

On an already running VM, register to receive on guest keyboard events:

```
>ipython
In [1]: from virtualbox import library

In [2]: import virtualbox

In [3]: vbox = virtualbox.VirtualBox()

In [4]: vm = vbox.find_machine('test_vm')

In [5]: s = vm.create_session()

In [6]: def test(a):
   ...:     print(a.scancodes)
   ...:

In [7]: s.console.keyboard.set_on_guest_keyboard(test)
Out[7]: 140448201250560

In [8]: [35]
[23]
[163]
[151]
[57]
[185]
[35]
[24]
[163]
[152]
```

See gist for more pyvbox examples.

1.3 Issues

Source code for *pyvbox* is hosted on GitHub. Please file bug reports with GitHub's issues system.

1.4 Compatibility

pyvbox utilises the VirtualBox project's vboxapi to gain access to the underlying COM API primitives. Therefore, pyvbox is compatible on systems which have a running vboxapi.

CHAPTER 2

Changelog

version 1.2.0 (28/08/2017)

- Searches for vboxapi installed in Anaconda on Windows. (@SethMichaelLarson PR #80)
- Added `__lt__` and `__gt__` methods for orderability on Python 3. (@SethMichaelLarson PR #82)

version 1.1.0 (02/06/2017)

- IGuest.create_session() now raises a more descriptive error if not able to connect with a zero-length password. (@SethMichaelLarson PR #70)
- Add sys.executable-derived paths in list to check for vboxapi (@SethMichaelLarson PR #69)
- Fix IGuestProcess.execute() on Python 3.x (@SethMichaelLarson PR #58)
- Fix errors to not output on Windows platforms. (@SethMichaelLarson PR #57)
- Fix error caused by attempting to set any attribute in the COM interface using setattr raising an error. (Reported by @josepegerent, patch by @SethMichaelLarson PR #74)

version 1.0.0 (18/01/2017)

- Support for 5.0.x VirtualBox.
- Introduce Major.Minor virtualbox build version assertion when creating a VirtualBox instance.
- Fix to IMachine.export_to (contribution from @z00m1n).

version 0.2.2 (05/08/2015)

- Cleanup managers at exit (reported by @jiml521).
- Add three time check for attribute in xpcom interface object before failing (reported by @shohamp).
- Update library.py to 4.3.28/src/VBox/Main/idl/VirtualBox.xidl

version 0.2.0

- This change introduces some significant (potential compatability breaking) updates from the latest VirtualBox.xidl.
- Bug fixes in IMachine (reported by @danikdanik).

- IHost API issue workaround by @wndhydrnt.

version 0.1.6 (01/08/2014)

- Bug fixes (compatability issue with py26 and virtual keyboard).
- Thanks to contributions by @D4rkC4t and @Guilherme Moro.

version 0.1.5 (11/05/2014)

- Improve error handling and documentation of error types.
- Appliance extension.
- Update to latest API (includes Paravirt provider).
- Thanks to contributions by @nilp0inter

version 0.1.4 (09/04/2014)

- Fixed bug in error class container.

version 0.1.3 (04/03/2014)

- Bug fix for API support.
- Added markup generation to library documentation.
- Improved Manager bootstrap design.
- Py3 compatibility (although vboxapi does not support py3).

version 0.1.2 (28/02/2014)

- Bug fix for virtualenv support
- Keyboard scancode decoder (Note: coded in the delivery suite on the day of the birth of my baby girl Sophia.)
- Refactored documentation

version 0.1.1 (17/02/2014)

- Minor improvements
- Additional extensions
- virtualenv support

version 0.1 (05/01/2014)

- As per roadmap v0.1
- type checking baseinteger
- update to latests Xidl

version 0.0.7 (09/10/2013)

- machine pool

version 0.0.6 (25/07/2013)

- now with event support

version 0.0.5 (23/07/2013)

- moved manage into library_ext Interfaces
- made library.py compatible with differences found between xpcom and COM (Linux Vs Windows)

version 0.0.4 (27/06/2013)

- added execute, context, and keyboard

version 0.0.3 (30/05/2012)

- added manage

version 0.0.2 (28/05/2013)

- library ext module

version 0.0.1 (27/05/2013)

- packaged

version 0.0.0 (20/05/2013)

- builder
- library primitives

CHAPTER 3

Library Reference

3.1 virtualbox – main module

This module is the root module for the pyvbox project. The name 'virtualbox' has been chosen to enable explicit naming when using this package. The author suggests that people new to VirtualBox's extensive COM interface should take a moment to delve into the API's documentation which will assist in understanding how VirtualBox's client server module functions.

3.1.1 Code reference

virtualbox.**import_vboxapi**(**args*, ***kwds*)

This import is designed to help when loading vboxapi inside of alternative Python environments (virtualenvs etc).

Return type vboxapi module

class virtualbox.**VirtualBox**([*interface*, *manager*])

The VirthalBox class is the primary interface used to interact with a VirtualBox server. It wraps the IVirtualBox interface which "represents the main interface exposed by the product that provides virtual machine management."

Optionally, this class can be initialised with an already connected COM IVirtualBox interface or by passing in a Manager object which implements a *virtualbox.Manager* get_virthalbox method.

class virtualbox.**Manager**(*mtype=None*, *mparams=None*)

The Manager maintains a single point of entry into vboxapi.

This object is responsible for the construction of *virtualbox.library_ext.ISession* and *virtualbox.library_ext.IVirtualBox*.

Parameters

- **mtype** (*str (Default None)*) – Type of manager i.e. WEBSERVICE.

- **mparams** (*tuple|list (Default None)*) – The params that the mtype manager object accepts.

manager
Create a default Manager object

Builds a singleton VirtualBoxManager object.

Note: It is not necessary to build this object when defining a Session or VirtualBox object as both of these classes will default to this object's global singleton during construction.

get_virtualbox()
Return a VirtualBox interface

Return type *library.IVirtualBox*

get_session()
Return a Session interface

Return type *library.ISession*

cast_object(*interface_object*, *interface_class*)
Cast the obj to the interface class

Return type interface_class(interface_object)

bin_path
return the virtualbox install directory

Return type str

__weakref__
list of weak references to the object (if defined)

class virtualbox.**WebServiceManager**(*url='http://localhost/'*, *user=''*, *password=''*)
The WebServiceManager extends the base Manager to include the ability to build a WEBSERVICE type vboxapi interface.

3.2 virtualbox.pool – machine pool management

3.2.1 Virtual Machine pool

The *MachinePool* manages a pool of linked clones against a defined "root machine". This module works with multiple processes running on the host machine at a time. It manages a resource lock over the root virtual machine to ensure consistency.

In this example the machine *win7* has a current version of guest editions installed and is in a powered off state.

Create multiple clones:

```
pool = MachinePool('win7')
sessions = []
for i in range(3):
    sessions.append(pool.acquire("Mick", "password"))

# You now have three running machines.
for session in sessions:
    with session.guest.create_session("Mick", "password") as gs:
        _, out, _ = gs.execute("ipconfig")
        print(out)
```

```
for session in sessions:
    pool.release(session)
```

A reliable version of the above code would look like this:

```
pool = MachinePool('win7')
sessions = []
try:
    for i in range(3):
        sessions.append(pool.acquire("Mick", "password"))

    # You now have three running machines.
    for session in sessions:
        with session.guest.create_session("Mick", "password") as gs:
            _, out, _ = gs.execute("ipconfig")
            print(out)

finally:
    for session in sessions:
        try:
            pool.release(session)
        except Exception as err:
            print("Error raised on release: %s" % err)
```

3.2.2 Code reference

class virtualbox.pool.**MachinePool**(*machine_name*)

MachinePool manages a pool of resources and enable cross process coordination of a linked machine clone.

acquire(*username*, *password*, *frontend='headless'*)

Acquire a Machine resource.

release(*session*)

Release a machine session resource.

3.3 virtualbox.library_ext – extensions to *virtualbox.library*

The *virtualbox.library_ext* is a container package that makes it simple to extend and replace the classes that have been automatically generated in *virtualbox.library*.

This simplifies the builder code significantly by not having to handle specific edge cases where bugs have been identified in the VirtualBox.xidl file. It also makes it simple to redefine default behaviour, or simply add various sugar to functions in an interface (such as defining defaults for function parameters).

3.3.1 Code reference

The documentation captured in this reference reflects the extensions or fixes applied to the default library.py.

class virtualbox.library_ext.**IVirtualBox**([*interface*, *manager*])

The VirtualBox interface object is the primary interface into VirtualBox's COM API. The default constructor can take a *library.Interface* object or a *virtualbox.Manager* object.

register_on_machine_state_changed(*callback*)
The *callback* function is called with a *IMachineStateChangedEvent* argument on a machine state changed event.

```
def callback(event):
    print("Machine %s state changed to %s" % (event.machine_id,
                                              event.state))

vbox = virtualbox.VirtualBox()
vbox.register_on_machine_state_changed(callback)
```

register_on_machine_data_changed(*callback*)
The *callback* function is called with a *IMachineDataChangedEvent* argument on a machine state changed event.

```
def callback(event):
    print("Settings data changed for %s" % event.machine_id)

vbox = virtualbox.VirtualBox()
vbox.register_on_machine_data_changed(callback)
```

register_on_machine_registered(*callback*)

The *callback* function is called with a *IMachineRegisteredEvent* argument on a machine registered event.

```
def callback(event):
    if event.registered:
        action = 'registered'
    else:
        action = 'unregistered'
    print("%s was %s" % (event.machine_id, action))

vbox = virtualbox.VirtualBox()
vbox.register_on_machine_registered(callback)
```

register_on_snapshot_deleted(*callback*)
The *callback* function is called with a *ISnapshotDeletedEvent* argument on a snapshot deleted event.

```
def callback(event):
    print(event.snapshot_id)

vbox = virtualbox.VirtualBox()
vbox.register_on_snapshot_deleted(callback)
```

register_on_snapshot_taken(*callback*)
The *callback* function is called with a *ISnapshotTakenEvent* argument on a snapshot taken event.

```
def callback(event):
    print(event.snapshot_id)

vbox = virtualbox.VirtualBox()
vbox.register_on_snapshot_taken(callback)
```

register_on_snapshot_changed(*callback*)
The *callback* function is called with a *ISnapshotChangedEvent* argument on a snapshot changed event.

```
def callback(event):
    print(event.snapshot_id)

vbox = virtualbox.VirtualBox()
vbox.register_on_snapshot_changed(callback)
```

register_on_guest_property_changed(*callback*)
The *callback* function is called with a *IGuestPropertyChangedEvent* argument on a guest property changed event.

```
def callback(event):
    print("%s %s %s" % (event.name, event.value, event.flags))

vbox = virtualbox.VirtualBox()
vbox.register_on_guest_property_changed(callback)
```

register_on_session_state_changed(*callback*)
The *callback* function is called with a *ISessionStateChangedEvent* argument on a session state changed event.

```
def callback(event):
    print("Session on machine %s is %s" % (event.machine_id,
                                           event.state))

vbox = virtualbox.VirtualBox()
vbox.register_on_session_state_changed(callback)
```

register_on_event_source_changed(*callback*)
The *callback* function is called with a *IEventSourceChangedEvent* on a event source changed event. This occurs when a listener is added or removed.

```
def callback(event):
    if event.add:
        action = 'added'
    else:
        action = 'removed'
    print("A listener was %s from vbox's event_source %s" % \
            action)

vbox.register_on_event_source_changed(callback)
```

register_on_extra_data_changed(*callback*)
The *callback* function is called with a *IExtraDataChangedEvent* argument on a extra data changed event.

```
def callback(event):
    print("%s %s=%s" % (event.machine_id, event.key, event.value))

vbox = virtualbox.VirtualBox()
vbox.register_on_extra_data_changed(callback)
```

register_on_extra_data_can_change(*callback*)
The *callback* function is called with a *IExtraDataCanChangeEvent* argument on a extra data can change event.

```
def callback(event):
    if event.key == 'blah':
        print("Veto served")
```

```
        event.add_veto("blah is mine...")
    else:
        print("Allow %s %s" % (event.key, event.value))

vbox = virtualbox.VirtualBox()
vbox.register_on_extra_data_can_change(callback)
```

To see this work simply run the following vboxmanage command:

```
vboxmanage setextradata global blah winner
```

class virtualbox.library_ext.**ISession**

Just like the *IVirtualBox* interface the *ISession* can be bootstrapped from a *virtualbox.Manager* object. This is special in that it represents a client process and allows for locking virtual machines.

To reduce complexity over management of an *ISession* lock, the base class has been extended to implement the *context management protocol*.

Using an ISession object:

```
vbox = virtualbox.VirtualBox()
vm = vbox.find_machine('test_vm')
with vm.create_session() as session:
    #do stuff with the session
```

class virtualbox.library_ext.**IGuest**

create_session(*user*, *password*[, *domain*, *session_name*, *timeout_ms*])

This method extends the default *IGuest.create_session* method by adding a polling block operation that waits for the guest session to be ready. It also defaults the values of *domain* to '' and *session_name* to 'pyvbox'.

If *timeout_ms* is not equal to 0, this method block until the session is ready and active for querying the Guest operating system. This test is performed by polling for the existence of *C:autoexec.bat* or */bin/sh*. If the timeout is exceeded a VBoxError will be raised.

Returns a IGuestSession object on completion.

update_guest_addtions([*source*, *arguments*, *flags*])

BUG FIX: This method fixes the bug in the definition for the *updateGuestAdditions* method. In the API definition this function is defined to take a list of *arguments* but the implementation only takes *source* and *flags*.

As an extension to this method, *source* is now an optional arguemnt. If the *source* path for the update ISO is not provided, this method will attempt to find a copy of the VBoxGuestAdditions.iso file from the VirtualBox install path.

Returns an IProgress object

class virtualbox.library_ext.**IGuestSession**

When an IGuestSession is created, it requires that the session is explicitly closed after its use. This is done by calling the *IGuestSession.close* method. To simply this behaviour, the default class has been extended to implement the *context management protocol*.

Using an IGuestSession ojbect:

```
guest = session.console.guest
with guest.create_session('user', 'password') as guest_session:
    #do stuff with the guest session
```

execute (*command*[, *arguments*, *stdin*, *environment*, *flags*, *priority*, *affinity*, *timeout_ms*])
Execute a command in the guest

class virtualbox.library_ext.**IEventSource**

register_callback (*callback*, *event_type*)
provide a helper function that wraps the *events.register_callback* method. *callback* is the function to be called back when this *IEventSource* raises *event_type*.

class virtualbox.library_ext.**IKeyboard**

put_keys ([*press_keys*, *hold_keys*, *press_delay*])
Press the keys listed by the *press_keys* list into the *IKeyboard* whilst holding down the *hold_keys*. Control the press speed by defining the *press_delay* which is the number of milliseconds between each press.

For a full list of defined keys, refer to:

```
virtualbox.library.IKeyboard.SCANCODES.keys()
```

register_on_guest_keyboard (*callback*)
The *callback* function is called with a *IGuestKeyboardEvent* argument when a guest keyboard event occurs.

```
def callback(event):
    print(event.scancodes)

session.console.keyboard.register_on_guest_keyboard(callback)
```

class virtualbox.library_ext.**IMouse**

register_on_guest_mouse (*callback*)
The *callback* function is called with a *IGuestMouseEvent* argument when mouse event occurs.

```
def callback(event):
    print(("%s %s %s" % (event.x, event.y, event.z))

session.console.mouse.set_guest_mouse(callback)
```

class virtualbox.library_ext.**IProgress**

__str__ ()
Returns a progress string in a human readable format.

class virtualbox.library_ext.**IMachine**

remove ([*delete*])
Unregister and delete this *Machine*. If *delete* is set to False, the machine will only be detached and unregistered from the VBoxSvr.

clone ([*snapshot_name_or_id*, *mode*, *options*, *name*, *uuid*, *groups*, *basefolder*, *register*])
Clone this *Machine*. The options for this method have been setup to default create a linked clone. Depending on the mode and the options VirtualBox will require the *Machine* to have different state.

To clone from a snapshot, the *snapshot_name_or_id* value needs to be defined. This value can be either an ISnapshot object or a unicode or str value for the name or the id of a snapshot.

If *name* is not defined, the chosen name will be the name of this *Machine* concatenated with " Clone". When deciding a final name, this method will check if the name already exists. If it exists, it will automatically append " (N)" to the end of the name string where N is the number that did not exist.

To understand the complexities behind the options of this method, please read through the documentation for the *library.IVirtualBox.create_machine* and *library.IMachine.clone_to* methods.

delete_config(*media*)
BUG FIX: This method fixes a bug in the interface definition for the default method name 'deleteConfig'. As it turns out, the actual name implemented is 'delete'.

create_session([*lock_type*, *session*])
A helper function to simplify the creation of a *ISession* lock over this *Machine*. *lock_type* defaults to *library.LockType.shared*. If *session* is not passed in, a new ISession object is created and returned.

launch_vm_process([*session*, *type_p*, *environment*])
This method sets the default values for the original *IMachine.launch_vm_process*. If *session* is not defined it will be created and on completion of the launch, will be unlocked. *type_p* is set to default 'gui' and *environment* is set to default ''.

class virtualbox.library_ext.**IConsole**

restore_snapshot([*snapshot*])
snapshot is now an optional argument. If it is not supplied, an attempt to pull the *machine.current_snapshot* is made, if there is no snapshot available, an Exception is raised.

register_on_network_adapter_changed(*callback*)
The *callback* function is called with a *INetworkAdapterChangedEvent* argument when a network adapter changed event occurs.

```
def callback(event):
    adapter = event.network_adapter
    print("Enabled = %s, connected = %s" % (adapter.enabled,
                                            adapter.cable_connected))

session.console.register_on_network_adapter_changed(callback)
```

register_on_serial_port_changed(*callback*)
The *callback* function is called with a *ISerialPortChangedEvent* argument when a serial port changed event occurs.

```
def callback(event):
    port = event.serial_port
    print("Enabled = %s, path = %s" % (port.enabled,
                                       port.path))

session.console.register_on_serial_port_changed(callback)
```

register_on_parallel_port_changed(*callback*)
The *callback* function is called with a *IParallelPortChangedEvent* argument on a parallel port changed event.

```
def callback(event):
    port = event.parallel_port
    print("Enabled = %s, path = %s" % (port.enabled,
                                       port.path))

session.console.register_on_parallel_port_changed(callback)
```

register_on_medium_changed(*callback*)
The *callback* function is called with a *IMediumChangedEvent* on a medium changed event.

```
def callback(event):
    medium = event.medimum_attachment
    print(medium.controller)

session.console.register_on_medium_changed(callback)
```

register_on_clipboard_mode_changed(*callback*)
The *callback* function is called with a *IClipboardModeChangedEvent* on a clipboard mode changed event.

```
def callback(event):
    print(event.clipboard_mode)

session.console.register_on_clipboard_mode_changed(callback)
```

register_on_drag_and_drop_mode_changed(*callback*)
The *callback* function is called with a *IDragAndDropModeChangedEvent* on a drag and drop mode changed event.

```
def callback(event):
    print(event.drag_and_drop_mode)

session.console.register_on_drag_and_drop_mode_changed(callback)
```

register_on_vrde_server_changed(*callback*)
The *callback* function is called with a *IVRDEServerChangedEvent* on a drag and drop mode changed event.

```
def callback(event):
    print("VirtualBox remote display extension server changed")

session.console.register_vdre_server_changed(callback)
```

register_on_additions_state_changed(*callback*)
The *callback* function is called with a *IAdditionsStateChangedEvent* argument on a additions state changed event. To find out what has changed, a probe into the attributes of IGuest is required.

```
def callback(event):
    print("State changed in IGuest...")

session.console.register_on_additions_state_changed(callback)
```

register_on_shared_folder_changed(*callback*)
The *callback* function is called with a *ISharedFolderChangedEvent* argument on a shared folder changed event.

```
def callback(event):
    print("Folder changed scope %s" % event.scope)

session.console.register_on_shared_folder_changed(callback)
```

register_on_state_changed(*callback*)
The *callback* function is called with a *IStateChangedEvent* on a machine state changed event.

```
def callback(event):
    print("State changed to %s" % event.state)

session.console.register_on_state_changed(callback)
```

register_on_event_source_changed(*callback*)
The *callback* function is called with a *IEventSourceChangedEvent* on a event source changed event. This occurs when a listener is added or removed.

```
def callback(event):
    if event.add:
        action = 'added'
    else:
        action = 'removed'
    print("A listener was %s from console's event_source %s" % \
            action)

session.console.register_on_event_source_changed(callback)
```

register_on_can_show_window(*callback*)
The *callback* function is called with a *ICanShowWindowEvent* on a show window event. This occurs when the console window is to be activated and brought to the foreground of the desktop of the host PC. If this behaviour is not desired a call to event.add_veto will stop this from happening.

```
def callback(event):
    print("veto this event")
    event.add_veto("No you shall never do this!")

session.console.register_on_can_show_window(callback)
```

register_on_show_window(*callback*)
The *callback* function is called with a *IShowWindowEvent* on a show window event. This occurs when the console window is to be activated and brought to the foreground of the desktop of the host PC.

```
def callback(event):
    print("Window id = %s" % event.win_id)

session.console.register_on_show_window(callback)
```

3.4 virtualbox.events – registration, listening and processing

The *virtualbox.events* module is responsible for the registering and unregistering callback functions against a specific event source and event type.

All callbacks registered by this module will be cleared *atexit*.

3.4.1 Code reference

virtualbox.events.**register_callback**(*callback*, *event_source*, *event_type*)
Register a callback function against an event_source for a given event_type.

Any object in the VirtualBox API that generates an event aggregates an event_source (IEventSource) object through its interface object. Specific event_type's (VBoxEventType) can be raised through this event_source.

Once a listener has been created and registered through to the VBoxSvr, a thread is spawned to block on the *event_source.get_event* call. When an event (IEvent) is successfully read, the callback will be called with a type case from the IEvent object to the Interface type that has an id of specific VBoxEventType that has been listened too.

An Integer is returned from this register_callback which is used as the ID of the registered callback function.

The following code snippet demonstrates how a callback can be registered against a specific event_type.

```
def on_property_change(event):
    print("%s %s %s" % (event.name, event.value, event.flags))

vbox = virtualbox.VirtualBox()
event.register_callback(on_property_change, vbox.event_source,
                        library.VBoxEventType.on_guest_property_changed)
```

virtualbox.events.**unregister_callback**(*callback_id*)

Unregister a callback function using the callback_id returned from the register_callback method.

Each event listener blocks on an event read for 1 second than checks the listener's quit Event status.

3.5 virtualbox.library – transform of VirtualBox.xidl

The *virtualbox.library* is generated using the VirtualBox project's VirtualBox.xidl file. This file contains a complete definition of the VirtualBox interface.

pyvbox ships with a builder in it's root folder called build.py. This builder is responsible for implementing the code that transforms VirtualBox.xidl into library.py.

3.5.1 Code reference

This code reference is the result of using *automodule* to generate code for the entire *virtualbox.library* module, followed by *autoclass* to generate doc for the extended classes found in *library_ext*.

3.5.2 virtualbox.library

Welcome to the **VirtualBox Main API documentation**. This documentation describes the so-called *VirtualBox Main API* which comprises all public COM interfaces and components provided by the VirtualBox server and by the VirtualBox client library.

VirtualBox employs a client-server design, meaning that whenever any part of VirtualBox is running – be it the Qt GUI, the VBoxManage command-line interface or any virtual machine –, a dedicated server process named VBoxSVC runs in the background. This allows multiple processes working with VirtualBox to cooperate without conflicts. These processes communicate to each other using inter-process communication facilities provided by the COM implementation of the host computer.

On Windows platforms, the VirtualBox Main API uses Microsoft COM, a native COM implementation. On all other platforms, Mozilla XPCOM, an open-source COM implementation, is used.

All the parts that a typical VirtualBox user interacts with (the Qt GUI and the VBoxManage command-line interface) are technically front-ends to the Main API and only use the interfaces that are documented in this Main API documentation. This ensures that, with any given release version of VirtualBox, all capabilities of the product that could be useful to an external client program are always exposed by way of this API.

The VirtualBox Main API (also called the *VirtualBox COM library*) contains two public component classes: *IVirtualBox* and *ISession*, which implement IVirtualBox and ISession interfaces respectively. These two classes are of supreme importance and will be needed in order for any front-end program to do anything useful. It is recommended to read the documentation of the mentioned interfaces first.

The *IVirtualBox* class is a singleton. This means that there can be only one object of this class on the local machine at any given time. This object is a parent of many other objects in the VirtualBox COM library and lives in the VBoxSVC process. In fact, when you create an instance of the *IVirtualBox*, the COM subsystem checks if the VBoxSVC process is already running, starts it if not, and returns you a reference to the VirtualBox object created in this process. When the last reference to this object is released, the VBoxSVC process ends (with a 5 second delay to protect from too frequent restarts).

The *ISession* class is a regular component. You can create as many Session objects as you need but all of them will live in a process which issues the object instantiation call. Session objects represent virtual machine sessions which are used to configure virtual machines and control their execution.

The naming of methods and attributes is very clearly defined: they all start with a lowercase letter (except if they start with an acronym), and are using CamelCase style otherwise. This naming only applies to the IDL description, and is modified by the various language bindings (some convert the first character to upper case, some not). See the SDK reference for more details about how to call a method or attribute from a specific programming language.

exception virtualbox.library.**VBoxErrorObjectNotFound**
Object corresponding to the supplied arguments does not exist.

exception virtualbox.library.**VBoxErrorInvalidVmState**
Current virtual machine state prevents the operation.

exception virtualbox.library.**VBoxErrorVmError**
Virtual machine error occurred attempting the operation.

exception virtualbox.library.**VBoxErrorFileError**
File not accessible or erroneous file contents.

exception virtualbox.library.**VBoxErrorIprtError**
Runtime subsystem error.

exception virtualbox.library.**VBoxErrorPdmError**
Pluggable Device Manager error.

exception virtualbox.library.**VBoxErrorInvalidObjectState**
Current object state prohibits operation.

exception virtualbox.library.**VBoxErrorHostError**
Host operating system related error.

exception virtualbox.library.**VBoxErrorNotSupported**
Requested operation is not supported.

exception virtualbox.library.**VBoxErrorXmlError**
Invalid XML found.

exception virtualbox.library.**VBoxErrorInvalidSessionState**
Current session state prohibits operation.

exception virtualbox.library.**VBoxErrorObjectInUse**
Object being in use prohibits operation.

exception virtualbox.library.**VBoxErrorPasswordIncorrect**
A provided password was incorrect.

exception virtualbox.library.**OleErrorFail**
Unspecified error

exception virtualbox.library.**OleErrorNointerface**
No such interface supported

exception virtualbox.library.**OleErrorAccessdenied**
General access denied error

exception virtualbox.library.**OleErrorNotimpl**
Not implemented

exception virtualbox.library.**OleErrorUnexpected**
Catastrophic failure

exception virtualbox.library.**OleErrorInvalidarg**
One or more arguments are invalid

class virtualbox.library.**SettingsVersion**(*value*)
Settings version of VirtualBox settings files. This is written to the "version" attribute of the root "VirtualBox" element in the settings file XML and indicates which VirtualBox version wrote the file.

null(0)
Null value, indicates invalid version.

v1_0(1)
Legacy settings version, not currently supported.

v1_1(2)
Legacy settings version, not currently supported.

v1_2(3)
Legacy settings version, not currently supported.

v1_3pre(4)
Legacy settings version, not currently supported.

v1_3(5)
Settings version "1.3", written by VirtualBox 2.0.12.

v1_4(6)
Intermediate settings version, understood by VirtualBox 2.1.x.

v1_5(7)
Intermediate settings version, understood by VirtualBox 2.1.x.

v1_6(8)
Settings version "1.6", written by VirtualBox 2.1.4 (at least).

v1_7(9)
Settings version "1.7", written by VirtualBox 2.2.x and 3.0.x.

v1_8(10)
Intermediate settings version "1.8", understood by VirtualBox 3.1.x.

v1_9(11)
Settings version "1.9", written by VirtualBox 3.1.x.

v1_10(12)
Settings version "1.10", written by VirtualBox 3.2.x.

v1_11(13)
Settings version "1.11", written by VirtualBox 4.0.x.

v1_12(14)
Settings version "1.12", written by VirtualBox 4.1.x.

v1_13(15)
Settings version "1.13", written by VirtualBox 4.2.x.

v1_14(16)
Settings version "1.14", written by VirtualBox 4.3.x.

v1_15(17)
Settings version "1.15", written by VirtualBox 5.0.x.

v1_16(18)
Settings version "1.16", written by VirtualBox 5.1.x.

future(99999)
Settings version greater than "1.15", written by a future VirtualBox version.

future = SettingsVersion(99999)

null = SettingsVersion(0)

v1_0 = SettingsVersion(1)

v1_1 = SettingsVersion(2)

v1_10 = SettingsVersion(12)

v1_11 = SettingsVersion(13)

v1_12 = SettingsVersion(14)

v1_13 = SettingsVersion(15)

v1_14 = SettingsVersion(16)

v1_15 = SettingsVersion(17)

v1_16 = SettingsVersion(18)

v1_2 = SettingsVersion(3)

v1_3 = SettingsVersion(5)

v1_3pre = SettingsVersion(4)

v1_4 = SettingsVersion(6)

v1_5 = SettingsVersion(7)

v1_6 = SettingsVersion(8)

v1_7 = SettingsVersion(9)

v1_8 = SettingsVersion(10)

v1_9 = SettingsVersion(11)

class virtualbox.library.**AccessMode**(*value*)
Access mode for opening files.

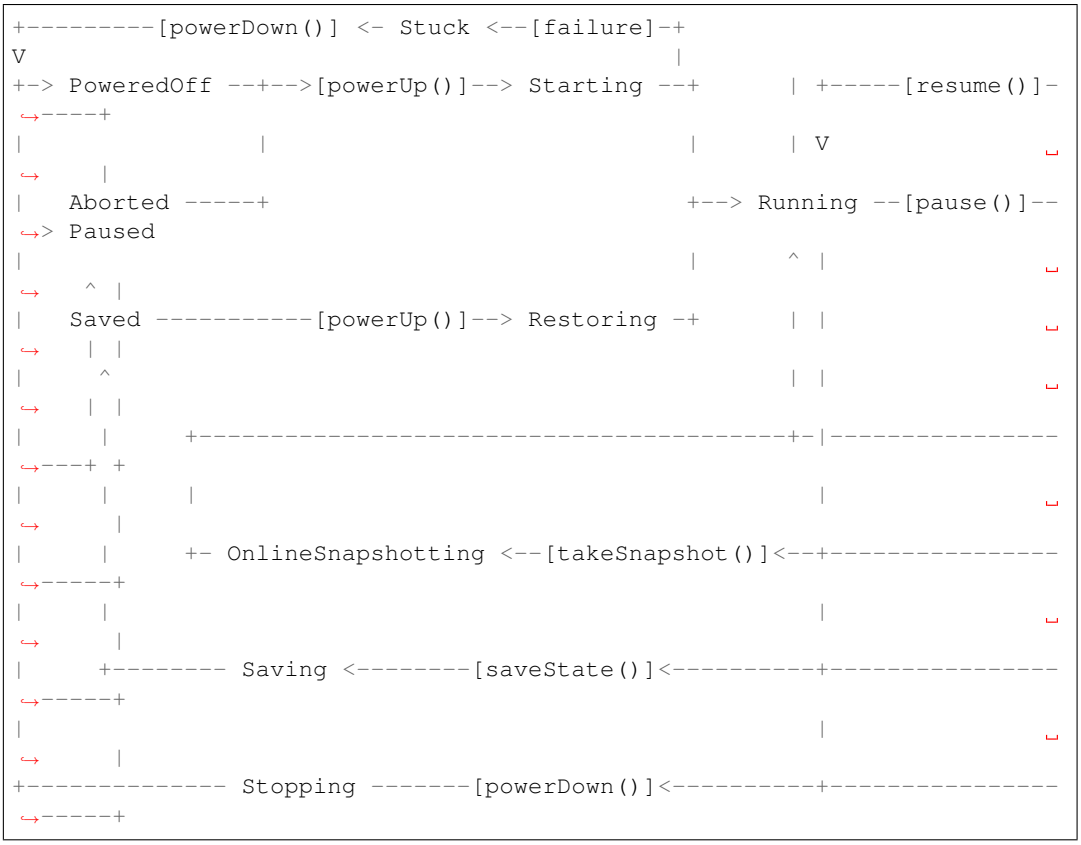
read_only(1)

read_write(2)

class virtualbox.library.**MachineState**(*value*)
Virtual machine execution state.

This enumeration represents possible values of the *IMachine.state()* attribute.

Below is the basic virtual machine state diagram. It shows how the state changes during virtual machine execution. The text in square braces shows a method of the IConsole or IMachine interface that performs the given state transition.



Note that states to the right from PoweredOff, Aborted and Saved in the above diagram are called *online VM states*. These states represent the virtual machine which is being executed in a dedicated process (usually with a GUI window attached to it where you can see the activity of the virtual machine and interact with it). There are two special pseudo-states, FirstOnline and LastOnline, that can be used in relational expressions to detect if the given machine state is online or not:

```
if (machine.GetState() >= MachineState_FirstOnline &&
machine.GetState() <= MachineState_LastOnline)
{
...the machine is being executed...
}
```

When the virtual machine is in one of the online VM states (that is, being executed), only a few machine settings can be modified. Methods working with such settings contain an explicit note about that. An attempt to change any other setting or perform a modifying operation during this time will result in the @c VBOX_E_INVALID_VM_STATE error.

All online states except Running, Paused and Stuck are transitional: they represent temporary conditions of the virtual machine that will last as long as the operation that initiated such a condition.

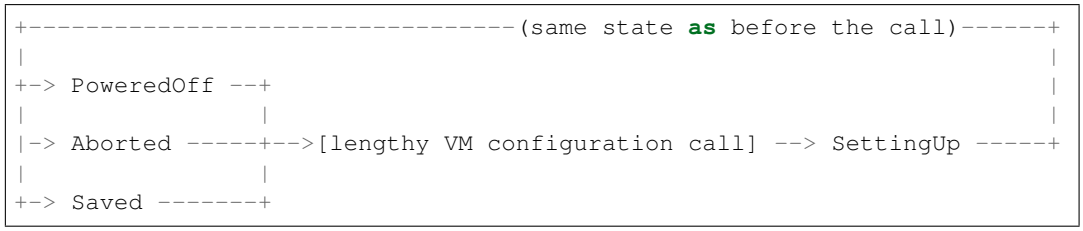
The Stuck state is a special case. It means that execution of the machine has reached the "Guru Meditation" condition. This condition indicates an internal VMM (virtual machine manager) failure which may happen as a result of either an unhandled low-level virtual hardware exception or one of the recompiler exceptions (such as the *too-many-traps* condition).

Note also that any online VM state may transit to the Aborted state. This happens if the process that

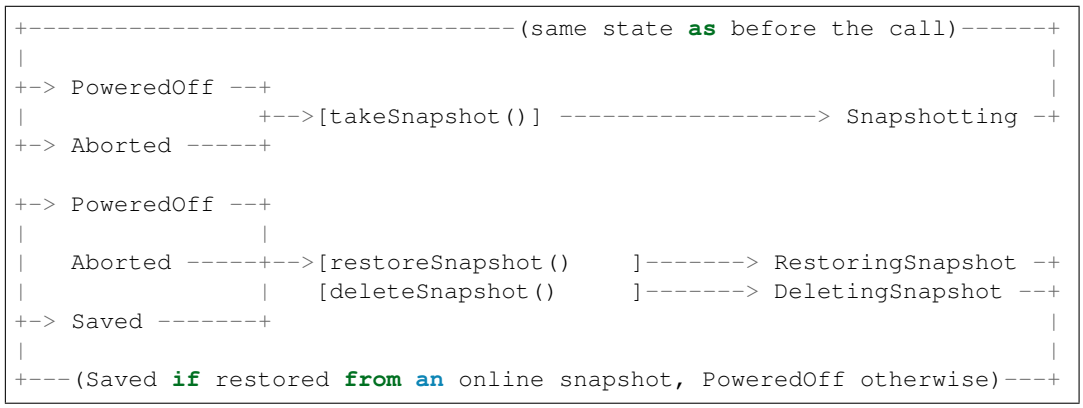
is executing the virtual machine terminates unexpectedly (for example, crashes). Other than that, the Aborted state is equivalent to PoweredOff.

There are also a few additional state diagrams that do not deal with virtual machine execution and therefore are shown separately. The states shown on these diagrams are called *offline VM states* (this includes PoweredOff, Aborted and Saved too).

The first diagram shows what happens when a lengthy setup operation is being executed (such as *IMachine.attach_device()*).



The next two diagrams demonstrate the process of taking a snapshot of a powered off virtual machine, restoring the state to that as of a snapshot or deleting a snapshot, respectively.



For whoever decides to touch this enum: In order to keep the comparisons involving FirstOnline and LastOnline pseudo-states valid, the numeric values of these states must be correspondingly updated if needed: for any online VM state, the condition FirstOnline <= state <= LastOnline must be @c true. The same relates to transient states for which the condition FirstOnline <= state <= LastOnline must be @c true.

null(0)
Null value (never used by the API).

powered_off(1)
The machine is not running and has no saved execution state; it has either never been started or been shut down successfully.

saved(2)
The machine is not currently running, but the execution state of the machine has been saved to an external file when it was running, from where it can be resumed.

teleported(3)
The machine was teleported to a different host (or process) and then powered off. Take care when powering it on again may corrupt resources it shares with the teleportation target (e.g. disk and network).

aborted(4)
The process running the machine has terminated abnormally. This may indicate a crash of the VM process in host execution context, or the VM process has been terminated externally.

running(5)
The machine is currently being executed.

For whoever decides to touch this enum: In order to keep the comparisons in the old source code valid, this state must immediately precede the Paused state.

@todo Lift this spectacularly wonderful restriction.

paused(6)
Execution of the machine has been paused.

For whoever decides to touch this enum: In order to keep the comparisons in the old source code valid, this state must immediately follow the Running state.

@todo Lift this spectacularly wonderful restriction.

stuck(7)
Execution of the machine has reached the "Guru Meditation" condition. This indicates a severe error in the hypervisor itself.

bird: Why this uncool name? Could we rename it to "GuruMeditation" or "Guru", perhaps? Or are there some other VMM states that are intended to be lumped in here as well?

teleporting(8)
The machine is about to be teleported to a different host or process. It is possible to pause a machine in this state, but it will go to the @c TeleportingPausedVM state and it will not be possible to resume it again unless the teleportation fails.

live_snapshotting(9)
A live snapshot is being taken. The machine is running normally, but some of the runtime configuration options are inaccessible. Also, if paused while in this state it will transition to @c OnlineSnapshotting and it will not be resume the execution until the snapshot operation has completed.

starting(10)
Machine is being started after powering it on from a zero execution state.

stopping(11)
Machine is being normally stopped powering it off, or after the guest OS has initiated a shutdown sequence.

saving(12)
Machine is saving its execution state to a file.

restoring(13)
Execution state of the machine is being restored from a file after powering it on from the saved execution state.

teleporting_paused_vm(14)
The machine is being teleported to another host or process, but it is not running. This is the paused variant of the @c Teleporting state.

teleporting_in(15)
Teleporting the machine state in from another host or process.

fault_tolerant_syncing(16)
The machine is being synced with a fault tolerant VM running elsewhere.

deleting_snapshot_online(17)
Like @c DeletingSnapshot, but the merging of media is ongoing in the background while the machine is running.

deleting_snapshot_paused(18)
Like @c DeletingSnapshotOnline, but the machine was paused when the merging of differencing media was started.

online_snapshotting(19)
Like @c LiveSnapshotting, but the machine was paused when the merging of differencing media was started.

restoring_snapshot(20)
A machine snapshot is being restored; this typically does not take long.

deleting_snapshot(21)
A machine snapshot is being deleted; this can take a long time since this may require merging differencing media. This value indicates that the machine is not running while the snapshot is being deleted.

setting_up(22)
Lengthy setup operation is in progress.

snapshotting(23)
Taking an (offline) snapshot.

first_online(5)
Pseudo-state: first online state (for use in relational expressions).

last_online(19)
Pseudo-state: last online state (for use in relational expressions).

first_transient(8)
Pseudo-state: first transient state (for use in relational expressions).

last_transient(23)
Pseudo-state: last transient state (for use in relational expressions).

aborted = MachineState(4)

deleting_snapshot = MachineState(21)

deleting_snapshot_online = MachineState(17)

deleting_snapshot_paused = MachineState(18)

fault_tolerant_syncing = MachineState(16)

first_online = MachineState(5)

first_transient = MachineState(8)

last_online = MachineState(19)

last_transient = MachineState(23)

live_snapshotting = MachineState(9)

null = MachineState(0)

online_snapshotting = MachineState(19)

paused = MachineState(6)

powered_off = MachineState(1)

restoring = MachineState(13)

restoring_snapshot = MachineState(20)

running = MachineState(5)

saved = MachineState(2)

saving = MachineState(12)

setting_up = MachineState(22)

snapshotting = MachineState(23)

starting = MachineState(10)

stopping = MachineState(11)

stuck = MachineState(7)

teleported = MachineState(3)

teleporting = MachineState(8)

teleporting_in = MachineState(15)

teleporting_paused_vm = MachineState(14)

class virtualbox.library.**SessionState**(*value*)

Session state. This enumeration represents possible values of *IMachine.session_state()* and *ISession.state()* attributes.

null(0)

Null value (never used by the API).

unlocked(1)

In *IMachine.session_state()* , this means that the machine is not locked for any sessions.

In *ISession.state()* , this means that no machine is currently locked for this session.

locked(2)

In *IMachine.session_state()* , this means that the machine is currently locked for a session, whose process identifier can then be found in the *IMachine.session_pid()* attribute.

In *ISession.state()* , this means that a machine is currently locked for this session, and the mutable machine object can be found in the *ISession.machine()* attribute (see *IMachine.lock_machine()* for details).

spawning(3)

A new process is being spawned for the machine as a result of *IMachine.launch_vm_process()* call. This state also occurs as a short transient state during an *IMachine.lock_machine()* call.

unlocking(4)

The session is being unlocked.

locked = SessionState(2)

null = SessionState(0)

spawning = SessionState(3)

unlocked = SessionState(1)

unlocking = SessionState(4)

class virtualbox.library.**CPUPropertyType**(*value*)

Virtual CPU property type. This enumeration represents possible values of the IMachine get- and setCPUProperty methods.

null(0)
Null value (never used by the API).

pae(1)
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.

long_mode(2)
This setting determines whether VirtualBox will advertise long mode (i.e. 64-bit guest support) and let the guest enter it.

triple_fault_reset(3)
This setting determines whether a triple fault within a guest will trigger an internal error condition and stop the VM (default) or reset the virtual CPU/VM and continue execution.

apic(4)
This setting determines whether an APIC is part of the virtual CPU. This feature can only be turned off when the X2APIC feature is off.

x2_apic(5)
This setting determines whether an x2APIC is part of the virtual CPU. Since this feature implies that the APIC feature is present, it automatically enables the APIC feature when set.

apic = CPUPropertyType(4)

long_mode = CPUPropertyType(2)

null = CPUPropertyType(0)

pae = CPUPropertyType(1)

triple_fault_reset = CPUPropertyType(3)

x2_apic = CPUPropertyType(5)

class virtualbox.library.**HWVirtExPropertyType**(*value*)
Hardware virtualization property type. This enumeration represents possible values for the *IMachine.get_hw_virt_ex_property()* and *IMachine.set_hw_virt_ex_property()* methods.

null(0)
Null value (never used by the API).

enabled(1)
Whether hardware virtualization (VT-x/AMD-V) is enabled at all. If such extensions are not available, they will not be used.

vpid(2)
Whether VT-x VPID is enabled. If this extension is not available, it will not be used.

nested_paging(3)
Whether Nested Paging is enabled. If this extension is not available, it will not be used.

unrestricted_execution(4)
Whether VT-x unrestricted execution is enabled. If this feature is not available, it will not be used.

large_pages(5)
Whether large page allocation is enabled; requires nested paging and a 64-bit host.

force(6)
Whether the VM should fail to start if hardware virtualization (VT-x/AMD-V) cannot be used. If not set, there will be an automatic fallback to software virtualization.

enabled = HWVirtExPropertyType(1)

force = HWVirtExPropertyType(6)

large_pages = HWVirtExPropertyType(5)

nested_paging = HWVirtExPropertyType(3)

null = HWVirtExPropertyType(0)

unrestricted_execution = HWVirtExPropertyType(4)

vpid = HWVirtExPropertyType(2)

class virtualbox.library.**ParavirtProvider**(*value*)
The paravirtualized guest interface provider. This enumeration represents possible values for the *IMachine.paravirt_provider()* attribute.

none(0)
No provider is used.

default(1)
A default provider is automatically chosen according to the guest OS type.

legacy(2)
Used for VMs which didn't used to have any provider settings. Usually interpreted as @c None for most VMs.

minimal(3)
A minimal set of features to expose to the paravirtualized guest.

hyper_v(4)
Microsoft Hyper-V.

kvm(5)
Linux KVM.

default = ParavirtProvider(1)

hyper_v = ParavirtProvider(4)

kvm = ParavirtProvider(5)

legacy = ParavirtProvider(2)

minimal = ParavirtProvider(3)

none = ParavirtProvider(0)

class virtualbox.library.**FaultToleranceState**(*value*)
Used with *IMachine.fault_tolerance_state()* .

inactive(1)
No fault tolerance enabled.

master(2)
Fault tolerant master VM.

standby(3)
Fault tolerant standby VM.

inactive = FaultToleranceState(1)

master = FaultToleranceState(2)

standby = FaultToleranceState(3)

class virtualbox.library.**LockType**(*value*)
Used with *IMachine.lock_machine()* .

null(0)
Placeholder value, do not use when obtaining a lock.

shared(1)
Request only a shared lock for remote-controlling the machine. Such a lock allows changing certain VM settings which can be safely modified for a running VM.

write(2)
Lock the machine for writing. This requests an exclusive lock, i.e. there cannot be any other API client holding any type of lock for this VM concurrently. Remember that a VM process counts as an API client which implicitly holds the equivalent of a shared lock during the entire VM runtime.

vm(3)
Lock the machine for writing, and create objects necessary for running a VM in this process.

null = LockType(0)

shared = LockType(1)

vm = LockType(3)

write = LockType(2)

class virtualbox.library.**SessionType**(*value*)
Session type. This enumeration represents possible values of the *ISession.type_p()* attribute.

null(0)
Null value (never used by the API).

write_lock(1)
Session has acquired an exclusive write lock on a machine using *IMachine.lock_machine()* .

remote(2)
Session has launched a VM process using *IMachine.launch_vm_process()*

shared(3)
Session has obtained a link to another session using *IMachine.lock_machine()*

null = SessionType(0)

remote = SessionType(2)

shared = SessionType(3)

write_lock = SessionType(1)

class virtualbox.library.**DeviceType**(*value*)
Device type.

null(0)
Null value, may also mean "no device" (not allowed for *IConsole.get_device_activity()*).

floppy(1)
Floppy device.

dvd(2)
CD/DVD-ROM device.

hard_disk(3)
Hard disk device.

network(4)
Network device.

usb(5)
USB device.

shared_folder(6)
Shared folder device.

graphics3_d(7)
Graphics device 3D activity.

dvd = DeviceType(2)

floppy = DeviceType(1)

graphics3_d = DeviceType(7)

hard_disk = DeviceType(3)

network = DeviceType(4)

null = DeviceType(0)

shared_folder = DeviceType(6)

usb = DeviceType(5)

class virtualbox.library.**DeviceActivity**(*value*)
Device activity for *IConsole.get_device_activity()* .

null(0)

idle(1)

reading(2)

writing(3)

class virtualbox.library.**ClipboardMode**(*value*)
Host-Guest clipboard interchange mode.

disabled(0)

host_to_guest(1)

guest_to_host(2)

bidirectional(3)

class virtualbox.library.**DnDMode**(*value*)
Drag and drop interchange mode.

disabled(0)

host_to_guest(1)

guest_to_host(2)

bidirectional(3)

class virtualbox.library.**Scope**(*value*)
Scope of the operation.

A generic enumeration used in various methods to define the action or argument scope.

global_p(0)

machine(1)

session(2)

class virtualbox.library.**BIOSBootMenuMode**(*value*)
BIOS boot menu mode.

disabled(0)

menu_only(1)

message_and_menu(2)

class virtualbox.library.**APICMode**(*value*)
BIOS APIC initialization mode. If the hardware does not support the mode then the code falls back to a lower mode.

disabled(0)

apic(1)

x2_apic(2)

class virtualbox.library.**ProcessorFeature**(*value*)
CPU features.

hw_virt_ex(0)

pae(1)

long_mode(2)

nested_paging(3)

class virtualbox.library.**FirmwareType**(*value*)
Firmware type.

bios(1)
BIOS Firmware.

efi(2)
EFI Firmware, bitness detected basing on OS type.

efi32(3)
EFI firmware, 32-bit.

efi64(4)
EFI firmware, 64-bit.

efidual(5)
EFI firmware, combined 32 and 64-bit.

bios = FirmwareType(1)

efi = FirmwareType(2)

efi32 = FirmwareType(3)

efi64 = FirmwareType(4)

efidual = FirmwareType(5)

class virtualbox.library.**PointingHIDType**(*value*)
Type of pointing device used in a virtual machine.

none(1)
No mouse.

ps2_mouse(2)
PS/2 auxiliary device, a.k.a. mouse.

usb_mouse(3)
USB mouse (relative pointer).

usb_tablet(4)
USB tablet (absolute pointer). Also enables a relative USB mouse in addition.

combo_mouse(5)
Combined device, working as PS/2 or USB mouse, depending on guest behavior. Using this device can have negative performance implications.

usb_multi_touch(6)
USB multi-touch device. Also enables the USB tablet and mouse devices.

combo_mouse = PointingHIDType(5)

none = PointingHIDType(1)

ps2_mouse = PointingHIDType(2)

usb_mouse = PointingHIDType(3)

usb_multi_touch = PointingHIDType(6)

usb_tablet = PointingHIDType(4)

class virtualbox.library.**KeyboardHIDType**(*value*)
Type of keyboard device used in a virtual machine.

none(1)
No keyboard.

ps2_keyboard(2)
PS/2 keyboard.

usb_keyboard(3)
USB keyboard.

combo_keyboard(4)
Combined device, working as PS/2 or USB keyboard, depending on guest behavior. Using of such device can have negative performance implications.

combo_keyboard = KeyboardHIDType(4)

none = KeyboardHIDType(1)

ps2_keyboard = KeyboardHIDType(2)

usb_keyboard = KeyboardHIDType(3)

class virtualbox.library.**BitmapFormat**(*value*)
Format of a bitmap. Generic values for formats used by the source bitmap, the screen shot or image update APIs.

opaque(0)
Unknown buffer format (the user may not assume any particular format of the buffer).

bgr(542263106)
Generic BGR format without alpha channel. Pixel layout depends on the number of bits per pixel:

32 - bits 31:24 undefined, bits 23:16 R, bits 15:8 G, bits 7:0 B.

16 - bits 15:11 R, bits 10:5 G, bits 4:0 B.

bgr0(810698562)
4 bytes per pixel: B, G, R, 0.

bgra(1095911234)
4 bytes per pixel: B, G, R, A.

rgba(1094862674)
4 bytes per pixel: R, G, B, A.

png(541544016)
PNG image.

jpeg(1195724874)
JPEG image.

bgr = BitmapFormat(542263106)

bgr0 = BitmapFormat(810698562)

bgra = BitmapFormat(1095911234)

jpeg = BitmapFormat(1195724874)

opaque = BitmapFormat(0)

png = BitmapFormat(541544016)

rgba = BitmapFormat(1094862674)

class virtualbox.library.**DhcpOpt**(*value*)

subnet_mask(1)

time_offset(2)

router(3)

time_server(4)

name_server(5)

domain_name_server(6)

log_server(7)

cookie(8)

lpr_server(9)

impress_server(10)

resourse_location_server(11)

host_name(12)

boot_file_size(13)

merit_dump_file(14)

domain_name(15)

swap_server(16)

root_path(17)

extension_path(18)

ip_forwarding_enable_disable(19)

non_local_source_routing_enable_disable(20)

policy_filter(21)

maximum_datagram_reassembly_size(22)

default_ip_time2_live(23)

path_mtu_aging_timeout(24)

ip_layer_parameters_per_interface(25)

interface_mtu(26)

all_subnets_are_local(27)

broadcast_address(28)

perform_mask_discovery(29)

mask_supplier(30)

perform_route_discovery(31)

router_solicitation_address(32)

static_route(33)

trailer_encapsulation(34)

arp_cache_timeout(35)

ethernet_encapsulation(36)

tcp_default_ttl(37)

tcp_keep_alive_interval(38)

tcp_keep_alive_garbage(39)

network_information_service_domain(40)

network_information_service_servers(41)

network_time_protocol_servers(42)

vendor_specific_information(43)

option_44(44)

option_45(45)

option_46(46)

option_47(47)

option_48(48)

option_49(49)

ip_address_lease_time(51)

option_64(64)

option_65(65)

tftp_server_name(66)

bootfile_name(67)

option_68(68)

option_69(69)

option_70(70)

option_71(71)

option_72(72)

option_73(73)

option_74(74)

option_75(75)

option_119(119)

class virtualbox.library.**DhcpOptEncoding**(*value*)

legacy(0)

hex_p(1)

class virtualbox.library.**VFSType**(*value*)
Virtual file systems supported by VFSExplorer.

file_p(1)

cloud(2)

s3(3)

web_dav(4)

class virtualbox.library.**ImportOptions**(*value*)
Import options, used with *IAppliance.import_machines()* .

keep_all_ma_cs(1)
Don't generate new MAC addresses of the attached network adapters.

keep_natma_cs(2)
Don't generate new MAC addresses of the attached network adapters when they are using NAT.

import_to_vdi(3)
Import all disks to VDI format

import_to_vdi = ImportOptions(3)

keep_all_ma_cs = ImportOptions(1)

keep_natma_cs = ImportOptions(2)

class virtualbox.library.**ExportOptions**(*value*)
Export options, used with *IAppliance.write()* .

create_manifest(1)
Write the optional manifest file (.mf) which is used for integrity checks prior import.

export_dvd_images(2)
Export DVD images. Default is not to export them as it is rarely needed for typical VMs.

strip_all_ma_cs(3)
Do not export any MAC address information. Default is to keep them to avoid losing information which can cause trouble after import, at the price of risking duplicate MAC addresses, if the import options are used to keep them.

strip_all_non_natma_cs(4)
Do not export any MAC address information, except for adapters using NAT. Default is to keep them to avoid losing information which can cause trouble after import, at the price of risking duplicate MAC addresses, if the import options are used to keep them.

create_manifest = ExportOptions(1)

export_dvd_images = ExportOptions(2)

strip_all_ma_cs = ExportOptions(3)

strip_all_non_natma_cs = ExportOptions(4)

class virtualbox.library.**CertificateVersion**(*value*)
X.509 certificate version numbers.

v1(1)

v2(2)

v3(3)

unknown(99)

class virtualbox.library.**VirtualSystemDescriptionType**(*value*)
Used with *IVirtualSystemDescription* to describe the type of a configuration value.

ignore(1)

os(2)

name(3)

product(4)

vendor(5)

version(6)

product_url(7)

vendor_url(8)

description(9)

license_p(10)

miscellaneous(11)

cpu(12)

memory(13)

hard_disk_controller_ide(14)

hard_disk_controller_sata(15)

hard_disk_controller_scsi(16)

hard_disk_controller_sas(17)

hard_disk_image(18)

floppy(19)

cdrom(20)

network_adapter(21)

usb_controller(22)

sound_card(23)

settings_file(24)
Not used/implemented right now, will be added later in 4.1.x.

settings_file = VirtualSystemDescriptionType(24)

class virtualbox.library.**VirtualSystemDescriptionValueType**(*value*)
Used with *IVirtualSystemDescription.get_values_by_type()* to describe the value type to fetch.

reference(1)

original(2)

auto(3)

extra_config(4)

class virtualbox.library.**GraphicsControllerType**(*value*)
Graphics controller type, used with *IMachine.unregister()* .

null(0)
Reserved value, invalid.

v_box_vga(1)
Default VirtualBox VGA device.

vmsvga(2)
VMware SVGA II device.

null = GraphicsControllerType(0)

v_box_vga = GraphicsControllerType(1)

vmsvga = GraphicsControllerType(2)

class virtualbox.library.**CleanupMode**(*value*)
Cleanup mode, used with *IMachine.unregister()* .

unregister_only(1)
Unregister only the machine, but neither delete snapshots nor detach media.

detach_all_return_none(2)
Delete all snapshots and detach all media but return none; this will keep all media registered.

detach_all_return_hard_disks_only(3)
Delete all snapshots, detach all media and return hard disks for closing, but not removable media.

full(4)
Delete all snapshots, detach all media and return all media for closing.

detach_all_return_hard_disks_only = CleanupMode(3)

detach_all_return_none = CleanupMode(2)

full = CleanupMode(4)

unregister_only = CleanupMode(1)

class virtualbox.library.**CloneMode**(*value*)
Clone mode, used with *IMachine.clone_to()* .

machine_state(1)
Clone the state of the selected machine.

machine_and_child_states(2)
Clone the state of the selected machine and its child snapshots if present.

all_states(3)
Clone all states (including all snapshots) of the machine, regardless of the machine object used.

all_states = CloneMode(3)

machine_and_child_states = CloneMode(2)

machine_state = CloneMode(1)

class virtualbox.library.**CloneOptions**(*value*)
Clone options, used with *IMachine.clone_to()* .

link(1)
Create a clone VM where all virtual disks are linked to the original VM.

keep_all_ma_cs(2)
Don't generate new MAC addresses of the attached network adapters.

keep_natma_cs(3)
Don't generate new MAC addresses of the attached network adapters when they are using NAT.

keep_disk_names(4)
Don't change the disk names.

keep_all_ma_cs = CloneOptions(2)

keep_disk_names = CloneOptions(4)

keep_natma_cs = CloneOptions(3)

link = CloneOptions(1)

class virtualbox.library.**AutostopType**(*value*)
Autostop types, used with *IMachine.autostop_type()* .

disabled(1)
Stopping the VM during system shutdown is disabled.

save_state(2)
The state of the VM will be saved when the system shuts down.

power_off(3)
The VM is powered off when the system shuts down.

acpi_shutdown(4)
An ACPI shutdown event is generated.

acpi_shutdown = AutostopType(4)

disabled = AutostopType(1)

power_off = AutostopType(3)

save_state = AutostopType(2)

class virtualbox.library.**HostNetworkInterfaceMediumType**(*value*)
Type of encapsulation. Ethernet encapsulation includes both wired and wireless Ethernet connections. *IHostNetworkInterface*

unknown(0)
The type of interface cannot be determined.

ethernet(1)
Ethernet frame encapsulation.

ppp(2)
Point-to-point protocol encapsulation.

slip(3)
Serial line IP encapsulation.

ethernet = HostNetworkInterfaceMediumType(1)

ppp = HostNetworkInterfaceMediumType(2)

slip = HostNetworkInterfaceMediumType(3)

unknown = HostNetworkInterfaceMediumType(0)

class virtualbox.library.**HostNetworkInterfaceStatus**(*value*)
Current status of the interface. *IHostNetworkInterface*

unknown(0)
The state of interface cannot be determined.

up(1)
The interface is fully operational.

down(2)
The interface is not functioning.

down = HostNetworkInterfaceStatus(2)

unknown = HostNetworkInterfaceStatus(0)

up = HostNetworkInterfaceStatus(1)

class virtualbox.library.**HostNetworkInterfaceType**(*value*)
Network interface type.

bridged(1)

host_only(2)

class virtualbox.library.**AdditionsFacilityType**(*value*)
Guest Additions facility IDs.

none(0)
No/invalid facility.

v_box_guest_driver(20)
VirtualBox base driver (VBoxGuest).

auto_logon(90)
Auto-logon modules (VBoxGINA, VBoxCredProv, pam_vbox).

v_box_service(100)
VirtualBox system service (VBoxService).

v_box_tray_client(101)
VirtualBox desktop integration (VBoxTray on Windows, VBoxClient on non-Windows).

seamless(1000)
Seamless guest desktop integration.

graphics(1100)
Guest graphics mode. If not enabled, seamless rendering will not work, resize hints are not immediately acted on and guest display resizes are probably not initiated by the guest additions.

all_p(2147483646)
All facilities selected.

all_p = AdditionsFacilityType(2147483646)

auto_logon = AdditionsFacilityType(90)

graphics = AdditionsFacilityType(1100)

none = AdditionsFacilityType(0)

seamless = AdditionsFacilityType(1000)

v_box_guest_driver = AdditionsFacilityType(20)

v_box_service = AdditionsFacilityType(100)

v_box_tray_client = AdditionsFacilityType(101)

class virtualbox.library.**AdditionsFacilityClass**(*value*)
Guest Additions facility classes.

none(0)
No/invalid class.

driver(10)
Driver.

service(30)
System service.

program(50)
Program.

feature(100)
Feature.

third_party(999)
Third party.

all_p(2147483646)
All facility classes selected.

all_p = AdditionsFacilityClass(2147483646)

driver = AdditionsFacilityClass(10)

feature = AdditionsFacilityClass(100)

none = AdditionsFacilityClass(0)

program = AdditionsFacilityClass(50)

service = AdditionsFacilityClass(30)

third_party = AdditionsFacilityClass(999)

class virtualbox.library.**AdditionsFacilityStatus**(*value*)
Guest Additions facility states.

inactive(0)
Facility is not active.

paused(1)
Facility has been paused.

pre_init(20)
Facility is preparing to initialize.

init(30)
Facility is initializing.

active(50)
Facility is up and running.

terminating(100)
Facility is shutting down.

terminated(101)
Facility successfully shut down.

failed(800)
Facility failed to start.

unknown(999)
Facility status is unknown.

active = AdditionsFacilityStatus(50)

failed = AdditionsFacilityStatus(800)

inactive = AdditionsFacilityStatus(0)

init = AdditionsFacilityStatus(30)

paused = AdditionsFacilityStatus(1)

pre_init = AdditionsFacilityStatus(20)

terminated = AdditionsFacilityStatus(101)

terminating = AdditionsFacilityStatus(100)

unknown = AdditionsFacilityStatus(999)

class virtualbox.library.**AdditionsRunLevelType**(*value*)
Guest Additions run level type.

none(0)
Guest Additions are not loaded.

system(1)
Guest drivers are loaded.

userland(2)
Common components (such as application services) are loaded.

desktop(3)
Per-user desktop components are loaded.

desktop = AdditionsRunLevelType(3)

none = AdditionsRunLevelType(0)

system = AdditionsRunLevelType(1)

userland = AdditionsRunLevelType(2)

class virtualbox.library.**AdditionsUpdateFlag**(*value*)
Guest Additions update flags.

none(0)
No flag set.

wait_for_update_start_only(1)
Starts the regular updating process and waits until the actual Guest Additions update inside the guest was started. This can be necessary due to needed interaction with the guest OS during the installation phase.

none = AdditionsUpdateFlag(0)

wait_for_update_start_only = AdditionsUpdateFlag(1)

class virtualbox.library.**GuestSessionStatus**(*value*)
Guest session status. This enumeration represents possible values of the *IGuestSession.status()* attribute.

undefined(0)
Guest session is in an undefined state.

starting(10)
Guest session is being started.

started(100)
Guest session has been started.

terminating(480)
Guest session is being terminated.

terminated(500)
Guest session terminated normally.

timed_out_killed(512)
Guest session timed out and was killed.

timed_out_abnormally(513)
Guest session timed out and was not killed successfully.

down(600)
Service/OS is stopping, guest session was killed.

error(800)
Something went wrong.

down = GuestSessionStatus(600)

error = GuestSessionStatus(800)

started = GuestSessionStatus(100)

starting = GuestSessionStatus(10)

terminated = GuestSessionStatus(500)

terminating = GuestSessionStatus(480)

timed_out_abnormally = GuestSessionStatus(513)

timed_out_killed = GuestSessionStatus(512)

undefined = GuestSessionStatus(0)

class virtualbox.library.**GuestSessionWaitForFlag**(*value*)
Guest session waiting flags. Multiple flags can be combined.

none(0)
No waiting flags specified. Do not use this.

start(1)
Wait for the guest session being started.

terminate(2)
Wait for the guest session being terminated.

status(4)
Wait for the next guest session status change.

none = GuestSessionWaitForFlag(0)

start = GuestSessionWaitForFlag(1)

status = GuestSessionWaitForFlag(4)

terminate = GuestSessionWaitForFlag(2)

class virtualbox.library.**GuestSessionWaitResult**(*value*)
Guest session waiting results. Depending on the session waiting flags (for more information see *GuestSessionWaitForFlag*) the waiting result can vary based on the session's current status.

To wait for a guest session to terminate after it has been created by *IGuest.create_session()* one would specify GuestSessionWaitResult_Terminate.

none(0)
No result was returned. Not being used.

start(1)
The guest session has been started.

terminate(2)
The guest session has been terminated.

status(3)
The guest session has changed its status. The status then can be retrieved via *IGuestSession.status()* .

error(4)
Error while executing the process.

timeout(5)
The waiting operation timed out. This also will happen when no event has been occurred matching the current waiting flags in a *IGuestSession.wait_for()* call.

wait_flag_not_supported(6)
A waiting flag specified in the *IGuestSession.wait_for()* call is not supported by the guest.

error = GuestSessionWaitResult(4)

none = GuestSessionWaitResult(0)

start = GuestSessionWaitResult(1)

status = GuestSessionWaitResult(3)

terminate = GuestSessionWaitResult(2)

timeout = GuestSessionWaitResult(5)

wait_flag_not_supported = GuestSessionWaitResult(6)

class virtualbox.library.**GuestUserState**(*value*)

State a guest user has been changed to.

unknown(0)
Unknown state. Not being used.

logged_in(1)
A guest user has been successfully logged into the guest OS. This property is not implemented yet!

logged_out(2)
A guest user has been successfully logged out of the guest OS. This property is not implemented yet!

locked(3)
A guest user has locked its account. This might include running a password-protected screensaver in the guest. This property is not implemented yet!

unlocked(4)
A guest user has unlocked its account. This property is not implemented yet!

disabled(5)
A guest user has been disabled by the guest OS. This property is not implemented yet!

idle(6)
A guest user currently is not using the guest OS. Currently only available for Windows guests since Windows 2000 SP2. On Windows guests this function currently only supports reporting contiguous idle times up to 49.7 days per user. The event will be triggered if a guest user is not active for at least 5 seconds. This threshold can be adjusted by either altering VBoxService's command line in the guest to –vminfo-user-idle-threshold , or by setting the per-VM guest property /VirtualBox/GuestAdd/VBoxService/–vminfo-user-idle-threshold with the RDONLYGUEST flag on the host. In both cases VBoxService needs to be restarted in order to get the changes applied.

in_use(7)
A guest user continued using the guest OS after being idle.

created(8)
A guest user has been successfully created. This property is not implemented yet!

deleted(9)
A guest user has been successfully deleted. This property is not implemented yet!

session_changed(10)
To guest OS has changed the session of a user. This property is not implemented yet!

credentials_changed(11)
To guest OS has changed the authentication credentials of a user. This might include changed passwords and authentication types. This property is not implemented yet!

role_changed(12)
To guest OS has changed the role of a user permanently, e.g. granting / denying administrative rights. This property is not implemented yet!

group_added(13)
To guest OS has added a user to a specific user group. This property is not implemented yet!

group_removed(14)
To guest OS has removed a user from a specific user group. This property is not implemented yet!

elevated(15)
To guest OS temporarily has elevated a user to perform a certain task. This property is not implemented yet!

created = GuestUserState(8)

credentials_changed = GuestUserState(11)

deleted = GuestUserState(9)

disabled = GuestUserState(5)

elevated = GuestUserState(15)

group_added = GuestUserState(13)

group_removed = GuestUserState(14)

idle = GuestUserState(6)

in_use = GuestUserState(7)

locked = GuestUserState(3)

logged_in = GuestUserState(1)

logged_out = GuestUserState(2)

role_changed = GuestUserState(12)

session_changed = GuestUserState(10)

unknown = GuestUserState(0)

unlocked = GuestUserState(4)

class virtualbox.library.**FileSeekOrigin**(*value*)
What a file seek (*IFile.seek()*) is relative to.

begin(0)
Seek from the beginning of the file.

current(1)
Seek from the current file position.

end(2)
Seek relative to the end of the file. To seek to the position two bytes from the end of the file, specify -2 as the seek offset.

begin = FileSeekOrigin(0)

current = FileSeekOrigin(1)

end = FileSeekOrigin(2)

class virtualbox.library.**ProcessInputFlag**(*value*)
Guest process input flags.

none(0)
No flag set.

end_of_file(1)
End of file (input) reached.

end_of_file = ProcessInputFlag(1)

none = ProcessInputFlag(0)

class virtualbox.library.**ProcessOutputFlag**(*value*)
Guest process output flags for specifying which type of output to retrieve.

none(0)
No flags set. Get output from stdout.

std_err(1)
Get output from stderr.

none = ProcessOutputFlag(0)

std_err = ProcessOutputFlag(1)

class virtualbox.library.**ProcessWaitForFlag**(*value*)
Process waiting flags. Multiple flags can be combined.

none(0)
No waiting flags specified. Do not use this.

start(1)
Wait for the process being started.

terminate(2)
Wait for the process being terminated.

std_in(4)
Wait for stdin becoming available.

std_out(8)
Wait for data becoming available on stdout.

std_err(16)
Wait for data becoming available on stderr.

none = ProcessWaitForFlag(0)

start = ProcessWaitForFlag(1)

std_err = ProcessWaitForFlag(16)

std_in = ProcessWaitForFlag(4)

std_out = ProcessWaitForFlag(8)

terminate = ProcessWaitForFlag(2)

class virtualbox.library.**ProcessWaitResult**(*value*)
Process waiting results. Depending on the process waiting flags (for more information see *ProcessWaitForFlag*) the waiting result can vary based on the processes' current status.

To wait for a guest process to terminate after it has been created by *IGuestSession.process_create()* or *IGuestSession.process_create_ex()* one would specify ProcessWaitFor_Terminate.

If a guest process has been started with ProcessCreateFlag_WaitForStdOut a client can wait with ProcessWaitResult_StdOut for new data to arrive on stdout; same applies for ProcessCreateFlag_WaitForStdErr and ProcessWaitResult_StdErr.

none(0)
No result was returned. Not being used.

start(1)
The process has been started.

terminate(2)
The process has been terminated.

status(3)
The process has changed its status. The status then can be retrieved via *IProcess.status()* .

error(4)
Error while executing the process.

timeout(5)
The waiting operation timed out. Also use if the guest process has timed out in the guest side (kill attempted).

std_in(6)
The process signalled that stdin became available for writing.

std_out(7)
Data on stdout became available for reading.

std_err(8)
Data on stderr became available for reading.

wait_flag_not_supported(9)
A waiting flag specified in the *IProcess.wait_for()* call is not supported by the guest.

error = ProcessWaitResult(4)

none = ProcessWaitResult(0)

start = ProcessWaitResult(1)

status = ProcessWaitResult(3)

std_err = ProcessWaitResult(8)

std_in = ProcessWaitResult(6)

std_out = ProcessWaitResult(7)

terminate = ProcessWaitResult(2)

timeout = ProcessWaitResult(5)

wait_flag_not_supported = ProcessWaitResult(9)

class virtualbox.library.**FileCopyFlag**(*value*)
File copying flags. Not flags are implemented yet.

none(0)
No flag set.

no_replace(1)
Do not replace the destination file if it exists. This flag is not implemented yet.

follow_links(2)
Follow symbolic links. This flag is not implemented yet.

update(4)
Only copy when the source file is newer than the destination file or when the destination file is missing. This flag is not implemented yet.

follow_links = FileCopyFlag(2)

no_replace = FileCopyFlag(1)

none = FileCopyFlag(0)

update = FileCopyFlag(4)

class virtualbox.library.**FsObjMoveFlags**(*value*)
File moving flags.

none(0)
No flag set.

replace(1)
Replace the destination file, symlink, etc if it exists, however this does not allow replacing any directories.

follow_links(2)
Follow symbolic links in the final components or not (only applied to the given source and target paths, not to anything else).

allow_directory_moves(4)
Allow moving directories accross file system boundraries. Because it is could be a big undertaking, we require extra assurance that we should do it when requested.

allow_directory_moves = FsObjMoveFlags(4)

follow_links = FsObjMoveFlags(2)

none = FsObjMoveFlags(0)

replace = FsObjMoveFlags(1)

class virtualbox.library.**DirectoryCreateFlag**(*value*)
Directory creation flags.

none(0)
No flag set.

parents(1)
No error if existing, make parent directories as needed.

none = DirectoryCreateFlag(0)

parents = DirectoryCreateFlag(1)

class virtualbox.library.**DirectoryCopyFlags**(*value*)
Directory copying flags. Not flags are implemented yet.

none(0)
No flag set.

copy_into_existing(1)
Allow copying into an existing destination directory.

copy_into_existing = DirectoryCopyFlags(1)

none = DirectoryCopyFlags(0)

class virtualbox.library.**DirectoryRemoveRecFlag**(*value*)
Directory recursive removement flags.

WARNING!! THE FLAGS ARE CURRENTLY IGNORED. THE METHOD APPLIES *DirectoryRemoveRecFlag.content_and_dir* REGARDLESS OF THE INPUT.

none(0)
No flag set.

content_and_dir(1)
Delete the content of the directory and the directory itself.

content_only(2)
Only delete the content of the directory, omit the directory it self.

content_and_dir = DirectoryRemoveRecFlag(1)

content_only = DirectoryRemoveRecFlag(2)

none = DirectoryRemoveRecFlag(0)

class virtualbox.library.**FsObjRenameFlag**(*value*)
Flags for use when renaming file system objects (files, directories, symlink, etc), see *IGuestSession.fs_obj_rename()* .

no_replace(0)
Do not replace any destination object.

replace(1)
This will attempt to replace any destination object other except directories. (The default is to fail if the destination exists.)

no_replace = FsObjRenameFlag(0)

replace = FsObjRenameFlag(1)

class virtualbox.library.**ProcessCreateFlag**(*value*)
Guest process execution flags. The values are passed to the guest additions, so its not possible to change (move) or reuse values.here. See EXECUTEPROCESSFLAG_XXX in GuestControlSvc.h.

none(0)
No flag set.

wait_for_process_start_only(1)
Only use the specified timeout value to wait for starting the guest process - the guest process itself then uses an infinite timeout.

ignore_orphaned_processes(2)
Do not report an error when executed processes are still alive when VBoxService or the guest OS is shutting down.

hidden(4)
Do not show the started process according to the guest OS guidelines.

profile(8)
Utilize the user's profile data when exeuting a process. Only available for Windows guests at the moment.

wait_for_std_out(16)
The guest process waits until all data from stdout is read out.

wait_for_std_err(32)
The guest process waits until all data from stderr is read out.

expand_arguments(64)
Expands environment variables in process arguments.

This is not yet implemented and is currently silently ignored. We will document the protocolVersion number for this feature once it appears, so don't use it till then.

unquoted_arguments(128)
Work around for Windows and OS/2 applications not following normal argument quoting and

escaping rules. The arguments are passed to the application without any extra quoting, just a single space between each. Present since VirtualBox 4.3.28 and 5.0 beta 3.

expand_arguments = ProcessCreateFlag(64)

hidden = ProcessCreateFlag(4)

ignore_orphaned_processes = ProcessCreateFlag(2)

none = ProcessCreateFlag(0)

profile = ProcessCreateFlag(8)

unquoted_arguments = ProcessCreateFlag(128)

wait_for_process_start_only = ProcessCreateFlag(1)

wait_for_std_err = ProcessCreateFlag(32)

wait_for_std_out = ProcessCreateFlag(16)

class virtualbox.library.**ProcessPriority**(*value*)
Process priorities.

invalid(0)
Invalid priority, do not use.

default(1)
Default process priority determined by the OS.

default = ProcessPriority(1)

invalid = ProcessPriority(0)

class virtualbox.library.**SymlinkType**(*value*)
Symbolic link types. This is significant when creating links on the Windows platform, ignored elsewhere.

unknown(0)
It is not known what is being targeted.

directory(1)
The link targets a directory.

file_p(2)
The link targets a file (or whatever else except directories).

directory = SymlinkType(1)

file_p = SymlinkType(2)

unknown = SymlinkType(0)

class virtualbox.library.**SymlinkReadFlag**(*value*)
Symbolic link reading flags.

none(0)
No flags set.

no_symlinks(1)
Don't allow symbolic links as part of the path.

no_symlinks = SymlinkReadFlag(1)

none = SymlinkReadFlag(0)

class virtualbox.library.**ProcessStatus**(*value*)
Process execution statuses.

undefined(0)
Process is in an undefined state.

starting(10)
Process is being started.

started(100)
Process has been started.

paused(110)
Process has been paused.

terminating(480)
Process is being terminated.

terminated_normally(500)
Process terminated normally.

terminated_signal(510)
Process terminated via signal.

terminated_abnormally(511)
Process terminated abnormally.

timed_out_killed(512)
Process timed out and was killed.

timed_out_abnormally(513)
Process timed out and was not killed successfully.

down(600)
Service/OS is stopping, process was killed.

error(800)
Something went wrong.

down = ProcessStatus(600)

error = ProcessStatus(800)

paused = ProcessStatus(110)

started = ProcessStatus(100)

starting = ProcessStatus(10)

terminated_abnormally = ProcessStatus(511)

terminated_normally = ProcessStatus(500)

terminated_signal = ProcessStatus(510)

terminating = ProcessStatus(480)

timed_out_abnormally = ProcessStatus(513)

timed_out_killed = ProcessStatus(512)

undefined = ProcessStatus(0)

class virtualbox.library.**ProcessInputStatus**(*value*)
Process input statuses.

undefined(0)
Undefined state.

broken(1)
Input pipe is broken.

available(10)
Input pipe became available for writing.

written(50)
Data has been successfully written.

overflow(100)
Too much input data supplied, data overflow.

available = ProcessInputStatus(10)

broken = ProcessInputStatus(1)

overflow = ProcessInputStatus(100)

undefined = ProcessInputStatus(0)

written = ProcessInputStatus(50)

class virtualbox.library.**PathStyle**(*value*)
The path style of a system. (Values matches the RTPATH_STR_F_STYLE_XXX defines in iprt/path.h!)

dos(1)
DOS-style paths with forward and backward slashes, drive letters and UNC. Known from DOS, OS/2 and Windows.

unix(2)
UNIX-style paths with forward slashes only.

unknown(8)
The path style is not known, most likely because the guest additions aren't active yet.

dos = PathStyle(1)

unix = PathStyle(2)

unknown = PathStyle(8)

class virtualbox.library.**FileAccessMode**(*value*)
File open access mode for use with *IGuestSession.file_open()* and *IGuestSession.file_open_ex()* .

read_only(1)
Open the file only with read access.

write_only(2)
Open the file only with write access.

read_write(3)
Open the file with both read and write access.

append_only(4)
Open the file for appending only, no read or seek access. Not yet implemented.

append_read(5)
Open the file for appending and read. Writes always goes to the end of the file while reads are done at the current or specified file position. Not yet implemented.

append_only = FileAccessMode(4)

append_read = FileAccessMode(5)

read_only = FileAccessMode(1)

read_write = FileAccessMode(3)

write_only = FileAccessMode(2)

class virtualbox.library.**FileOpenAction**(*value*)

What action *IGuestSession.file_open()* and *IGuestSession.file_open_ex()* should take whether the file being opened exists or not.

open_existing(1)
Opens an existing file, fails if no file exists. (Was "oe".)

open_or_create(2)
Opens an existing file, creates a new one if no file exists. (Was "oc".)

create_new(3)
Creates a new file is no file exists, fails if there is a file there already. (Was "ce".)

create_or_replace(4)
Creates a new file, replace any existing file. (Was "ca".)

Currently undefined whether we will inherit mode and ACLs from the existing file or replace them.

open_existing_truncated(5)
Opens and truncate an existing file, fails if no file exists. (Was "ot".)

append_or_create(99)
Opens an existing file and places the file pointer at the end of the file, creates the file if it does not exist. This action implies write access. (Was "oa".)

<!– @todo r=bird: See iprt/file.h, RTFILE_O_APPEND - not an action/disposition! Moving the file pointer to the end, is almost fine, but implying 'write' access isn't. That is something that is exclusively reserved for the opening mode. –> Deprecated. Only here for historical reasons. Do not use!

append_or_create = FileOpenAction(99)

create_new = FileOpenAction(3)

create_or_replace = FileOpenAction(4)

open_existing = FileOpenAction(1)

open_existing_truncated = FileOpenAction(5)

open_or_create = FileOpenAction(2)

class virtualbox.library.**FileSharingMode**(*value*)

File sharing mode for *IGuestSession.file_open_ex()* .

read(1)
Only share read access to the file.

write(2)
Only share write access to the file.

read_write(3)
Share both read and write access to the file, but deny deletion.

delete(4)
Only share delete access, denying read and write.

read_delete(5)
Share read and delete access to the file, denying writing.

write_delete(6)
Share write and delete access to the file, denying reading.

all_p(7)
Share all access, i.e. read, write and delete, to the file.

all_p = FileSharingMode(7)

delete = FileSharingMode(4)

read = FileSharingMode(1)

read_delete = FileSharingMode(5)

read_write = FileSharingMode(3)

write = FileSharingMode(2)

write_delete = FileSharingMode(6)

class virtualbox.library.**FileOpenExFlags**(*value*)
Open flags for *IGuestSession.file_open_ex()* .

none(0)
No flag set.

none = FileOpenExFlags(0)

class virtualbox.library.**FileStatus**(*value*)
File statuses.

undefined(0)
File is in an undefined state.

opening(10)
Guest file is opening.

open_p(100)
Guest file has been successfully opened.

closing(150)
Guest file closing.

closed(200)
Guest file has been closed.

down(600)
Service/OS is stopping, guest file was closed.

error(800)
Something went wrong.

closed = FileStatus(200)

closing = FileStatus(150)

down = FileStatus(600)

error = FileStatus(800)

open_p = FileStatus(100)

opening = FileStatus(10)

undefined = FileStatus(0)

class virtualbox.library.**FsObjType**(*value*)
File system object (file) types.

unknown(1)
Used either if the object has type that is not in this enum, or if the type has not yet been determined or set.

fifo(2)
FIFO or named pipe, depending on the platform/terminology.

dev_char(3)
Character device.

directory(4)
Directory.

dev_block(5)
Block device.

file_p(6)
Regular file.

symlink(7)
Symbolic link.

socket(8)
Socket.

white_out(9)
A white-out file. Found in union mounts where it is used for hiding files after deletion, I think.

dev_block = FsObjType(5)

dev_char = FsObjType(3)

directory = FsObjType(4)

fifo = FsObjType(2)

file_p = FsObjType(6)

socket = FsObjType(8)

symlink = FsObjType(7)

unknown = FsObjType(1)

white_out = FsObjType(9)

class virtualbox.library.**DnDAction**(*value*)
Possible actions of a drag'n drop operation.

ignore(0)
Do nothing.

copy(1)
Copy the item to the target.

move(2)
Move the item to the target.

link(3)
Link the item from within the target.

copy = DnDAction(1)

ignore = DnDAction(0)

link = DnDAction(3)

move = DnDAction(2)

class virtualbox.library.**DirectoryOpenFlag**(*value*)
Directory open flags.

none(0)
No flag set.

no_symlinks(1)
Don't allow symbolic links as part of the path.

no_symlinks = DirectoryOpenFlag(1)

none = DirectoryOpenFlag(0)

class virtualbox.library.**MediumState**(*value*)
Virtual medium state. *IMedium*

not_created(0)
Associated medium storage does not exist (either was not created yet or was deleted).

created(1)
Associated storage exists and accessible; this gets set if the accessibility check performed by *IMedium.refresh_state()* was successful.

locked_read(2)
Medium is locked for reading (see *IMedium.lock_read()*), no data modification is possible.

locked_write(3)
Medium is locked for writing (see *IMedium.lock_write()*), no concurrent data reading or modification is possible.

inaccessible(4)
Medium accessibility check (see *IMedium.refresh_state()*) has not yet been performed, or else, associated medium storage is not accessible. In the first case, *IMedium.last_access_error()* is empty, in the second case, it describes the error that occurred.

creating(5)
Associated medium storage is being created.

deleting(6)
Associated medium storage is being deleted.

created = MediumState(1)

creating = MediumState(5)

deleting = MediumState(6)

inaccessible = MediumState(4)

locked_read = MediumState(2)

locked_write = MediumState(3)

not_created = MediumState(0)

class virtualbox.library.**MediumType**(*value*)

Virtual medium type. For each *IMedium* , this defines how the medium is attached to a virtual machine (see *IMediumAttachment*) and what happens when a snapshot (see *ISnapshot*) is taken of a virtual machine which has the medium attached. At the moment DVD and floppy media are always of type "writethrough".

normal(0)
Normal medium (attached directly or indirectly, preserved when taking snapshots).

immutable(1)
Immutable medium (attached indirectly, changes are wiped out the next time the virtual machine is started).

writethrough(2)
Write through medium (attached directly, ignored when taking snapshots).

shareable(3)
Allow using this medium concurrently by several machines. Present since VirtualBox 3.2.0, and accepted since 3.2.8.

readonly(4)
A readonly medium, which can of course be used by several machines. Present and accepted since VirtualBox 4.0.

multi_attach(5)
A medium which is indirectly attached, so that one base medium can be used for several VMs which have their own differencing medium to store their modifications. In some sense a variant of Immutable with unset AutoReset flag in each differencing medium. Present and accepted since VirtualBox 4.0.

immutable = MediumType(1)

multi_attach = MediumType(5)

normal = MediumType(0)

readonly = MediumType(4)

shareable = MediumType(3)

writethrough = MediumType(2)

class virtualbox.library.**MediumVariant**(*value*)

Virtual medium image variant. More than one flag may be set. *IMedium*

standard(0)
No particular variant requested, results in using the backend default.

vmdk_split2_g(1)
VMDK image split in chunks of less than 2GByte.

vmdk_raw_disk(2)
VMDK image representing a raw disk.

vmdk_stream_optimized(4)
VMDK streamOptimized image. Special import/export format which is read-only/append-only.

vmdk_esx(8)
VMDK format variant used on ESX products.

vdi_zero_expand(256)
Fill new blocks with zeroes while expanding image file.

fixed(65536)
Fixed image. Only allowed for base images.

diff(131072)
Differencing image. Only allowed for child images.

no_create_dir(1073741824)
Special flag which suppresses automatic creation of the subdirectory. Only used when passing the medium variant as an input parameter.

diff = MediumVariant(131072)

fixed = MediumVariant(65536)

no_create_dir = MediumVariant(1073741824)

standard = MediumVariant(0)

vdi_zero_expand = MediumVariant(256)

vmdk_esx = MediumVariant(8)

vmdk_raw_disk = MediumVariant(2)

vmdk_split2_g = MediumVariant(1)

vmdk_stream_optimized = MediumVariant(4)

class virtualbox.library.**DataType**(*value*)

int32(0)

int8(1)

string(2)

class virtualbox.library.**DataFlags**(*value*)

none(0)

mandatory(1)

expert(2)

array(4)

flag_mask(7)

class virtualbox.library.**MediumFormatCapabilities**(*value*)
Medium format capability flags.

uuid(1)
Supports UUIDs as expected by VirtualBox code.

create_fixed(2)
Supports creating fixed size images, allocating all space instantly.

create_dynamic(4)
Supports creating dynamically growing images, allocating space on demand.

create_split2_g(8)
Supports creating images split in chunks of a bit less than 2 GBytes.

differencing(16)
Supports being used as a format for differencing media (see *IMedium.create_diff_storage()*).

asynchronous(32)
Supports asynchronous I/O operations for at least some configurations.

file_p(64)
The format backend operates on files (the *IMedium.location()* attribute of the medium specifies a file used to store medium data; for a list of supported file extensions see *IMediumFormat.describe_file_extensions()*).

properties(128)
The format backend uses the property interface to configure the storage location and properties (the *IMediumFormat.describe_properties()* method is used to get access to properties supported by the given medium format).

tcp_networking(256)
The format backend uses the TCP networking interface for network access.

vfs(512)
The format backend supports virtual filesystem functionality.

discard(1024)
The format backend supports discarding blocks.

preferred(2048)
Indicates that this is a frequently used format backend.

capability_mask(4095)

asynchronous = MediumFormatCapabilities(32)

create_dynamic = MediumFormatCapabilities(4)

create_fixed = MediumFormatCapabilities(2)

create_split2_g = MediumFormatCapabilities(8)

differencing = MediumFormatCapabilities(16)

discard = MediumFormatCapabilities(1024)

file_p = MediumFormatCapabilities(64)

preferred = MediumFormatCapabilities(2048)

properties = MediumFormatCapabilities(128)

tcp_networking = MediumFormatCapabilities(256)

uuid = MediumFormatCapabilities(1)

vfs = MediumFormatCapabilities(512)

class virtualbox.library.**KeyboardLED**(*value*)
Keyboard LED indicators.

num_lock(1)

caps_lock(2)

scroll_lock(4)

class virtualbox.library.**MouseButtonState**(*value*)
Mouse button state.

left_button(1)

right_button(2)

middle_button(4)

wheel_up(8)

wheel_down(16)

x_button1(32)

x_button2(64)

mouse_state_mask(127)

class virtualbox.library.**TouchContactState**(*value*)
Touch event contact state.

none(0)
The touch has finished.

in_contact(1)
Whether the touch is really touching the device.

in_range(2)
Whether the touch is close enough to the device to be detected.

contact_state_mask(3)

in_contact = TouchContactState(1)

in_range = TouchContactState(2)

none = TouchContactState(0)

class virtualbox.library.**FramebufferCapabilities**(*value*)
Framebuffer capability flags.

update_image(1)
Requires NotifyUpdateImage. NotifyUpdate must not be called.

vhwa(2)
Supports VHWA interface. If set, then IFramebuffer::processVHWACommand can be called.

visible_region(4)
Supports visible region. If set, then IFramebuffer::setVisibleRegion can be called.

update_image = FramebufferCapabilities(1)

vhwa = FramebufferCapabilities(2)

visible_region = FramebufferCapabilities(4)

class virtualbox.library.**GuestMonitorStatus**(*value*)
The current status of the guest display.

disabled(0)
The guest monitor is disabled in the guest.

enabled(1)
The guest monitor is enabled in the guest.

blank(2)
The guest monitor is enabled in the guest but should display nothing.

blank = GuestMonitorStatus(2)

disabled = GuestMonitorStatus(0)

enabled = GuestMonitorStatus(1)

class virtualbox.library.**ScreenLayoutMode**(*value*)
How IDisplay::setScreenLayout method should work.

apply_p(0)
If the guest is already at desired mode then the API might avoid setting the mode.

reset(1)
Always set the new mode even if the guest is already at desired mode.

apply_p = ScreenLayoutMode(0)

reset = ScreenLayoutMode(1)

class virtualbox.library.**NetworkAttachmentType**(*value*)
Network attachment type.

null(0)
Null value, also means "not attached".

nat(1)

bridged(2)

internal(3)

host_only(4)

generic(5)

nat_network(6)

null = NetworkAttachmentType(0)

class virtualbox.library.**NetworkAdapterType**(*value*)
Network adapter type.

null(0)
Null value (never used by the API).

am79_c970_a(1)
AMD PCNet-PCI II network card (Am79C970A).

am79_c973(2)
AMD PCNet-FAST III network card (Am79C973).

i82540_em(3)
Intel PRO/1000 MT Desktop network card (82540EM).

i82543_gc(4)
Intel PRO/1000 T Server network card (82543GC).

i82545_em(5)
Intel PRO/1000 MT Server network card (82545EM).

virtio(6)
Virtio network device.

am79_c970_a = NetworkAdapterType(1)

am79_c973 = NetworkAdapterType(2)

i82540_em = NetworkAdapterType(3)

i82543_gc = NetworkAdapterType(4)

i82545_em = NetworkAdapterType(5)

null = NetworkAdapterType(0)

virtio = NetworkAdapterType(6)

class virtualbox.library.**NetworkAdapterPromiscModePolicy**(*value*)
The promiscuous mode policy of an interface.

deny(1)
Deny promiscuous mode requests.

allow_network(2)
Allow promiscuous mode, but restrict the scope it to the internal network so that it only applies to other VMs.

allow_all(3)
Allow promiscuous mode, include unrelated traffic going over the wire and internally on the host.

allow_all = NetworkAdapterPromiscModePolicy(3)

allow_network = NetworkAdapterPromiscModePolicy(2)

deny = NetworkAdapterPromiscModePolicy(1)

class virtualbox.library.**PortMode**(*value*)
The PortMode enumeration represents possible communication modes for the virtual serial port device.

disconnected(0)
Virtual device is not attached to any real host device.

host_pipe(1)
Virtual device is attached to a host pipe.

host_device(2)
Virtual device is attached to a host device.

raw_file(3)
Virtual device is attached to a raw file.

tcp(4)
Virtual device is attached to a TCP socket.

disconnected = PortMode(0)

host_device = PortMode(2)

host_pipe = PortMode(1)

raw_file = PortMode(3)

tcp = PortMode(4)

class virtualbox.library.**USBControllerType**(*value*)
The USB controller type. *IUSBController.type_p()* .

null(0)
@c null value. Never used by the API.

ohci(1)

ehci(2)

xhci(3)

last(4)
Last element (invalid). Used for parameter checks.

last = USBControllerType(4)

null = USBControllerType(0)

class virtualbox.library.**USBConnectionSpeed**(*value*)
USB device/port speed state. This enumeration represents speeds at which a USB device can communicate with the host.

The speed is a function of both the device itself and the port which it is attached to, including hubs and cables in the path.

Due to differences in USB stack implementations on various hosts, the reported speed may not exactly match the actual speed.

IHostUSBDevice

null(0)
@c null value. Never returned by the API.

low(1)
Low speed, 1.5 Mbps.

full(2)
Full speed, 12 Mbps.

high(3)
High speed, 480 Mbps.

super_p(4)
SuperSpeed, 5 Gbps.

super_plus(5)
SuperSpeedPlus, 10 Gbps.

full = USBConnectionSpeed(2)

high = USBConnectionSpeed(3)

low = USBConnectionSpeed(1)

null = USBConnectionSpeed(0)

super_p = USBConnectionSpeed(4)

super_plus = USBConnectionSpeed(5)

class virtualbox.library.**USBDeviceState**(*value*)
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.usb_device_filters()`) are activated. They can either ignore the device, or put it to USBDeviceState_Held state, or do nothing. Unless the device is ignored by global filters, filters of all currently running guests (*IUSBDeviceFilters.device_filters()*) are activated that can put it to USBDeviceState_Captured state.

If the device was ignored by global filters, or didn't match any filters at all (including guest ones), it is handled by the host in a normal way. In this case, the device state is determined by the host and

can be one of USBDeviceState_Unavailable, USBDeviceState_Busy or USBDeviceState_Available, depending on the current device usage.

Besides auto-capturing based on filters, the device can be manually captured by guests (*IConsole.attach_usb_device()*) if its state is USBDeviceState_Busy, USBDeviceState_Available or USBDeviceState_Held.

Due to differences in USB stack implementations in Linux and Win32, states USBDeviceState_Busy and USBDeviceState_Unavailable are applicable only to the Linux version of the product. This also means that (*IConsole.attach_usb_device()*) can only succeed on Win32 if the device state is USBDeviceState_Held.

IHostUSBDevice, *IHostUSBDeviceFilter*

not_supported(0)
Not supported by the VirtualBox server, not available to guests.

unavailable(1)
Being used by the host computer exclusively, not available to guests.

busy(2)
Being used by the host computer, potentially available to guests.

available(3)
Not used by the host computer, available to guests (the host computer can also start using the device at any time).

held(4)
Held by the VirtualBox server (ignored by the host computer), available to guests.

captured(5)
Captured by one of the guest computers, not available to anybody else.

available = USBDeviceState(3)

busy = USBDeviceState(2)

captured = USBDeviceState(5)

held = USBDeviceState(4)

not_supported = USBDeviceState(0)

unavailable = USBDeviceState(1)

class virtualbox.library.**USBDeviceFilterAction**(*value*)
Actions for host USB device filters. *IHostUSBDeviceFilter*, *USBDeviceState*

null(0)
Null value (never used by the API).

ignore(1)
Ignore the matched USB device.

hold(2)
Hold the matched USB device.

hold = USBDeviceFilterAction(2)

ignore = USBDeviceFilterAction(1)

null = USBDeviceFilterAction(0)

class virtualbox.library.**AudioDriverType**(*value*)
Host audio driver type.

null(0)
Null value, also means "dummy audio driver".

win_mm(1)
Windows multimedia (Windows hosts only, not supported at the moment).

oss(2)
Open Sound System (Linux / Unix hosts only).

alsa(3)
Advanced Linux Sound Architecture (Linux hosts only).

direct_sound(4)
DirectSound (Windows hosts only).

core_audio(5)
CoreAudio (Mac hosts only).

mmpm(6)
Reserved for historical reasons.

pulse(7)
PulseAudio (Linux hosts only).

sol_audio(8)
Solaris audio (Solaris hosts only, not supported at the moment).

alsa = AudioDriverType(3)

core_audio = AudioDriverType(5)

direct_sound = AudioDriverType(4)

mmpm = AudioDriverType(6)

null = AudioDriverType(0)

oss = AudioDriverType(2)

pulse = AudioDriverType(7)

sol_audio = AudioDriverType(8)

win_mm = AudioDriverType(1)

class virtualbox.library.**AudioControllerType**(*value*)
Virtual audio controller type.

ac97(0)

sb16(1)

hda(2)

class virtualbox.library.**AudioCodecType**(*value*)
The exact variant of audio codec hardware presented to the guest; see *IAudioAdapter.audio_codec()*.

null(0)
@c null value. Never used by the API.

sb16(1)
SB16; this is the only option for the SB16 device.

stac9700(2)
A STAC9700 AC'97 codec.

ad1980(3)
An AD1980 AC'97 codec. Recommended for Linux guests.

stac9221(4)
A STAC9221 HDA codec.

ad1980 = AudioCodecType(3)

null = AudioCodecType(0)

sb16 = AudioCodecType(1)

stac9221 = AudioCodecType(4)

stac9700 = AudioCodecType(2)

class virtualbox.library.**AuthType**(*value*)
VirtualBox authentication type.

null(0)
Null value, also means "no authentication".

external(1)

guest(2)

null = AuthType(0)

class virtualbox.library.**Reason**(*value*)
Internal event reason type.

unspecified(0)
Null value, means "no known reason".

host_suspend(1)
Host is being suspended (power management event).

host_resume(2)
Host is being resumed (power management event).

host_battery_low(3)
Host is running low on battery (power management event).

snapshot(4)
A snapshot of the VM is being taken.

host_battery_low = Reason(3)

host_resume = Reason(2)

host_suspend = Reason(1)

snapshot = Reason(4)

unspecified = Reason(0)

class virtualbox.library.**StorageBus**(*value*)
The bus type of the storage controller (IDE, SATA, SCSI, SAS or Floppy); see *IStorageController.bus()* .

null(0)
@c null value. Never used by the API.

ide(1)

sata(2)

scsi(3)

floppy(4)

sas(5)

usb(6)

pc_ie(7)

null = StorageBus(0)

class virtualbox.library.**StorageControllerType**(*value*)

The exact variant of storage controller hardware presented to the guest; see *IStorageController.controller_type()*.

null(0)
@c null value. Never used by the API.

lsi_logic(1)
A SCSI controller of the LsiLogic variant.

bus_logic(2)
A SCSI controller of the BusLogic variant.

intel_ahci(3)
An Intel AHCI SATA controller; this is the only variant for SATA.

piix3(4)
An IDE controller of the PIIX3 variant.

piix4(5)
An IDE controller of the PIIX4 variant.

ich6(6)
An IDE controller of the ICH6 variant.

i82078(7)
A floppy disk controller; this is the only variant for floppy drives.

lsi_logic_sas(8)
A variant of the LsiLogic controller using SAS.

usb(9)
Special USB based storage controller.

nv_me(10)
An NVMe storage controller.

bus_logic = StorageControllerType(2)

i82078 = StorageControllerType(7)

ich6 = StorageControllerType(6)

intel_ahci = StorageControllerType(3)

lsi_logic = StorageControllerType(1)

lsi_logic_sas = StorageControllerType(8)

null = StorageControllerType(0)

nv_me = StorageControllerType(10)

piix3 = StorageControllerType(4)

piix4 = StorageControllerType(5)

usb = StorageControllerType(9)

class virtualbox.library.**ChipsetType**(*value*)
Type of emulated chipset (mostly southbridge).

null(0)
@c null value. Never used by the API.

piix3(1)
A PIIX3 (PCI IDE ISA Xcelerator) chipset.

ich9(2)
A ICH9 (I/O Controller Hub) chipset.

ich9 = ChipsetType(2)

null = ChipsetType(0)

piix3 = ChipsetType(1)

class virtualbox.library.**NATAliasMode**(*value*)

alias_log(1)

alias_proxy_only(2)

alias_use_same_ports(4)

class virtualbox.library.**NATProtocol**(*value*)
Protocol definitions used with NAT port-forwarding rules.

udp(0)
Port-forwarding uses UDP protocol.

tcp(1)
Port-forwarding uses TCP protocol.

tcp = NATProtocol(1)

udp = NATProtocol(0)

class virtualbox.library.**BandwidthGroupType**(*value*)
Type of a bandwidth control group.

null(0)
Null type, must be first.

disk(1)
The bandwidth group controls disk I/O.

network(2)
The bandwidth group controls network I/O.

disk = BandwidthGroupType(1)

network = BandwidthGroupType(2)

null = BandwidthGroupType(0)

class virtualbox.library.**VBoxEventType**(*value*)
Type of an event. See *IEvent* for an introduction to VirtualBox event handling.

invalid(0)
Invalid event, must be first.

any_p(1)
Wildcard for all events. Events of this type are never delivered, and only used in *IEventSource.register_listener()* call to simplify registration.

vetoable(2)
Wildcard for all vetoable events. Events of this type are never delivered, and only used in *IEventSource.register_listener()* call to simplify registration.

machine_event(3)
Wildcard for all machine events. Events of this type are never delivered, and only used in *IEventSource.register_listener()* call to simplify registration.

snapshot_event(4)
Wildcard for all snapshot events. Events of this type are never delivered, and only used in *IEventSource.register_listener()* call to simplify registration.

input_event(5)
Wildcard for all input device (keyboard, mouse) events. Events of this type are never delivered, and only used in *IEventSource.register_listener()* call to simplify registration.

last_wildcard(31)
Last wildcard.

on_machine_state_changed(32)
See *IMachineStateChangedEvent* IMachineStateChangedEvent.

on_machine_data_changed(33)
See *IMachineDataChangedEvent* IMachineDataChangedEvent.

on_extra_data_changed(34)
See *IExtraDataChangedEvent* IExtraDataChangedEvent.

on_extra_data_can_change(35)
See *IExtraDataCanChangeEvent* IExtraDataCanChangeEvent.

on_medium_registered(36)
See *IMediumRegisteredEvent* IMediumRegisteredEvent.

on_machine_registered(37)
See *IMachineRegisteredEvent* IMachineRegisteredEvent.

on_session_state_changed(38)
See *ISessionStateChangedEvent* ISessionStateChangedEvent.

on_snapshot_taken(39)
See *ISnapshotTakenEvent* ISnapshotTakenEvent.

on_snapshot_deleted(40)
See *ISnapshotDeletedEvent* ISnapshotDeletedEvent.

on_snapshot_changed(41)
See *ISnapshotChangedEvent* ISnapshotChangedEvent.

on_guest_property_changed(42)
See *IGuestPropertyChangedEvent* IGuestPropertyChangedEvent.

on_mouse_pointer_shape_changed(43)
See *IMousePointerShapeChangedEvent* IMousePointerShapeChangedEvent.

on_mouse_capability_changed(44)
See *IMouseCapabilityChangedEvent* IMouseCapabilityChangedEvent.

on_keyboard_leds_changed(45)
See *IKeyboardLedsChangedEvent* IKeyboardLedsChangedEvent.

on_state_changed(46)
See *IStateChangedEvent* IStateChangedEvent.

on_additions_state_changed(47)
See *IAdditionsStateChangedEvent* IAdditionsStateChangedEvent.

on_network_adapter_changed(48)
See *INetworkAdapterChangedEvent* INetworkAdapterChangedEvent.

on_serial_port_changed(49)
See *ISerialPortChangedEvent* ISerialPortChangedEvent.

on_parallel_port_changed(50)
See *IParallelPortChangedEvent* IParallelPortChangedEvent.

on_storage_controller_changed(51)
See *IStorageControllerChangedEvent* IStorageControllerChangedEvent.

on_medium_changed(52)
See *IMediumChangedEvent* IMediumChangedEvent.

on_vrde_server_changed(53)
See *IVRDEServerChangedEvent* IVRDEServerChangedEvent.

on_usb_controller_changed(54)
See *IUSBControllerChangedEvent* IUSBControllerChangedEvent.

on_usb_device_state_changed(55)
See *IUSBDeviceStateChangedEvent* IUSBDeviceStateChangedEvent.

on_shared_folder_changed(56)
See *ISharedFolderChangedEvent* ISharedFolderChangedEvent.

on_runtime_error(57)
See *IRuntimeErrorEvent* IRuntimeErrorEvent.

on_can_show_window(58)
See *ICanShowWindowEvent* ICanShowWindowEvent.

on_show_window(59)
See *IShowWindowEvent* IShowWindowEvent.

on_cpu_changed(60)
See *ICPUChangedEvent* ICPUChangedEvent.

on_vrde_server_info_changed(61)
See *IVRDEServerInfoChangedEvent* IVRDEServerInfoChangedEvent.

on_event_source_changed(62)
See *IEventSourceChangedEvent* IEventSourceChangedEvent.

on_cpu_execution_cap_changed(63)
See *ICPUExecutionCapChangedEvent* ICPUExecutionCapChangedEvent.

on_guest_keyboard(64)
See *IGuestKeyboardEvent* IGuestKeyboardEvent.

on_guest_mouse(65)
See *IGuestMouseEvent* IGuestMouseEvent.

on_nat_redirect(66)
See *INATRedirectEvent* INATRedirectEvent.

on_host_pci_device_plug(67)
See *IHostPCIDevicePlugEvent* IHostPCIDevicePlugEvent.

on_v_box_svc_availability_changed(68)
See *IVBoxSVCAvailabilityChangedEvent* IVBoxSVCAvailablityChangedEvent.

on_bandwidth_group_changed(69)
See *IBandwidthGroupChangedEvent* IBandwidthGroupChangedEvent.

on_guest_monitor_changed(70)
See *IGuestMonitorChangedEvent* IGuestMonitorChangedEvent.

on_storage_device_changed(71)
See *IStorageDeviceChangedEvent* IStorageDeviceChangedEvent.

on_clipboard_mode_changed(72)
See *IClipboardModeChangedEvent* IClipboardModeChangedEvent.

on_dn_d_mode_changed(73)
See *IDnDModeChangedEvent* IDnDModeChangedEvent.

on_nat_network_changed(74)
See `INATNetworkChangedEvent` INATNetworkChangedEvent.

on_nat_network_start_stop(75)
See *INATNetworkStartStopEvent* INATNetworkStartStopEvent.

on_nat_network_alter(76)
See `INATNetworkAlterEvent` INATNetworkAlterEvent.

on_nat_network_creation_deletion(77)
See `INATNetworkCreationDeletionEvent` INATNetworkCreationDeletionEvent.

on_nat_network_setting(78)
See `INATNetworkSettingEvent` INATNetworkSettingEvent.

on_nat_network_port_forward(79)
See `INATNetworkPortForwardEvent` INATNetworkPortForwardEvent.

on_guest_session_state_changed(80)
See *IGuestSessionStateChangedEvent* IGuestSessionStateChangedEvent.

on_guest_session_registered(81)
See *IGuestSessionRegisteredEvent* IGuestSessionRegisteredEvent.

on_guest_process_registered(82)
See *IGuestProcessRegisteredEvent* IGuestProcessRegisteredEvent.

on_guest_process_state_changed(83)
See *IGuestProcessStateChangedEvent* IGuestProcessStateChangedEvent.

on_guest_process_input_notify(84)
See *IGuestProcessInputNotifyEvent* IGuestProcessInputNotifyEvent.

on_guest_process_output(85)
See *IGuestProcessOutputEvent* IGuestProcessOutputEvent.

on_guest_file_registered(86)
See *IGuestFileRegisteredEvent* IGuestFileRegisteredEvent.

on_guest_file_state_changed(87)
See *IGuestFileStateChangedEvent* IGuestFileStateChangedEvent.

on_guest_file_offset_changed(88)
See *IGuestFileOffsetChangedEvent* IGuestFileOffsetChangedEvent.

on_guest_file_read(89)
See *IGuestFileReadEvent* IGuestFileReadEvent.

For performance reasons this is a separate event to not unnecessarily overflow the event queue.

on_guest_file_write(90)
See *IGuestFileWriteEvent* IGuestFileWriteEvent.

For performance reasons this is a separate event to not unnecessarily overflow the event queue.

on_video_capture_changed(91)
See *IVideoCaptureChangedEvent* IVideoCapturedChangeEvent.

on_guest_user_state_changed(92)
See *IGuestUserStateChangedEvent* IGuestUserStateChangedEvent.

on_guest_multi_touch(93)
See *IGuestMouseEvent* IGuestMouseEvent.

on_host_name_resolution_configuration_change(94)
See `IHostNameResolutionConfigurationChangeEvent` IHostNameResolutionConfigurationChangeEvent.

on_snapshot_restored(95)
See *ISnapshotRestoredEvent* ISnapshotRestoredEvent.

on_medium_config_changed(96)
See *IMediumConfigChangedEvent* IMediumConfigChangedEvent.

last(97)
Must be last event, used for iterations and structures relying on numerical event values.

any_p = VBoxEventType(1)

input_event = VBoxEventType(5)

invalid = VBoxEventType(0)

last = VBoxEventType(97)

last_wildcard = VBoxEventType(31)

machine_event = VBoxEventType(3)

on_additions_state_changed = VBoxEventType(47)

on_bandwidth_group_changed = VBoxEventType(69)

on_can_show_window = VBoxEventType(58)

on_clipboard_mode_changed = VBoxEventType(72)

on_cpu_changed = VBoxEventType(60)

on_cpu_execution_cap_changed = VBoxEventType(63)

on_dn_d_mode_changed = VBoxEventType(73)

on_event_source_changed = VBoxEventType(62)

on_extra_data_can_change = VBoxEventType(35)

```
on_extra_data_changed = VBoxEventType(34)
on_guest_file_offset_changed = VBoxEventType(88)
on_guest_file_read = VBoxEventType(89)
on_guest_file_registered = VBoxEventType(86)
on_guest_file_state_changed = VBoxEventType(87)
on_guest_file_write = VBoxEventType(90)
on_guest_keyboard = VBoxEventType(64)
on_guest_monitor_changed = VBoxEventType(70)
on_guest_mouse = VBoxEventType(65)
on_guest_multi_touch = VBoxEventType(93)
on_guest_process_input_notify = VBoxEventType(84)
on_guest_process_output = VBoxEventType(85)
on_guest_process_registered = VBoxEventType(82)
on_guest_process_state_changed = VBoxEventType(83)
on_guest_property_changed = VBoxEventType(42)
on_guest_session_registered = VBoxEventType(81)
on_guest_session_state_changed = VBoxEventType(80)
on_guest_user_state_changed = VBoxEventType(92)
on_host_name_resolution_configuration_change = VBoxEventType(94)
on_host_pci_device_plug = VBoxEventType(67)
on_keyboard_leds_changed = VBoxEventType(45)
on_machine_data_changed = VBoxEventType(33)
on_machine_registered = VBoxEventType(37)
on_machine_state_changed = VBoxEventType(32)
on_medium_changed = VBoxEventType(52)
on_medium_config_changed = VBoxEventType(96)
on_medium_registered = VBoxEventType(36)
on_mouse_capability_changed = VBoxEventType(44)
on_mouse_pointer_shape_changed = VBoxEventType(43)
on_nat_network_alter = VBoxEventType(76)
on_nat_network_changed = VBoxEventType(74)
on_nat_network_creation_deletion = VBoxEventType(77)
on_nat_network_port_forward = VBoxEventType(79)
on_nat_network_setting = VBoxEventType(78)
on_nat_network_start_stop = VBoxEventType(75)
on_nat_redirect = VBoxEventType(66)
```

on_network_adapter_changed = VBoxEventType(48)

on_parallel_port_changed = VBoxEventType(50)

on_runtime_error = VBoxEventType(57)

on_serial_port_changed = VBoxEventType(49)

on_session_state_changed = VBoxEventType(38)

on_shared_folder_changed = VBoxEventType(56)

on_show_window = VBoxEventType(59)

on_snapshot_changed = VBoxEventType(41)

on_snapshot_deleted = VBoxEventType(40)

on_snapshot_restored = VBoxEventType(95)

on_snapshot_taken = VBoxEventType(39)

on_state_changed = VBoxEventType(46)

on_storage_controller_changed = VBoxEventType(51)

on_storage_device_changed = VBoxEventType(71)

on_usb_controller_changed = VBoxEventType(54)

on_usb_device_state_changed = VBoxEventType(55)

on_v_box_svc_availability_changed = VBoxEventType(68)

on_video_capture_changed = VBoxEventType(91)

on_vrde_server_changed = VBoxEventType(53)

on_vrde_server_info_changed = VBoxEventType(61)

snapshot_event = VBoxEventType(4)

vetoable = VBoxEventType(2)

class virtualbox.library.**GuestMouseEventMode**(*value*)

The mode (relative, absolute, multi-touch) of a pointer event.

@todo A clear pattern seems to be emerging that we should usually have multiple input devices active for different types of reporting, so we should really have different event types for relative (including wheel), absolute (not including wheel) and multi-touch events.

relative(0)

Relative event.

absolute(1)

Absolute event.

absolute = GuestMouseEventMode(1)

relative = GuestMouseEventMode(0)

class virtualbox.library.**GuestMonitorChangedEventType**(*value*)

How the guest monitor has been changed.

enabled(0)

The guest monitor has been enabled by the guest.

disabled(1)

The guest monitor has been disabled by the guest.

new_origin(2)
The guest monitor origin has changed in the guest.

disabled = GuestMonitorChangedEventType(1)

enabled = GuestMonitorChangedEventType(0)

new_origin = GuestMonitorChangedEventType(2)

class virtualbox.library.**IVirtualBoxErrorInfo**(*interface=None*)
The IVirtualBoxErrorInfo interface represents extended error information.

Extended error information can be set by VirtualBox components after unsuccessful or partially successful method invocation. This information can be retrieved by the calling party as an IVirtualBoxErrorInfo object and then shown to the client in addition to the plain 32-bit result code.

In MS COM, this interface extends the IErrorInfo interface, in XPCOM, it extends the nsIException interface. In both cases, it provides a set of common attributes to retrieve error information.

Sometimes invocation of some component's method may involve methods of other components that may also fail (independently of this method's failure), or a series of non-fatal errors may precede a fatal error that causes method failure. In cases like that, it may be desirable to preserve information about all errors happened during method invocation and deliver it to the caller. The *next_p()* attribute is intended specifically for this purpose and allows to represent a chain of errors through a single IVirtualBoxErrorInfo object set after method invocation.

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).

result_code
Get int value for 'resultCode' Result code of the error. Usually, it will be the same as the result code returned by the method that provided this error information, but not always. For example, on Win32, CoCreateInstance() will most likely return E_NOINTERFACE upon unsuccessful component instantiation attempt, but not the value the component factory returned. Value is typed 'long', not 'result', to make interface usable from scripting languages.

In MS COM, there is no equivalent. In XPCOM, it is the same as nsIException::result.

result_detail
Get int value for 'resultDetail' Optional result data of this error. This will vary depending on the actual error usage. By default this attribute is not being used.

interface_id
Get str value for 'interfaceID' UUID of the interface that defined the error.

In MS COM, it is the same as IErrorInfo::GetGUID, except for the data type. In XPCOM, there is no equivalent.

component
Get str value for 'component' Name of the component that generated the error.

In MS COM, it is the same as IErrorInfo::GetSource. In XPCOM, there is no equivalent.

text
Get str value for 'text' Text description of the error.

In MS COM, it is the same as IErrorInfo::GetDescription. In XPCOM, it is the same as nsIException::message.

next_p
Get IVirtualBoxErrorInfo value for 'next' Next error object if there is any, or @c null otherwise.

In MS COM, there is no equivalent. In XPCOM, it is the same as nsIException::inner.

class virtualbox.library.**INATNetwork**(*interface=None*)

TBD: the idea, technically we can start any number of the NAT networks, but we should expect that at some point we will get collisions because of port-forwanding rules. so perhaps we should support only single instance of NAT network.

network_name

Get or set str value for 'networkName' TBD: the idea, technically we can start any number of the NAT networks, but we should expect that at some point we will get collisions because of port-forwanding rules. so perhaps we should support only single instance of NAT network.

enabled

Get or set bool value for 'enabled'

network

Get or set str value for 'network' This is CIDR IPv4 string. Specifying it user defines IPv4 addresses of gateway (low address + 1) and DHCP server (= low address + 2). Note: If there are defined IPv4 port-forward rules update of network will be ignored (because new assignment could break existing rules).

gateway

Get str value for 'gateway' This attribute is read-only. It's recalculated on changing network attribute (low address of network + 1).

i_pv6_enabled

Get or set bool value for 'IPv6Enabled' This attribute define whether gateway will support IPv6 or not.

i_pv6_prefix

Get or set str value for 'IPv6Prefix' This a CIDR IPv6 defining prefix for link-local addresses autoconfiguration within network. Note: ignored if attribute IPv6Enabled is false.

advertise_default_i_pv6_route_enabled

Get or set bool value for 'advertiseDefaultIPv6RouteEnabled'

need_dhcp_server

Get or set bool value for 'needDhcpServer'

event_source

Get IEventSource value for 'eventSource'

port_forward_rules4

Get str value for 'portForwardRules4' Array of NAT port-forwarding rules in string representation, in the following format: "name:protocolid:[host ip]:host port:[guest ip]:guest port".

local_mappings

Get str value for 'localMappings' Array of mappings (address,offset),e.g. ("127.0.1.1=4") maps 127.0.1.1 to networkid + 4.

add_local_mapping(*hostid*, *offset*)

in hostid of type str

in offset of type int

loopback_ip6

Get or set int value for 'loopbackIp6' Offset in ipv6 network from network id for address mapped into loopback6 interface of the host.

port_forward_rules6

Get str value for 'portForwardRules6' Array of NAT port-forwarding rules in string representation, in the following format: "name:protocolid:[host ip]:host port:[guest ip]:guest port".

add_port_forward_rule(*is_ipv6*, *rule_name*, *proto*, *host_ip*, *host_port*, *guest_ip*, *guest_port*)
Protocol handled with the rule.

in is_ipv6 of type bool

in rule_name of type str
in proto of type *NATProtocol* Protocol handled with the rule.
in host_ip of type str IP of the host interface to which the rule should apply. An empty ip address is acceptable, in which case the NAT engine binds the handling socket to any interface.
in host_port of type int The port number to listen on.
in guest_ip of type str The IP address of the guest which the NAT engine will forward matching packets to. An empty IP address is not acceptable.
in guest_port of type int The port number to forward.

remove_port_forward_rule(*i_sipv6*, *rule_name*)
in i_sipv6 of type bool

in rule_name of type str

start(*trunk_type*)
Type of internal network trunk.
in trunk_type of type str Type of internal network trunk.

stop()

class virtualbox.library.**IDHCPServer**(*interface=None*)
The IDHCPServer interface represents the VirtualBox DHCP server configuration.

To enumerate all the DHCP servers on the host, use the *IVirtualBox.dhcp_servers()* attribute.

event_source
Get IEventSource value for 'eventSource'

enabled
Get or set bool value for 'enabled' specifies if the DHCP server is enabled

ip_address
Get str value for 'IPAddress' specifies server IP

network_mask
Get str value for 'networkMask' specifies server network mask

network_name
Get str value for 'networkName' specifies internal network name the server is used for

lower_ip
Get str value for 'lowerIP' specifies from IP address in server address range

upper_ip
Get str value for 'upperIP' specifies to IP address in server address range

add_global_option(*option*, *value*)
in option of type *DhcpOpt*

in value of type str

global_options
Get str value for 'globalOptions'

vm_configs
Get str value for 'vmConfigs'

add_vm_slot_option(*vmname*, *slot*, *option*, *value*)
in vmname of type str

in slot of type int

in option of type *DhcpOpt*

in value of type str

remove_vm_slot_options(*vmname*, *slot*)
in vmname of type str

in slot of type int

get_vm_slot_options(*vmname*, *slot*)
in vmname of type str

in slot of type int

return option of type str

get_mac_options(*mac*)
in mac of type str

return option of type str

set_configuration(*ip_address*, *network_mask*, *from_ip_address*, *to_ip_address*)
configures the server
in ip_address of type str server IP address
in network_mask of type str server network mask
in from_ip_address of type str server From IP address for address range
in to_ip_address of type str server To IP address for address range
raises *OleErrorInvalidarg* invalid configuration supplied

start(*network_name*, *trunk_name*, *trunk_type*)
Starts DHCP server process.
in network_name of type str Name of internal network DHCP server should attach to.
in trunk_name of type str Name of internal network trunk.
in trunk_type of type str Type of internal network trunk.
raises *OleErrorFail* Failed to start the process.

stop()
Stops DHCP server process.
raises *OleErrorFail* Failed to stop the process.

class virtualbox.library.**IVFSExplorer**(*interface=None*)
The VFSExplorer interface unifies access to different file system types. This includes local file systems as well remote file systems like S3. For a list of supported types see *VFSType* . An instance of this is returned by *IAppliance.create_vfs_explorer()* .

path
Get str value for 'path' Returns the current path in the virtual file system.

type_p
Get VFSType value for 'type' Returns the file system type which is currently in use.

update()
Updates the internal list of files/directories from the current directory level. Use *entry_list()* to get the full list after a call to this method.
return progress of type *IProgress* Progress object to track the operation completion.

cd(*dir_p*)
Change the current directory level.

in dir_p of type str The name of the directory to go in.
return progress of type *IProgress* Progress object to track the operation completion.

cd_up()
Go one directory upwards from the current directory level.
return progress of type *IProgress* Progress object to track the operation completion.

entry_list()
Returns a list of files/directories after a call to *update()* . The user is responsible for keeping this internal list up do date.
out names of type str The list of names for the entries.
out types of type int The list of types for the entries. *FsObjType*
out sizes of type int The list of sizes (in bytes) for the entries.
out modes of type int The list of file modes (in octal form) for the entries.

exists(*names*)
Checks if the given file list exists in the current directory level.
in names of type str The names to check.
return exists of type str The names which exist.

remove(*names*)
Deletes the given files in the current directory level.
in names of type str The names to remove.
return progress of type *IProgress* Progress object to track the operation completion.

class virtualbox.library.**ICertificate**(*interface=None*)
X.509 certificate details.

version_number
Get CertificateVersion value for 'versionNumber' Certificate version number.

serial_number
Get str value for 'serialNumber' Certificate serial number.

signature_algorithm_oid
Get str value for 'signatureAlgorithmOID' The dotted OID of the signature algorithm.

signature_algorithm_name
Get str value for 'signatureAlgorithmName' The signature algorithm name if known (if known).

issuer_name
Get str value for 'issuerName' Issuer name. Each member of the array is on the format COMPONENT=NAME, e.g. "C=DE", "ST=Example", "L=For Instance", "O=Beispiel GmbH", "CN=beispiel.example.org".

subject_name
Get str value for 'subjectName' Subject name. Same format as issuerName.

friendly_name
Get str value for 'friendlyName' Friendly subject name or similar.

validity_period_not_before
Get str value for 'validityPeriodNotBefore' Certificate not valid before ISO time stamp.

validity_period_not_after
Get str value for 'validityPeriodNotAfter' Certificate not valid after ISO time stamp.

public_key_algorithm_oid
Get str value for 'publicKeyAlgorithmOID' The dotted OID of the public key algorithm.

public_key_algorithm
Get str value for 'publicKeyAlgorithm' The public key algorithm name (if known).

subject_public_key
Get str value for 'subjectPublicKey' The raw public key bytes.

issuer_unique_identifier
Get str value for 'issuerUniqueIdentifier' Unique identifier of the issuer (empty string if not present).

subject_unique_identifier
Get str value for 'subjectUniqueIdentifier' Unique identifier of this certificate (empty string if not present).

certificate_authority
Get bool value for 'certificateAuthority' Whether this certificate is a certificate authority. Will return E_FAIL if this attribute is not present.

key_usage
Get int value for 'keyUsage' Key usage mask. Will return 0 if not present.

extended_key_usage
Get str value for 'extendedKeyUsage' Array of dotted extended key usage OIDs. Empty array if not present.

raw_cert_data
Get str value for 'rawCertData' The raw certificate bytes.

self_signed
Get bool value for 'selfSigned' Set if self signed certificate.

trusted
Get bool value for 'trusted' Set if the certificate is trusted (by the parent object).

expired
Get bool value for 'expired' Set if the certificate has expired (relevant to the parent object)/

is_currently_expired()
Tests if the certificate has expired at the present time according to the X.509 validity of the certificate.

return result of type bool

query_info(*what*)
Way to extend the interface.

in what of type int

return result of type str

class virtualbox.library.**IInternalMachineControl**(*interface=None*)
Updates the VM state.

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.

update_state(*state*)
Updates the VM state.

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.

in state of type *MachineState*

begin_power_up(*progress*)
Tells VBoxSVC that *IConsole.power_up()* is under ways and gives it the progress

object that should be part of any pending *IMachine.launch_vm_process()* operations. The progress object may be called back to reflect an early cancelation, so some care have to be taken with respect to any cancelation callbacks. The console object will call *IInternalMachineControl.end_power_up()* to signal the completion of the progress object.

in progress of type *IProgress*

end_power_up(*result*)
Tells VBoxSVC that *IConsole.power_up()* has completed. This method may query status information from the progress object it received in *IInternalMachineControl.begin_power_up()* and copy it over to any in-progress *IMachine.launch_vm_process()* call in order to complete that progress object.

in result of type int

begin_powering_down()
Called by the VM process to inform the server it wants to stop the VM execution and power down.
out progress of type *IProgress* Progress object created by VBoxSVC to wait until the VM is powered down.

end_powering_down(*result*, *err_msg*)
Called by the VM process to inform the server that powering down previously requested by #beginPoweringDown is either successfully finished or there was a failure.
in result of type int @c S_OK to indicate success.
in err_msg of type str @c human readable error message in case of failure.
raises *VBoxErrorFileError* Settings file not accessible.
raises *VBoxErrorXmlError* Could not parse the settings file.

run_usb_device_filters(*device*)
Asks the server to run USB devices filters of the associated machine against the given USB device and tell if there is a match.

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).

in device of type *IUSBDevice*

out matched of type bool

out masked_interfaces of type int

capture_usb_device(*id_p*, *capture_filename*)
Requests a capture of the given host USB device. When the request is completed, the VM process will get a *IInternalSessionControl.on_usb_device_attach()* notification.

in id_p of type str

in capture_filename of type str

detach_usb_device(*id_p*, *done*)
Notification that a VM is going to detach (@a done = @c false) or has already detached (@a done = @c true) the given USB device. When the @a done = @c true request is completed, the VM process will get a *IInternalSessionControl.on_usb_device_detach()* notification.

In the @a done = @c 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.

in id_p of type str

in done of type bool

auto_capture_usb_devices()
Requests a capture all matching USB devices attached to the host. When the request is completed, the VM process will get a *IInternalSessionControl.on_usb_device_attach()* notification per every captured device.

detach_all_usb_devices(*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 @a done = @c false the VM is announcing its intentions, while when @a done = @c true the VM is reporting what it has done.

In the @a done = @c 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.

in done of type bool

on_session_end(*session*)
Triggered by the given session object when the session is about to close normally.
in session of type ***ISession*** Session that is being closed
return progress of type ***IProgress*** Used to wait until the corresponding machine is actually dissociated from the given session on the server. Returned only when this session is a direct one.

finish_online_merge_medium()
Gets called by *IInternalSessionControl.online_merge_medium()* . All necessary state information is available at the called object.

pull_guest_properties()
Get the list of the guest properties matching a set of patterns along with their values, time stamps and flags and give responsibility for managing properties to the console.
out names of type str The names of the properties returned.
out values of type str The values of the properties returned. The array entries match the corresponding entries in the @a name array.
out timestamps of type int The time stamps of the properties returned. The array entries match the corresponding entries in the @a name array.
out flags of type str The flags of the properties returned. The array entries match the corresponding entries in the @a name array.

push_guest_property(*name*, *value*, *timestamp*, *flags*)
Update a single guest property in IMachine.
in name of type str The name of the property to be updated.
in value of type str The value of the property.
in timestamp of type int The timestamp of the property.
in flags of type str The flags of the property.

lock_media()
Locks all media attached to the machine for writing and parents of attached differencing media (if any) for reading. This operation is atomic so that if it fails no media is actually locked.

This method is intended to be called when the machine is in Starting or Restoring state. The locked media will be automatically unlocked when the machine is powered off or crashed.

unlock_media()
Unlocks all media previously locked using *IInternalMachineControl.lock_media()* .

This method is intended to be used with teleportation so that it is possible to teleport between processes on the same machine.

eject_medium(*attachment*)
Tells VBoxSVC that the guest has ejected the medium associated with the medium attachment.
in attachment of type *IMediumAttachment* The medium attachment where the eject happened.
return new_attachment of type *IMediumAttachment* A new reference to the medium attachment, as the config change can result in the creation of a new instance.

report_vm_statistics(*valid_stats*, *cpu_user*, *cpu_kernel*, *cpu_idle*, *mem_total*, *mem_free*, *mem_balloon*, *mem_shared*, *mem_cache*, *paged_total*, *mem_alloc_total*, *mem_free_total*, *mem_balloon_total*, *mem_shared_total*, *vm_net_rx*, *vm_net_tx*)
Passes statistics collected by VM (including guest statistics) to VBoxSVC.
in valid_stats of type int Mask defining which parameters are valid. For example: 0x11 means that cpuIdle and XXX are valid. Other parameters should be ignored.
in cpu_user of type int Percentage of processor time spent in user mode as seen by the guest.
in cpu_kernel of type int Percentage of processor time spent in kernel mode as seen by the guest.
in cpu_idle of type int Percentage of processor time spent idling as seen by the guest.
in mem_total of type int Total amount of physical guest RAM.
in mem_free of type int Free amount of physical guest RAM.
in mem_balloon of type int Amount of ballooned physical guest RAM.
in mem_shared of type int Amount of shared physical guest RAM.
in mem_cache of type int Total amount of guest (disk) cache memory.
in paged_total of type int Total amount of space in the page file.
in mem_alloc_total of type int Total amount of memory allocated by the hypervisor.
in mem_free_total of type int Total amount of free memory available in the hypervisor.
in mem_balloon_total of type int Total amount of memory ballooned by the hypervisor.
in mem_shared_total of type int Total amount of shared memory in the hypervisor.
in vm_net_rx of type int Network receive rate for VM.
in vm_net_tx of type int Network transmit rate for VM.

authenticate_external(*auth_params*)
Verify credentials using the external auth library.
in auth_params of type str The auth parameters, credentials, etc.
out result of type str The authentification result.

class virtualbox.library.**IBIOSSettings**(*interface=None*)
The IBIOSSettings interface represents BIOS settings of the virtual machine. This is used only in the *IMachine.bios_settings()* attribute.

logo_fade_in
Get or set bool value for 'logoFadeIn' Fade in flag for BIOS logo animation.

logo_fade_out
Get or set bool value for 'logoFadeOut' Fade out flag for BIOS logo animation.

logo_display_time
Get or set int value for 'logoDisplayTime' BIOS logo display time in milliseconds (0 = default).

logo_image_path
Get or set str value for 'logoImagePath' Local file system path for external BIOS splash image. Empty string means the default image is shown on boot.

boot_menu_mode
Get or set BIOSBootMenuMode value for 'bootMenuMode' Mode of the BIOS boot device menu.

acpi_enabled
Get or set bool value for 'ACPIEnabled' ACPI support flag.

ioapic_enabled
Get or set bool value for 'IOAPICEnabled' I/O-APIC support flag. If set, VirtualBox will provide an I/O-APIC and support IRQs above 15.

apic_mode
Get or set APICMode value for 'APICMode' APIC mode to set up by the firmware.

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

pxe_debug_enabled
Get or set bool value for 'PXEDebugEnabled' PXE debug logging flag. If set, VirtualBox will write extensive PXE trace information to the release log.

non_volatile_storage_file
Get str value for 'nonVolatileStorageFile' The location of the file storing the non-volatile memory content when the VM is powered off. The file does not always exist.

This feature will be realized after VirtualBox v4.3.0.

class virtualbox.library.**IPCIAddress**(*interface=None*)
Address on the PCI bus.

bus
Get or set int value for 'bus' Bus number.

device
Get or set int value for 'device' Device number.

dev_function
Get or set int value for 'devFunction' Device function number.

as_long()
Convert PCI address into long.

return result of type int

from_long(*number*)
Make PCI address from long.

in number of type int

class virtualbox.library.**IPCIDeviceAttachment**(*interface=None*)
Information about PCI attachments.

name
Get str value for 'name' Device name.

is_physical_device
Get bool value for 'isPhysicalDevice' If this is physical or virtual device.

host_address
Get int value for 'hostAddress' Address of device on the host, applicable only to host devices.

guest_address
Get int value for 'guestAddress' Address of device in the guest.

class virtualbox.library.**IEmulatedUSB**(*interface=None*)
Manages emulated USB devices.

webcam_attach(*path*, *settings*)
Attaches the emulated USB webcam to the VM, which will use a host video capture device.
in path of type str The host path of the capture device to use.
in settings of type str Optional settings.

webcam_detach(*path*)
Detaches the emulated USB webcam from the VM
in path of type str The host path of the capture device to detach.

webcams
Get str value for 'webcams' Lists attached virtual webcams.

class virtualbox.library.**IVRDEServerInfo**(*interface=None*)
Contains information about the remote desktop (VRDE) server capabilities and status. This is used in the *IConsole.vrde_server_info()* attribute.

active
Get bool value for 'active' Whether the remote desktop connection is active.

port
Get int value for 'port' VRDE server port number. If this property is equal to 0, then the VRDE server failed to start, usually because there are no free IP ports to bind to. If this property is equal to -1, then the VRDE server has not yet been started.

number_of_clients
Get int value for 'numberOfClients' How many times a client connected.

begin_time
Get int value for 'beginTime' When the last connection was established, in milliseconds since 1970-01-01 UTC.

end_time
Get int value for 'endTime' When the last connection was terminated or the current time, if connection is still active, in milliseconds since 1970-01-01 UTC.

bytes_sent
Get int value for 'bytesSent' How many bytes were sent in last or current, if still active, connection.

bytes_sent_total
Get int value for 'bytesSentTotal' How many bytes were sent in all connections.

bytes_received
Get int value for 'bytesReceived' How many bytes were received in last or current, if still active, connection.

bytes_received_total
Get int value for 'bytesReceivedTotal' How many bytes were received in all connections.

user
Get str value for 'user' Login user name supplied by the client.

domain
Get str value for 'domain' Login domain name supplied by the client.

client_name
Get str value for 'clientName' The client name supplied by the client.

client_ip
Get str value for 'clientIP' The IP address of the client.

client_version
Get int value for 'clientVersion' The client software version number.

encryption_style
Get int value for 'encryptionStyle' Public key exchange method used when connection was established. Values: 0 - RDP4 public key exchange scheme. 1 - X509 certificates were sent to client.

class virtualbox.library.**IHostNetworkInterface**(*interface=None*)
Represents one of host's network interfaces. IP V6 address and network mask are strings of 32 hexadecimal digits grouped by four. Groups are separated by colons. For example, fe80:0000:0000:0000:021e:c2ff:fed2:b030.

name
Get str value for 'name' Returns the host network interface name.

short_name
Get str value for 'shortName' Returns the host network interface short name.

id_p
Get str value for 'id' Returns the interface UUID.

network_name
Get str value for 'networkName' Returns the name of a virtual network the interface gets attached to.

dhcp_enabled
Get bool value for 'DHCPEnabled' Specifies whether the DHCP is enabled for the interface.

ip_address
Get str value for 'IPAddress' Returns the IP V4 address of the interface.

network_mask
Get str value for 'networkMask' Returns the network mask of the interface.

ipv6_supported
Get bool value for 'IPV6Supported' Specifies whether the IP V6 is supported/enabled for the interface.

ipv6_address
Get str value for 'IPV6Address' Returns the IP V6 address of the interface.

ipv6_network_mask_prefix_length
Get int value for 'IPV6NetworkMaskPrefixLength' Returns the length IP V6 network mask prefix of the interface.

hardware_address
Get str value for 'hardwareAddress' Returns the hardware address. For Ethernet it is MAC address.

medium_type
Get HostNetworkInterfaceMediumType value for 'mediumType' Type of protocol encapsulation used.

status
Get HostNetworkInterfaceStatus value for 'status' Status of the interface.

interface_type
Get HostNetworkInterfaceType value for 'interfaceType' specifies the host interface type.

enable_static_ip_config(*ip_address*, *network_mask*)
sets and enables the static IP V4 configuration for the given interface.
in ip_address of type str IP address.
in network_mask of type str network mask.

enable_static_ip_config_v6(*ipv6_address*, *ipv6_network_mask_prefix_length*)
sets and enables the static IP V6 configuration for the given interface.
in ipv6_address of type str IP address.
in ipv6_network_mask_prefix_length of type int network mask.

enable_dynamic_ip_config()
enables the dynamic IP configuration.

dhcp_rediscover()
refreshes the IP configuration for DHCP-enabled interface.

class virtualbox.library.**IHostVideoInputDevice**(*interface=None*)
Represents one of host's video capture devices, for example a webcam.

name
Get str value for 'name' User friendly name.

path
Get str value for 'path' The host path of the device.

alias
Get str value for 'alias' An alias which can be used for IConsole::webcamAttach

class virtualbox.library.**ISystemProperties**(*interface=None*)
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.

min_guest_ram
Get int value for 'minGuestRAM' Minimum guest system memory in Megabytes.

max_guest_ram
Get int value for 'maxGuestRAM' Maximum guest system memory in Megabytes.

min_guest_vram
Get int value for 'minGuestVRAM' Minimum guest video memory in Megabytes.

max_guest_vram
Get int value for 'maxGuestVRAM' Maximum guest video memory in Megabytes.

min_guest_cpu_count
Get int value for 'minGuestCPUCount' Minimum CPU count.

max_guest_cpu_count
Get int value for 'maxGuestCPUCount' Maximum CPU count.

max_guest_monitors
Get int value for 'maxGuestMonitors' Maximum of monitors which could be connected.

info_vd_size
Get int value for 'infoVDSize' Maximum size of a virtual disk image in bytes. Informational value, does not reflect the limits of any virtual disk image format.

serial_port_count
Get int value for 'serialPortCount' Maximum number of serial ports associated with every *IMachine* instance.

parallel_port_count
Get int value for 'parallelPortCount' Maximum number of parallel ports associated with every *IMachine* instance.

max_boot_position
Get int value for '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. *IMachine.set_boot_order()*

raw_mode_supported
Get bool value for 'rawModeSupported' Indicates whether VirtualBox was built with raw-mode support.

When this reads as False, the *HWVirtExPropertyType.enabled* setting will be ignored and assumed to be True.

exclusive_hw_virt
Get or set bool value for 'exclusiveHwVirt' Exclusive use of hardware virtualization by VirtualBox. When enabled, VirtualBox assumes it can obtain full and exclusive access to the VT-x or AMD-V feature of the host. To share hardware virtualization with other hypervisors, this property must be disabled.

This is ignored on OS X, the kernel mediates hardware access there.

default_machine_folder
Get or set str value for 'defaultMachineFolder' Full path to the default directory used to create new or open existing machines when a machine settings file name contains no path.

Starting with VirtualBox 4.0, by default, this attribute contains the full path of folder named "VirtualBox VMs" in the user's home directory, which depends on the host platform.

When setting this attribute, a full path must be specified. Setting this property to @c null or an empty string or the special value "Machines" (for compatibility reasons) will restore that default value.

If the folder specified herein does not exist, it will be created automatically as needed.

IVirtualBox.create_machine() , *IVirtualBox.open_machine()*

logging_level
Get or set str value for 'loggingLevel' Specifies the logging level in current use by VirtualBox.

medium_formats
Get IMediumFormat value for 'mediumFormats' List of all medium storage formats supported by this VirtualBox installation.

Keep in mind that the medium format identifier (*IMediumFormat.id_p()*) used in other API calls like *IVirtualBox.create_medium()* to refer to a particular medium format is a case-insensitive string. This means that, for example, all of the following strings:

```
"VDI"
"vdi"
"VdI"
```

refer to the same medium format.

Note that the virtual medium framework is backend-based, therefore the list of supported formats depends on what backends are currently installed.

IMediumFormat

default_hard_disk_format
Get or set str value for 'defaultHardDiskFormat' Identifier of the default medium format used by VirtualBox.

The medium format set by this attribute is used by VirtualBox when the medium format was not specified explicitly. One example is *IVirtualBox.create_medium()* with the empty format argument. A more complex example is implicit creation of differencing media when taking a snapshot of a virtual machine: this operation will try to use a format of the parent medium first and if this format does not support differencing media the default format specified by this argument will be used.

The list of supported medium formats may be obtained by the *medium_formats()* call. Note that the default medium format must have a capability to create differencing media; otherwise operations that create media implicitly may fail unexpectedly.

The initial value of this property is "VDI" in the current version of the VirtualBox product, but may change in the future.

Setting this property to @c null or empty string will restore the initial value.

medium_formats() , *IMediumFormat.id_p()* , *IVirtualBox.create_medium()*

free_disk_space_warning
Get or set int value for 'freeDiskSpaceWarning' Issue a warning if the free disk space is below (or in some disk intensive operation is expected to go below) the given size in bytes.

free_disk_space_percent_warning
Get or set int value for 'freeDiskSpacePercentWarning' Issue a warning if the free disk space is below (or in some disk intensive operation is expected to go below) the given percentage.

free_disk_space_error
Get or set int value for 'freeDiskSpaceError' Issue an error if the free disk space is below (or in some disk intensive operation is expected to go below) the given size in bytes.

free_disk_space_percent_error
Get or set int value for 'freeDiskSpacePercentError' Issue an error if the free disk space is below (or in some disk intensive operation is expected to go below) the given percentage.

vrde_auth_library
Get or set str value for 'VRDEAuthLibrary' Library that provides authentication for Remote Desktop clients. The library is used if a virtual machine's authentication type is set to "external" in the VM RemoteDisplay configuration.

The system library extension (".DLL" or ".so") must be omitted. A full path can be specified; if not, then the library must reside on the system's default library path.

The default value of this property is "VBoxAuth". There is a library of that name in one of the default VirtualBox library directories.

For details about VirtualBox authentication libraries and how to implement them, please refer to the VirtualBox manual.

Setting this property to @c null or empty string will restore the initial value.

web_service_auth_library
Get or set str value for '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.vrde_auth_library()* , there is no per-VM setting for this, as the webservice is a global resource (if it is running). Only for this setting (for the webservice), setting this value to a literal "null" string disables authentication, meaning that `IWebsessionManager.logon()` will always succeed, no matter what user name and password are supplied.

The initial value of this property is "VBoxAuth", meaning that the webservice will use the same authentication library that is used by default for VRDE (again, see *ISystemProperties.vrde_auth_library()*). The format and calling convention of authentication libraries is the same for the webservice as it is for VRDE.

Setting this property to @c null or empty string will restore the initial value.

default_vrde_ext_pack
Get or set str value for 'defaultVRDEExtPack' The name of the extension pack providing the default VRDE.

This attribute is for choosing between multiple extension packs providing VRDE. If only one is installed, it will automatically be the default one. The attribute value can be empty if no VRDE extension pack is installed.

For details about VirtualBox Remote Desktop Extension and how to implement one, please refer to the VirtualBox SDK.

log_history_count
Get or set int value for 'logHistoryCount' This value specifies how many old release log files are kept.

default_audio_driver
Get AudioDriverType value for 'defaultAudioDriver' This value hold the default audio driver for the current system.

autostart_database_path
Get or set str value for 'autostartDatabasePath' The path to the autostart database. Depending on the host this might be a filesystem path or something else.

default_additions_iso
Get or set str value for 'defaultAdditionsISO' The path to the default Guest Additions ISO image. Can be empty if the location is not known in this installation.

default_frontend
Get or set str value for 'defaultFrontend' Selects which VM frontend should be used by default when launching a VM through the *IMachine.launch_vm_process()* method. Empty or @c null strings do not define a particular default, it is up to *IMachine.launch_vm_process()* to select one. See the description of *IMachine.launch_vm_process()* for the valid frontend types.

This global setting is overridden by the per-VM attribute *IMachine.default_frontend()* or a frontend type passed to *IMachine.launch_vm_process()* .

screen_shot_formats
Get BitmapFormat value for 'screenShotFormats' Supported bitmap formats which can be used with takeScreenShot and takeScreenShotToArray methods.

get_max_network_adapters (*chipset*)
Maximum total number of network adapters associated with every *IMachine* instance.
in chipset of type *ChipsetType* The chipset type to get the value for.
return max_network_adapters of type int The maximum total number of network adapters allowed.

get_max_network_adapters_of_type(*chipset*, *type_p*)
Maximum number of network adapters of a given attachment type, associated with every *IMachine* instance.
in chipset of type *ChipsetType* The chipset type to get the value for.
in type_p of type *NetworkAttachmentType* Type of attachment.
return max_network_adapters of type int The maximum number of network adapters allowed for particular chipset and attachment type.

get_max_devices_per_port_for_storage_bus(*bus*)
Returns the maximum number of devices which can be attached to a port for the given storage bus.
in bus of type *StorageBus* The storage bus type to get the value for.
return max_devices_per_port of type int The maximum number of devices which can be attached to the port for the given storage bus.

get_min_port_count_for_storage_bus(*bus*)
Returns the minimum number of ports the given storage bus supports.
in bus of type *StorageBus* The storage bus type to get the value for.
return min_port_count of type int The minimum number of ports for the given storage bus.

get_max_port_count_for_storage_bus(*bus*)
Returns the maximum number of ports the given storage bus supports.
in bus of type *StorageBus* The storage bus type to get the value for.
return max_port_count of type int The maximum number of ports for the given storage bus.

get_max_instances_of_storage_bus(*chipset*, *bus*)
Returns the maximum number of storage bus instances which can be configured for each VM. This corresponds to the number of storage controllers one can have. Value may depend on chipset type used.
in chipset of type *ChipsetType* The chipset type to get the value for.
in bus of type *StorageBus* The storage bus type to get the value for.
return max_instances of type int The maximum number of instances for the given storage bus.

get_device_types_for_storage_bus(*bus*)
Returns list of all the supported device types (*DeviceType*) for the given type of storage bus.
in bus of type *StorageBus* The storage bus type to get the value for.
return device_types of type *DeviceType* The list of all supported device types for the given storage bus.

get_default_io_cache_setting_for_storage_controller(*controller_type*)
Returns the default I/O cache setting for the given storage controller
in controller_type of type *StorageControllerType* The storage controller type to get the setting for.
return enabled of type bool Returned flag indicating the default value

get_storage_controller_hotplug_capable(*controller_type*)
Returns whether the given storage controller supports hot-plugging devices.
in controller_type of type *StorageControllerType* The storage controller to check the setting for.
return hotplug_capable of type bool Returned flag indicating whether the controller is hotplug capable

get_max_instances_of_usb_controller_type(*chipset*, *type_p*)
Returns the maximum number of USB controller instances which can be configured for each VM. This corresponds to the number of USB controllers one can have. Value may depend on chipset type used.
in chipset of type *ChipsetType* The chipset type to get the value for.

in type_p of type ***USBControllerType*** The USB controller type to get the value for.
return max_instances of type int The maximum number of instances for the given USB controller type.

class virtualbox.library.**IAdditionsFacility**(*interface=None*)
Structure representing a Guest Additions facility.

class_type
Get AdditionsFacilityClass value for 'classType' The class this facility is part of.

last_updated
Get int value for 'lastUpdated' Time stamp of the last status update, in milliseconds since 1970-01-01 UTC.

name
Get str value for 'name' The facility's friendly name.

status
Get AdditionsFacilityStatus value for 'status' The current status.

type_p
Get AdditionsFacilityType value for 'type' The facility's type ID.

class virtualbox.library.**IDnDBase**(*interface=None*)
Base abstract interface for drag'n drop.

formats
Get str value for 'formats' Returns all supported drag'n drop formats.

protocol_version
Get int value for 'protocolVersion' Returns the protocol version which is used to communicate with the guest.

is_format_supported(*format_p*)
Checks if a specific drag'n drop MIME / Content-type format is supported.
in format_p of type str Format to check for.
return supported of type bool Returns @c true if the specified format is supported, @c false if not.

add_formats(*formats*)
Adds MIME / Content-type formats to the supported formats.
in formats of type str Collection of formats to add.

remove_formats(*formats*)
Removes MIME / Content-type formats from the supported formats.
in formats of type str Collection of formats to remove.

class virtualbox.library.**IDnDSource**(*interface=None*)
Abstract interface for handling drag'n drop sources.

drag_is_pending(*screen_id*)
Ask the source if there is any drag and drop operation pending. If no drag and drop operation is pending currently, DnDAction_Ignore is returned.
in screen_id of type int The screen ID where the drag and drop event occurred.
out formats of type str On return the supported mime types.
out allowed_actions of type ***DnDAction*** On return the actions which are allowed.
return default_action of type ***DnDAction*** On return the default action to use.
raises ***VBoxErrorVmError*** VMM device is not available.

drop(*format_p*, *action*)
Informs the source that a drop event occurred for a pending drag and drop operation.

in format_p of type str The mime type the data must be in.
in action of type *DnDAction* The action to use.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorVmError* VMM device is not available.

receive_data()
Receive the data of a previously drag and drop event from the source.
return data of type str The actual data.
raises *VBoxErrorVmError* VMM device is not available.

class virtualbox.library.**IGuestDnDSource**(*interface=None*)
Implementation of the *IDnDSource* object for source drag'n drop operations on the guest.

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IDnDTarget**(*interface=None*)
Abstract interface for handling drag'n drop targets.

enter(*screen_id*, *y*, *x*, *default_action*, *allowed_actions*, *formats*)
Informs the target about a drag and drop enter event.
in screen_id of type int The screen ID where the drag and drop event occurred.
in y of type int Y-position of the event.
in x of type int X-position of the event.
in default_action of type *DnDAction* The default action to use.
in allowed_actions of type *DnDAction* The actions which are allowed.
in formats of type str The supported MIME types.
return result_action of type *DnDAction* The resulting action of this event.
raises *VBoxErrorVmError* VMM device is not available.

move(*screen_id*, *x*, *y*, *default_action*, *allowed_actions*, *formats*)
Informs the target about a drag and drop move event.
in screen_id of type int The screen ID where the drag and drop event occurred.
in x of type int X-position of the event.
in y of type int Y-position of the event.
in default_action of type *DnDAction* The default action to use.
in allowed_actions of type *DnDAction* The actions which are allowed.
in formats of type str The supported MIME types.
return result_action of type *DnDAction* The resulting action of this event.
raises *VBoxErrorVmError* VMM device is not available.

leave(*screen_id*)
Informs the target about a drag and drop leave event.
in screen_id of type int The screen ID where the drag and drop event occurred.
raises *VBoxErrorVmError* VMM device is not available.

drop(*screen_id*, *x*, *y*, *default_action*, *allowed_actions*, *formats*)
Informs the target about a drop event.
in screen_id of type int The screen ID where the Drag and Drop event occurred.
in x of type int X-position of the event.
in y of type int Y-position of the event.
in default_action of type *DnDAction* The default action to use.
in allowed_actions of type *DnDAction* The actions which are allowed.
in formats of type str The supported MIME types.
out format_p of type str The resulting format of this event.
return result_action of type *DnDAction* The resulting action of this event.
raises *VBoxErrorVmError* VMM device is not available.

send_data(*screen_id*, *format_p*, *data*)
Initiates sending data to the target.
in screen_id of type int The screen ID where the drag and drop event occurred.
in format_p of type str The MIME type the data is in.
in data of type str The actual data.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorVmError* VMM device is not available.

cancel()
Requests cancelling the current operation. The target can veto the request in case the operation is not cancelable at the moment.
return veto of type bool Whether the target has vetoed cancelling the operation.
raises *VBoxErrorVmError* VMM device is not available.

class virtualbox.library.**IGuestDnDTarget**(*interface=None*)
Implementation of the *IDnDTarget* object for target drag'n drop operations on the guest.

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IDirectory**(*interface=None*)
Abstract parent interface for directories handled by VirtualBox.

directory_name
Get str value for 'directoryName' The path specified when opening the directory.

filter_p
Get str value for 'filter' Directory listing filter to (specified when opening the directory).

close()
Closes this directory. After closing operations like reading the next directory entry will not be possible anymore.

read()
Reads the next directory entry of this directory.
return obj_info of type *IFsObjInfo* Object information of the current directory entry read.
Also see *IFsObjInfo* .
raises *VBoxErrorObjectNotFound* No more directory entries to read.

class virtualbox.library.**IGuestDirectory**(*interface=None*)
Implementation of the *IDirectory* object for directories in the guest.

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IFile**(*interface=None*)
Abstract parent interface for files handled by VirtualBox.

event_source
Get IEventSource value for 'eventSource' Event source for file events.

id_p
Get int value for 'id' The ID VirtualBox internally assigned to the open file.

initial_size
Get int value for 'initialSize' The initial size in bytes when opened.

offset
Get int value for 'offset' The current file position.

The file current position always applies to the *IFile.read()* method, which updates it upon return. Same goes for the *IFile.write()* method except when *IFile.access_mode()*

is *FileAccessMode.append_only* or *FileAccessMode.append_read* , where it will always write to the end of the file and will leave this attribute unchanged.

The *IFile.seek()* is used to change this attribute without transfering any file data like read and write does.

status
Get FileStatus value for 'status' Current file status.

file_name
Get str value for 'fileName' Full path of the actual file name of this file. <!– r=bird: The 'actual' file name is too tough, we cannot guarentee that on unix guests. Seeing how IGuestDirectory did things, I'm questioning the 'Full path' part too. Not urgent to check. –>

creation_mode
Get int value for 'creationMode' The UNIX-style creation mode specified when opening the file.

open_action
Get FileOpenAction value for 'openAction' The opening action specified when opening the file.

access_mode
Get FileAccessMode value for 'accessMode' The file access mode.

close()
Closes this file. After closing operations like reading data, writing data or querying information will not be possible anymore.

query_info()
Queries information about this file.
return obj_info of type *IFsObjInfo* Object information of this file. Also see *IFsObjInfo* .
raises *OleErrorNotimpl* The method is not implemented yet.

query_size()
Queries the current file size.
return size of type int Queried file size.
raises *OleErrorNotimpl* The method is not implemented yet.

read(*to_read*, *timeout_ms*)
Reads data from this file.
in to_read of type int Number of bytes to read.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return data of type str Array of data read.
raises *OleErrorNotimpl* The method is not implemented yet.

read_at(*offset*, *to_read*, *timeout_ms*)
Reads data from an offset of this file.
in offset of type int Offset in bytes to start reading.
in to_read of type int Number of bytes to read.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return data of type str Array of data read.
raises *OleErrorNotimpl* The method is not implemented yet.

seek(*offset*, *whence*)
Changes the current file position of this file.

The file current position always applies to the *IFile.read()* method. Same for the *IFile.write()* method it except when the *IFile.access_mode()* is *FileAccessMode.append_only* or *FileAccessMode.append_read*.
in offset of type int Offset to seek relative to the position specified by @a whence.
in whence of type *FileSeekOrigin* One of the *FileSeekOrigin* seek starting points.
return new_offset of type int The new file offset after the seek operation.

set_acl(*acl*, *mode*)
Sets the ACL of this file.
in acl of type str The ACL specification string. To-be-defined.
in mode of type int UNIX-style mode mask to use if @a acl is empty. As mention in *IGuestSession.directory_create()* this is realized on a best effort basis and the exact behavior depends on the Guest OS.
raises *OleErrorNotimpl* The method is not implemented yet.

set_size(*size*)
Changes the file size.
in size of type int The new file size.
raises *OleErrorNotimpl* The method is not implemented yet.

write(*data*, *timeout_ms*)
Writes bytes to this file.
in data of type str Array of bytes to write. The size of the array also specifies how much to write.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return written of type int How much bytes were written.

write_at(*offset*, *data*, *timeout_ms*)
Writes bytes at a certain offset to this file.
in offset of type int Offset in bytes to start writing.
in data of type str Array of bytes to write. The size of the array also specifies how much to write.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return written of type int How much bytes were written.
raises *OleErrorNotimpl* The method is not implemented yet.

class virtualbox.library.**IGuestFile**(*interface=None*)
Implementation of the *IFile* object for files in the guest.

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IFsObjInfo**(*interface=None*)
Abstract parent interface for VirtualBox file system object information. This can be information about a file or a directory, for example.

access_time
Get int value for 'accessTime' Time of last access (st_atime).

allocated_size
Get int value for 'allocatedSize' Disk allocation size (st_blocks * DEV_BSIZE).

birth_time
Get int value for 'birthTime' Time of file birth (st_birthtime).

change_time
Get int value for 'changeTime' Time of last status change (st_ctime).

device_number
Get int value for 'deviceNumber' The device number of a character or block device type object (st_rdev).

file_attributes
Get str value for 'fileAttributes' File attributes. Not implemented yet.

generation_id
Get int value for 'generationId' The current generation number (st_gen).

gid
Get int value for 'GID' The group the filesystem object is assigned (st_gid).

group_name
Get str value for 'groupName' The group name.

hard_links
Get int value for 'hardLinks' Number of hard links to this filesystem object (st_nlink).

modification_time
Get int value for 'modificationTime' Time of last data modification (st_mtime).

name
Get str value for 'name' The object's name.

node_id
Get int value for 'nodeId' The unique identifier (within the filesystem) of this filesystem object (st_ino).

node_id_device
Get int value for 'nodeIdDevice' The device number of the device which this filesystem object resides on (st_dev).

object_size
Get int value for 'objectSize' The logical size (st_size). For normal files this is the size of the file. For symbolic links, this is the length of the path name contained in the symbolic link. For other objects this fields needs to be specified.

type_p
Get FsObjType value for 'type' The object type. See *FsObjType* for more.

uid
Get int value for 'UID' The user owning the filesystem object (st_uid).

user_flags
Get int value for 'userFlags' User flags (st_flags).

user_name
Get str value for 'userName' The user name.

class virtualbox.library.**IGuestFsObjInfo**(*interface=None*)
Represents the guest implementation of the *IFsObjInfo* object.

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**ISnapshot**(*interface=None*)
The ISnapshot interface represents a snapshot of the virtual machine.

Together with the differencing media that are created when a snapshot is taken, a machine can be brought back to the exact state it was in when the snapshot was taken.

The ISnapshot interface has no methods, only attributes; snapshots are controlled through methods of the *IMachine* interface which also manage the media associated with the snapshot. The following operations exist:

IMachine.take_snapshot() creates a new snapshot by creating new, empty differencing images for the machine's media and saving the VM settings and (if the VM is running) the current VM state in the snapshot.

The differencing images will then receive all data written to the machine's media, while their parent (base) images remain unmodified after the snapshot has been taken (see *IMedium* for details about differencing images). This simplifies restoring a machine to the state of a snapshot: only the differencing images need to be deleted.

The current machine state is not changed by taking a snapshot except that *IMachine.current_snapshot()* is set to the newly created snapshot, which is also added to the machine's snapshots tree.

IMachine.restore_snapshot() resets a machine to the state of a previous snapshot by deleting the differencing image of each of the machine's media and setting the machine's settings and state to the state that was saved in the snapshot (if any).

This destroys the machine's current state. After calling this, *IMachine.current_snapshot()* points to the snapshot that was restored.

IMachine.delete_snapshot() deletes a snapshot without affecting the current machine state.

This does not change the current machine state, but instead frees the resources allocated when the snapshot was taken: the settings and machine state file are deleted (if any), and the snapshot's differencing image for each of the machine's media gets merged with its parent image.

Neither the current machine state nor other snapshots are affected by this operation, except that parent media will be modified to contain the disk data associated with the snapshot being deleted.

When deleting the current snapshot, the *IMachine.current_snapshot()* attribute is set to the current snapshot's parent or @c null if it has no parent. Otherwise the attribute is unchanged.

Each snapshot contains a copy of virtual machine's settings (hardware configuration etc.). This copy is contained in an immutable (read-only) instance of *IMachine* which is available from the snapshot's *machine()* attribute. When restoring the snapshot, these settings are copied back to the original machine.

In addition, if the machine was running when the snapshot was taken (*IMachine.state()* is *MachineState.running*), the current VM state is saved in the snapshot (similarly to what happens when a VM's state is saved). The snapshot is then said to be *online* because when restoring it, the VM will be running.

If the machine was in *MachineState.saved* saved saved, the snapshot receives a copy of the execution state file (*IMachine.state_file_path()*).

Otherwise, if the machine was not running (*MachineState.powered_off* or *MachineState.aborted*), the snapshot is *offline*; it then contains a so-called "zero execution state", representing a machine that is powered off.

id_p
Get str value for 'id' UUID of the snapshot.

name
Get or set str value for 'name' Short name of the snapshot. Setting this attribute causes *IMachine.save_settings()* to be called implicitly.

description
Get or set str value for 'description' Optional description of the snapshot. Setting this attribute causes *IMachine.save_settings()* to be called implicitly.

time_stamp
Get int value for 'timeStamp' Time stamp of the snapshot, in milliseconds since 1970-01-01 UTC.

online
Get bool value for 'online' @c true if this snapshot is an online snapshot and @c false otherwise.

When this attribute is @c true, the *IMachine.state_file_path()* attribute of the *machine()* object associated with this snapshot will point to the saved state file. Otherwise, it will be an empty string.

machine
Get IMachine value for 'machine' Virtual machine this snapshot is taken on. This object stores all settings the machine had when taking this snapshot.

The returned machine object is immutable, i.e. no any settings can be changed.

parent
Get ISnapshot value for 'parent' Parent snapshot (a snapshot this one is based on), or @c null if the snapshot has no parent (i.e. is the first snapshot).

children
Get ISnapshot value for 'children' Child snapshots (all snapshots having this one as a parent). By inspecting this attribute starting with a machine's root snapshot (which can be obtained by calling *IMachine.find_snapshot()* with a @c null UUID), a machine's snapshots tree can be iterated over.

get_children_count()
Returns the number of direct children of this snapshot.

return children_count of type int

class virtualbox.library.**IMediumAttachment**(*interface=None*)
The IMediumAttachment interface links storage media to virtual machines. For each medium (*IMedium*) which has been attached to a storage controller (*IStorageController*) of a machine (*IMachine*) via the *IMachine.attach_device()* method, one instance of IMediumAttachment is added to the machine's *IMachine.medium_attachments()* array attribute.

Each medium attachment specifies the storage controller as well as a port and device number and the IMedium instance representing a virtual hard disk or floppy or DVD image.

For removable media (DVDs or floppies), there are two additional options. For one, the IMedium instance can be @c null to represent an empty drive with no media inserted (see *IMachine.mount_medium()*); secondly, the medium can be one of the pseudo-media for host drives listed in `IHost.dvd_drives()` or `IHost.floppy_drives()` .

Attaching Hard Disks

Hard disks are attached to virtual machines using the *IMachine.attach_device()* method and detached using the *IMachine.detach_device()* method. Depending on a medium's type (see *IMedium.type_p()*), hard disks are attached either *directly* or *indirectly*.

When a hard disk is being attached directly, it is associated with the virtual machine and used for hard disk operations when the machine is running. When a hard disk is being attached indirectly, a new differencing hard disk linked to it is implicitly created and this differencing hard disk is associated with the machine and used for hard disk operations. This also means that if *IMachine.attach_device()* performs a direct attachment then the same hard disk will be returned in

response to the subsequent *IMachine.get_medium()* call; however if an indirect attachment is performed then *IMachine.get_medium()* will return the implicitly created differencing hard disk, not the original one passed to *IMachine.attach_device()* . In detail:

Normal base hard disks that do not have children (i.e. differencing hard disks linked to them) and that are not already attached to virtual machines in snapshots are attached **directly**. Otherwise, they are attached **indirectly** because having dependent children or being part of the snapshot makes it impossible to modify hard disk contents without breaking the integrity of the dependent party. The *IMedium.read_only()* attribute allows to quickly determine the kind of the attachment for the given hard disk. Note that if a normal base hard disk is to be indirectly attached to a virtual machine with snapshots then a special procedure called *smart attachment* is performed (see below). **Normal differencing** hard disks are like normal base hard disks: they are attached **directly** if they do not have children and are not attached to virtual machines in snapshots, and **indirectly** otherwise. Note that the smart attachment procedure is never performed for differencing hard disks. **Immutable** hard disks are always attached **indirectly** because they are designed to be non-writable. If an immutable hard disk is attached to a virtual machine with snapshots then a special procedure called smart attachment is performed (see below). **Writethrough** hard disks are always attached **directly**, also as designed. This also means that writethrough hard disks cannot have other hard disks linked to them at all. **Shareable** hard disks are always attached **directly**, also as designed. This also means that shareable hard disks cannot have other hard disks linked to them at all. They behave almost like writethrough hard disks, except that shareable hard disks can be attached to several virtual machines which are running, allowing concurrent accesses. You need special cluster software running in the virtual machines to make use of such disks.

Note that the same hard disk, regardless of its type, may be attached to more than one virtual machine at a time. In this case, the machine that is started first gains exclusive access to the hard disk and attempts to start other machines having this hard disk attached will fail until the first machine is powered down.

Detaching hard disks is performed in a *deferred* fashion. This means that the given hard disk remains associated with the given machine after a successful *IMachine.detach_device()* call until *IMachine.save_settings()* is called to save all changes to machine settings to disk. This deferring is necessary to guarantee that the hard disk configuration may be restored at any time by a call to *IMachine.discard_settings()* before the settings are saved (committed).

Note that if *IMachine.discard_settings()* is called after indirectly attaching some hard disks to the machine but before a call to *IMachine.save_settings()* is made, it will implicitly delete all differencing hard disks implicitly created by *IMachine.attach_device()* for these indirect attachments. Such implicitly created hard disks will also be immediately deleted when detached explicitly using the *IMachine.detach_device()* call if it is made before *IMachine.save_settings()* . This implicit deletion is safe because newly created differencing hard disks do not contain any user data.

However, keep in mind that detaching differencing hard disks that were implicitly created by *IMachine.attach_device()* before the last *IMachine.save_settings()* call will **not** implicitly delete them as they may already contain some data (for example, as a result of virtual machine execution). If these hard disks are no more necessary, the caller can always delete them explicitly using *IMedium.delete_storage()* after they are actually de-associated from this machine by the *IMachine.save_settings()* call.

Smart Attachment

When normal base or immutable hard disks are indirectly attached to a virtual machine then some additional steps are performed to make sure the virtual machine will have the most recent "view" of the hard disk being attached. These steps include walking through the machine's snapshots starting from the current one and going through ancestors up to the first snapshot. Hard disks attached to the virtual machine in all of the encountered snapshots are checked whether they are descendants of

the given normal base or immutable hard disk. The first found child (which is the differencing hard disk) will be used instead of the normal base or immutable hard disk as a parent for creating a new differencing hard disk that will be actually attached to the machine. And only if no descendants are found or if the virtual machine does not have any snapshots then the normal base or immutable hard disk will be used itself as a parent for this differencing hard disk.

It is easier to explain what smart attachment does using the following example:

```
BEFORE attaching B.vdi:       AFTER attaching B.vdi:

Snapshot 1 (B.vdi)            Snapshot 1 (B.vdi)
Snapshot 2 (D1->B.vdi)        Snapshot 2 (D1->B.vdi)
Snapshot 3 (D2->D1.vdi)       Snapshot 3 (D2->D1.vdi)
Snapshot 4 (none)             Snapshot 4 (none)
CurState   (none)             CurState   (D3->D2.vdi)

NOT
...
CurState   (D3->B.vdi)
```

The first column is the virtual machine configuration before the base hard disk B.vdi is attached, the second column shows the machine after this hard disk is attached. Constructs like D1->B.vdi and similar mean that the hard disk that is actually attached to the machine is a differencing hard disk, D1.vdi, which is linked to (based on) another hard disk, B.vdi.

As we can see from the example, the hard disk B.vdi was detached from the machine before taking Snapshot 4. Later, after Snapshot 4 was taken, the user decides to attach B.vdi again. B.vdi has dependent child hard disks (D1.vdi, D2.vdi), therefore it cannot be attached directly and needs an indirect attachment (i.e. implicit creation of a new differencing hard disk). Due to the smart attachment procedure, the new differencing hard disk (D3.vdi) will be based on D2.vdi, not on B.vdi itself, since D2.vdi is the most recent view of B.vdi existing for this snapshot branch of the given virtual machine.

Note that if there is more than one descendant hard disk of the given base hard disk found in a snapshot, and there is an exact device, channel and bus match, then this exact match will be used. Otherwise, the youngest descendant will be picked up.

There is one more important aspect of the smart attachment procedure which is not related to snapshots at all. Before walking through the snapshots as described above, the backup copy of the current list of hard disk attachment is searched for descendants. This backup copy is created when the hard disk configuration is changed for the first time after the last *IMachine.save_settings()* call and used by *IMachine.discard_settings()* to undo the recent hard disk changes. When such a descendant is found in this backup copy, it will be simply re-attached back, without creating a new differencing hard disk for it. This optimization is necessary to make it possible to re-attach the base or immutable hard disk to a different bus, channel or device slot without losing the contents of the differencing hard disk actually attached to the machine in place of it.

medium
: Get IMedium value for 'medium' Medium object associated with this attachment; it can be @c null for removable devices.

controller
: Get str value for 'controller' Name of the storage controller of this attachment; this refers to one of the controllers in *IMachine.storage_controllers()* by name.

port
: Get int value for 'port' Port number of this attachment. See *IMachine.attach_device()* for the meaning of this value for the different controller types.

device
Get int value for 'device' Device slot number of this attachment. See *IMachine.attach_device()* for the meaning of this value for the different controller types.

type_p
Get DeviceType value for 'type' Device type of this attachment.

passthrough
Get bool value for 'passthrough' Pass I/O requests through to a device on the host.

temporary_eject
Get bool value for 'temporaryEject' Whether guest-triggered eject results in unmounting the medium.

is_ejected
Get bool value for 'isEjected' Signals that the removable medium has been ejected. This is not necessarily equivalent to having a @c null medium association.

non_rotational
Get bool value for 'nonRotational' Whether the associated medium is non-rotational.

discard
Get bool value for 'discard' Whether the associated medium supports discarding unused blocks.

hot_pluggable
Get bool value for 'hotPluggable' Whether this attachment is hot pluggable or not.

bandwidth_group
Get IBandwidthGroup value for 'bandwidthGroup' The bandwidth group this medium attachment is assigned to.

class virtualbox.library.**IMedium**(*interface=None*)
The IMedium interface represents virtual storage for a machine's hard disks, CD/DVD or floppy drives. It will typically represent a disk image on the host, for example a VDI or VMDK file representing a virtual hard disk, or an ISO or RAW file representing virtual removable media, but can also point to a network location (e.g. for iSCSI targets).

Instances of IMedium are connected to virtual machines by way of medium attachments, which link the storage medium to a particular device slot of a storage controller of the virtual machine. In the VirtualBox API, virtual storage is therefore always represented by the following chain of object links:

IMachine.storage_controllers() contains an array of storage controllers (IDE, SATA, SCSI, SAS or a floppy controller; these are instances of *IStorageController*). *IMachine.medium_attachments()* contains an array of medium attachments (instances of *IMediumAttachment* created by *IMachine.attach_device()*), each containing a storage controller from the above array, a port/device specification, and an instance of IMedium representing the medium storage (image file).

For removable media, the storage medium is optional; a medium attachment with no medium represents a CD/DVD or floppy drive with no medium inserted. By contrast, hard disk attachments will always have an IMedium object attached. Each IMedium in turn points to a storage unit (such as a file on the host computer or a network resource) that holds actual data. This location is represented by the *location()* attribute.

Existing media are opened using *IVirtualBox.open_medium()* ; new hard disk media can be created with the VirtualBox API using the *IVirtualBox.create_medium()* method. Differencing hard disks (see below) are usually implicitly created by VirtualBox as needed, but may also be created explicitly using *create_diff_storage()* . VirtualBox cannot create CD/DVD or

floppy images (ISO and RAW files); these should be created with external tools and then opened from within VirtualBox.

Only for CD/DVDs and floppies, an IMedium instance can also represent a host drive. In that case the *id_p()* attribute contains the UUID of one of the drives in `IHost.dvd_drives()` or `IHost.floppy_drives()`.

Media registries

When a medium has been opened or created using one of the aforementioned APIs, it becomes "known" to VirtualBox. Known media can be attached to virtual machines and re-found through *IVirtualBox.open_medium()* . They also appear in the global *IVirtualBox.hard_disks()* , *IVirtualBox.dvd_images()* and *IVirtualBox.floppy_images()* arrays.

Prior to VirtualBox 4.0, opening a medium added it to a global media registry in the VirtualBox.xml file, which was shared between all machines and made transporting machines and their media from one host to another difficult.

Starting with VirtualBox 4.0, media are only added to a registry when they are *attached* to a machine using *IMachine.attach_device()* . For backwards compatibility, which registry a medium is added to depends on which VirtualBox version created a machine:

If the medium has first been attached to a machine which was created by VirtualBox 4.0 or later, it is added to that machine's media registry in the machine XML settings file. This way all information about a machine's media attachments is contained in a single file and can be transported easily. For older media attachments (i.e. if the medium was first attached to a machine which was created with a VirtualBox version before 4.0), media continue to be registered in the global VirtualBox settings file, for backwards compatibility.

See *IVirtualBox.open_medium()* for more information.

Media are removed from media registries by the *IMedium.close()* , *delete_storage()* and *merge_to()* methods.

Accessibility checks

VirtualBox defers media accessibility checks until the *refresh_state()* method is called explicitly on a medium. This is done to make the VirtualBox object ready for serving requests as fast as possible and let the end-user application decide if it needs to check media accessibility right away or not.

As a result, when VirtualBox starts up (e.g. the VirtualBox object gets created for the first time), all known media are in the "Inaccessible" state, but the value of the *last_access_error()* attribute is an empty string because no actual accessibility check has been made yet.

After calling *refresh_state()* , a medium is considered *accessible* if its storage unit can be read. In that case, the *state()* attribute has a value of "Created". If the storage unit cannot be read (for example, because it is located on a disconnected network resource, or was accidentally deleted outside VirtualBox), the medium is considered *inaccessible*, which is indicated by the "Inaccessible" state. The exact reason why the medium is inaccessible can be obtained by reading the *last_access_error()* attribute.

Medium types

There are five types of medium behavior which are stored in the *type_p()* attribute (see *MediumType*) and which define the medium's behavior with attachments and snapshots.

All media can be also divided in two groups: *base* media and *differencing* media. A base medium contains all sectors of the medium data in its own storage and therefore can be used independently. In contrast, a differencing medium is a "delta" to some other medium and contains only those sectors

which differ from that other medium, which is then called a *parent*. The differencing medium is said to be *linked to* that parent. The parent may be itself a differencing medium, thus forming a chain of linked media. The last element in that chain must always be a base medium. Note that several differencing media may be linked to the same parent medium.

Differencing media can be distinguished from base media by querying the *parent()* attribute: base media do not have parents they would depend on, so the value of this attribute is always @c null for them. Using this attribute, it is possible to walk up the medium tree (from the child medium to its parent). It is also possible to walk down the tree using the *children()* attribute.

Note that the type of all differencing media is "normal"; all other values are meaningless for them. Base media may be of any type.

Automatic composition of the file name part

Another extension to the *IMedium.location()* attribute is that there is a possibility to cause VirtualBox to compose a unique value for the file name part of the location using the UUID of the hard disk. This applies only to hard disks in *MediumState.not_created* state, e.g. before the storage unit is created, and works as follows. You set the value of the *IMedium.location()* attribute to a location specification which only contains the path specification but not the file name part and ends with either a forward slash or a backslash character. In response, VirtualBox will generate a new UUID for the hard disk and compose the file name using the following pattern:

```
<path>/{<uuid>}.<ext>
```

where <path> is the supplied path specification, <uuid> is the newly generated UUID and <ext> is the default extension for the storage format of this hard disk. After that, you may call any of the methods that create a new hard disk storage unit and they will use the generated UUID and file name.

id_p
Get str value for 'id' UUID of the medium. For a newly created medium, this value is a randomly generated UUID.

For media in one of MediumState_NotCreated, MediumState_Creating or MediumState_Deleting states, the value of this property is undefined and will most likely be an empty UUID.

description
Get or set str value for 'description' Optional description of the medium. For a newly created medium the value of this attribute is an empty string.

Medium types that don't support this attribute will return E_NOTIMPL in attempt to get or set this attribute's value.

For some storage types, reading this attribute may return an outdated (last known) value when *state()* is *MediumState.inaccessible* or *MediumState.locked_write* because the value of this attribute is stored within the storage unit itself. Also note that changing the attribute value is not possible in such case, as well as when the medium is the *MediumState.locked_read* state.

state
Get MediumState value for 'state' Returns the current medium state, which is the last state set by the accessibility check performed by *refresh_state()* . If that method has not yet been called on the medium, the state is "Inaccessible"; as opposed to truly inaccessible media, the value of *last_access_error()* will be an empty string in that case.

As of version 3.1, this no longer performs an accessibility check automatically; call *refresh_state()* for that.

variant
Get MediumVariant value for 'variant' Returns the storage format variant information for this medium as an array of the flags described at *MediumVariant* . Before *refresh_state()* is called this method returns an undefined value.

location
Get str value for 'location' Location of the storage unit holding medium data.

The format of the location string is medium type specific. For medium types using regular files in a host's file system, the location string is the full file name.

name
Get str value for 'name' Name of the storage unit holding medium data.

The returned string is a short version of the *location()* attribute that is suitable for representing the medium in situations where the full location specification is too long (such as lists and comboboxes in GUI frontends). This string is also used by frontends to sort the media list alphabetically when needed.

For example, for locations that are regular files in the host's file system, the value of this attribute is just the file name (+ extension), without the path specification.

Note that as opposed to the *location()* attribute, the name attribute will not necessary be unique for a list of media of the given type and format.

device_type
Get DeviceType value for 'deviceType' Kind of device (DVD/Floppy/HardDisk) which is applicable to this medium.

host_drive
Get bool value for 'hostDrive' True if this corresponds to a drive on the host.

size
Get int value for 'size' Physical size of the storage unit used to hold medium data (in bytes).

For media whose *state()* is *MediumState.inaccessible* , the value of this property is the last known size. For *MediumState.not_created* media, the returned value is zero.

format_p
Get str value for 'format' Storage format of this medium.

The value of this attribute is a string that specifies a backend used to store medium data. The storage format is defined when you create a new medium or automatically detected when you open an existing medium, and cannot be changed later.

The list of all storage formats supported by this VirtualBox installation can be obtained using *ISystemProperties.medium_formats()* .

medium_format
Get IMediumFormat value for 'mediumFormat' Storage medium format object corresponding to this medium.

The value of this attribute is a reference to the medium format object that specifies the backend properties used to store medium data. The storage format is defined when you create a new medium or automatically detected when you open an existing medium, and cannot be changed later.

@c null is returned if there is no associated medium format object. This can e.g. happen for medium objects representing host drives and other special medium objects.

type_p
Get or set MediumType value for 'type' Type (role) of this medium.

The following constraints apply when changing the value of this attribute:

If a medium is attached to a virtual machine (either in the current state or in one of the snapshots), its type cannot be changed.

As long as the medium has children, its type cannot be set to *MediumType.writethrough* .

The type of all differencing media is *MediumType.normal* and cannot be changed.

The type of a newly created or opened medium is set to *MediumType.normal* , except for DVD and floppy media, which have a type of *MediumType.writethrough* .

allowed_types
Get MediumType value for 'allowedTypes' Returns which medium types can selected for this medium.

parent
Get IMedium value for 'parent' Parent of this medium (the medium this medium is directly based on).

Only differencing media have parents. For base (non-differencing) media, @c null is returned.

children
Get IMedium value for 'children' Children of this medium (all differencing media directly based on this medium). A @c null array is returned if this medium does not have any children.

base
Get IMedium value for 'base' Base medium of this medium.

If this is a differencing medium, its base medium is the medium the given medium branch starts from. For all other types of media, this property returns the medium object itself (i.e. the same object this property is read on).

read_only
Get bool value for 'readOnly' Returns @c true if this medium is read-only and @c false otherwise.

A medium is considered to be read-only when its contents cannot be modified without breaking the integrity of other parties that depend on this medium such as its child media or snapshots of virtual machines where this medium is attached to these machines. If there are no children and no such snapshots then there is no dependency and the medium is not read-only.

The value of this attribute can be used to determine the kind of the attachment that will take place when attaching this medium to a virtual machine. If the value is @c false then the medium will be attached directly. If the value is @c true then the medium will be attached indirectly by creating a new differencing child medium for that. See the interface description for more information.

Note that all *MediumType.immutable* Immutable media are always read-only while all *MediumType.writethrough* Writethrough media are always not.

The read-only condition represented by this attribute is related to the medium type and usage, not to the current *IMedium.state()* medium state and not to the read-only state of the storage unit.

logical_size
Get int value for 'logicalSize' Logical size of this medium (in bytes), as reported to the guest OS running inside the virtual machine this medium is attached to. The logical size is defined when the medium is created and cannot be changed later.

For media whose state is *state()* is *MediumState.inaccessible* , the value of this property is the last known logical size. For *MediumState.not_created* media, the returned value is zero.

auto_reset
Get or set bool value for 'autoReset' Whether this differencing medium will be automatically reset each time a virtual machine it is attached to is powered up. This attribute is automatically set to @c true for the last differencing image of an "immutable" medium (see *MediumType*).

See *reset()* for more information about resetting differencing media.

Reading this property on a base (non-differencing) medium will always @c false. Changing the value of this property in this case is not supported.

last_access_error
Get str value for 'lastAccessError' Text message that represents the result of the last accessibility check performed by *refresh_state()* .

An empty string is returned if the last accessibility check was successful or has not yet been called. As a result, if *state()* is "Inaccessible" and this attribute is empty, then *refresh_state()* has yet to be called; this is the default value of media after VirtualBox initialization. A non-empty string indicates a failure and should normally describe a reason of the failure (for example, a file read error).

machine_ids
Get str value for 'machineIds' Array of UUIDs of all machines this medium is attached to.

A @c null array is returned if this medium is not attached to any machine or to any machine's snapshot.

The returned array will include a machine even if this medium is not attached to that machine in the current state but attached to it in one of the machine's snapshots. See *get_snapshot_ids()* for details.

set_ids(*set_image_id*, *image_id*, *set_parent_id*, *parent_id*)
Changes the UUID and parent UUID for a hard disk medium.
in set_image_id of type bool Select whether a new image UUID is set or not.
in image_id of type str New UUID for the image. If an empty string is passed, then a new UUID is automatically created, provided that @a setImageId is @c true. Specifying a zero UUID is not allowed.
in set_parent_id of type bool Select whether a new parent UUID is set or not.
in parent_id of type str New parent UUID for the image. If an empty string is passed, then a new UUID is automatically created, provided @a setParentId is @c true. A zero UUID is valid.
raises *OleErrorInvalidarg* Invalid parameter combination.
raises *VBoxErrorNotSupported* Medium is not a hard disk medium.

refresh_state()
If the current medium state (see *MediumState*) is one of "Created", "Inaccessible" or "LockedRead", then this performs an accessibility check on the medium and sets the value of the *state()* attribute accordingly; that value is also returned for convenience.

For all other state values, this does not perform a refresh but returns the state only.

The refresh, if performed, may take a long time (several seconds or even minutes, depending on the storage unit location and format) because it performs an accessibility check of the storage unit. This check may cause a significant delay if the storage unit of the given medium is, for example, a file located on a network share which is not currently accessible due to connectivity problems. In that case, the call will not return until a timeout interval defined by the host OS

for this operation expires. For this reason, it is recommended to never read this attribute on the main UI thread to avoid making the UI unresponsive.

If the last known state of the medium is "Created" and the accessibility check fails, then the state would be set to "Inaccessible", and *last_access_error()* may be used to get more details about the failure. If the state of the medium is "LockedRead", then it remains the same, and a non-empty value of *last_access_error()* will indicate a failed accessibility check in this case.

Note that not all medium states are applicable to all medium types.
return state of type *MediumState* New medium state.

get_snapshot_ids(*machine_id*)
Returns an array of UUIDs of all snapshots of the given machine where this medium is attached to.

If the medium is attached to the machine in the current state, then the first element in the array will always be the ID of the queried machine (i.e. the value equal to the @c machineId argument), followed by snapshot IDs (if any).

If the medium is not attached to the machine in the current state, then the array will contain only snapshot IDs.

The returned array may be @c null if this medium is not attached to the given machine at all, neither in the current state nor in one of the snapshots.
in machine_id of type str UUID of the machine to query.
return snapshot_ids of type str Array of snapshot UUIDs of the given machine using this medium.

lock_read()
Locks this medium for reading.

A read lock is shared: many clients can simultaneously lock the same medium for reading unless it is already locked for writing (see *lock_write()*) in which case an error is returned.

When the medium is locked for reading, it cannot be modified from within VirtualBox. This means that any method that changes the properties of this medium or contents of the storage unit will return an error (unless explicitly stated otherwise). That includes an attempt to start a virtual machine that wants to write to the medium.

When the virtual machine is started up, it locks for reading all media it uses in read-only mode. If some medium cannot be locked for reading, the startup procedure will fail. A medium is typically locked for reading while it is used by a running virtual machine but has a depending differencing image that receives the actual write operations. This way one base medium can have multiple child differencing images which can be written to simultaneously. Read-only media such as DVD and floppy images are also locked for reading only (so they can be in use by multiple machines simultaneously).

A medium is also locked for reading when it is the source of a write operation such as *clone_to()* or *merge_to()* .

The medium locked for reading must be unlocked by abandoning the returned token object, see *IToken* . Calls to *lock_read()* can be nested and the lock is actually released when all callers have abandoned the token.

This method sets the medium state (see *state()*) to "LockedRead" on success. The medium's previous state must be one of "Created", "Inaccessible" or "LockedRead".

Locking an inaccessible medium is not an error; this method performs a logical lock that prevents modifications of this medium through the VirtualBox API, not a physical

file-system lock of the underlying storage unit.

This method returns the current state of the medium *before* the operation.

return token of type ***IToken*** Token object, when this is released (reference count reaches 0) then the lock count is decreased. The lock is released when the lock count reaches 0.

raises ***VBoxErrorInvalidObjectState*** Invalid medium state (e.g. not created, locked, inaccessible,

creating, deleting).

lock_write()

Locks this medium for writing.

A write lock, as opposed to *lock_read()* , is exclusive: there may be only one client holding a write lock, and there may be no read locks while the write lock is held. As a result, read-locking fails if a write lock is held, and write-locking fails if either a read or another write lock is held.

When a medium is locked for writing, it cannot be modified from within VirtualBox, and it is not guaranteed that the values of its properties are up-to-date. Any method that changes the properties of this medium or contents of the storage unit will return an error (unless explicitly stated otherwise).

When a virtual machine is started up, it locks for writing all media it uses to write data to. If any medium could not be locked for writing, the startup procedure will fail. If a medium has differencing images, then while the machine is running, only the last ("leaf") differencing image is locked for writing, whereas its parents are locked for reading only.

A medium is also locked for writing when it is the target of a write operation such as *clone_to()* or *merge_to()* .

The medium locked for writing must be unlocked by abandoning the returned token object, see *IToken* . Write locks *cannot* be nested.

This method sets the medium state (see *state()*) to "LockedWrite" on success. The medium's previous state must be either "Created" or "Inaccessible".

Locking an inaccessible medium is not an error; this method performs a logical lock that prevents modifications of this medium through the VirtualBox API, not a physical file-system lock of the underlying storage unit.

return token of type ***IToken*** Token object, when this is released (reference count reaches 0) then the lock is released.

raises ***VBoxErrorInvalidObjectState*** Invalid medium state (e.g. not created, locked, inaccessible,

creating, deleting).

close()

Closes this medium.

The medium must not be attached to any known virtual machine and must not have any known child media, otherwise the operation will fail.

When the medium is successfully closed, it is removed from the list of registered media, but its storage unit is not deleted. In particular, this means that this medium can later be opened again using the *IVirtualBox.open_medium()* call.

Note that after this method successfully returns, the given medium object becomes uninitialized. This means that any attempt to call any of its methods or attributes will fail with the "Object not ready" (E_ACCESSDENIED) error.

raises *VBoxErrorInvalidObjectState* Invalid medium state (other than not created, created or
inaccessible).
raises *VBoxErrorObjectInUse* Medium attached to virtual machine.
raises *VBoxErrorFileError* Settings file not accessible.
raises *VBoxErrorXmlError* Could not parse the settings file.

get_property(*name*)
Returns the value of the custom medium property with the given name.

The list of all properties supported by the given medium format can be obtained with *IMediumFormat.describe_properties()*.

If this method returns an empty string in @a value, the requested property is supported but currently not assigned any value.
in name of type str Name of the property to get.
return value of type str Current property value.
raises *VBoxErrorObjectNotFound* Requested property does not exist (not supported by the format).
raises *OleErrorInvalidarg* @a name is @c null or empty.

set_property(*name*, *value*)
Sets the value of the custom medium property with the given name.

The list of all properties supported by the given medium format can be obtained with *IMediumFormat.describe_properties()*.

Setting the property value to @c null or an empty string is equivalent to deleting the existing value. A default value (if it is defined for this property) will be used by the format backend in this case.
in name of type str Name of the property to set.
in value of type str Property value to set.
raises *VBoxErrorObjectNotFound* Requested property does not exist (not supported by the format).
raises *OleErrorInvalidarg* @a name is @c null or empty.

get_properties(*names*)
Returns values for a group of properties in one call.

The names of the properties to get are specified using the @a names argument which is a list of comma-separated property names or an empty string if all properties are to be returned. Currently the value of this argument is ignored and the method always returns all existing properties.

The list of all properties supported by the given medium format can be obtained with *IMediumFormat.describe_properties()*.

The method returns two arrays, the array of property names corresponding to the @a names argument and the current values of these properties. Both arrays have the same number of elements with each element at the given index in the first array corresponds to an element at the same index in the second array.

For properties that do not have assigned values, an empty string is returned at the appropriate index in the @a returnValues array.
in names of type str Names of properties to get.
out return_names of type str Names of returned properties.
return return_values of type str Values of returned properties.

set_properties(*names*, *values*)
Sets values for a group of properties in one call.

The names of the properties to set are passed in the @a names array along with the new values for them in the @a values array. Both arrays have the same number of elements with each element at the given index in the first array corresponding to an element at the same index in the second array.

If there is at least one property name in @a names that is not valid, the method will fail before changing the values of any other properties from the @a names array.

Using this method over *set_property()* is preferred if you need to set several properties at once since it is more efficient.

The list of all properties supported by the given medium format can be obtained with *IMediumFormat.describe_properties()* .

Setting the property value to @c null or an empty string is equivalent to deleting the existing value. A default value (if it is defined for this property) will be used by the format backend in this case.

in names of type str Names of properties to set.

in values of type str Values of properties to set.

create_base_storage(*logical_size*, *variant*)

Starts creating a hard disk storage unit (fixed/dynamic, according to the variant flags) in in the background. The previous storage unit created for this object, if any, must first be deleted using *delete_storage()* , otherwise the operation will fail.

Before the operation starts, the medium is placed in *MediumState.creating* state. If the create operation fails, the medium will be placed back in *MediumState.not_created* state.

After the returned progress object reports that the operation has successfully completed, the medium state will be set to *MediumState.created* , the medium will be remembered by this VirtualBox installation and may be attached to virtual machines.

in logical_size of type int Maximum logical size of the medium in bytes.

in variant of type *MediumVariant* Exact image variant which should be created (as a combination of *MediumVariant* flags).

return progress of type *IProgress* Progress object to track the operation completion.

raises *VBoxErrorNotSupported* The variant of storage creation operation is not supported. See

delete_storage()

Starts deleting the storage unit of this medium.

The medium must not be attached to any known virtual machine and must not have any known child media, otherwise the operation will fail. It will also fail if there is no storage unit to delete or if deletion is already in progress, or if the medium is being in use (locked for read or for write) or inaccessible. Therefore, the only valid state for this operation to succeed is *MediumState.created* .

Before the operation starts, the medium is placed in *MediumState.deleting* state and gets removed from the list of remembered hard disks (media registry). If the delete operation fails, the medium will be remembered again and placed back to *MediumState.created* state.

After the returned progress object reports that the operation is complete, the medium state will be set to *MediumState.not_created* and you will be able to use one of the storage creation methods to create it again.

close()

If the deletion operation fails, it is not guaranteed that the storage unit still exists. You may check the *IMedium.state()* value to answer this question.

return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorObjectInUse* Medium is attached to a virtual machine.
raises *VBoxErrorNotSupported* Storage deletion is not allowed because neither of storage creation
operations are supported. See

create_diff_storage(*target*, *variant*)
Starts creating an empty differencing storage unit based on this medium in the format and at the location defined by the @a target argument.

The target medium must be in *MediumState.not_created* state (i.e. must not have an existing storage unit). Upon successful completion, this operation will set the type of the target medium to *MediumType.normal* and create a storage unit necessary to represent the differencing medium data in the given format (according to the storage format of the target object).

After the returned progress object reports that the operation is successfully complete, the target medium gets remembered by this VirtualBox installation and may be attached to virtual machines.

The medium will be set to *MediumState.locked_read* state for the duration of this operation.
in target of type *IMedium* Target medium.
in variant of type *MediumVariant* Exact image variant which should be created (as a combination of *MediumVariant* flags).
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorObjectInUse* Medium not in @c NotCreated state.

merge_to(*target*)
Starts merging the contents of this medium and all intermediate differencing media in the chain to the given target medium.

The target medium must be either a descendant of this medium or its ancestor (otherwise this method will immediately return a failure). It follows that there are two logical directions of the merge operation: from ancestor to descendant (*forward merge*) and from descendant to ancestor (*backward merge*). Let us consider the following medium chain:

Base <- Diff_1 <- Diff_2

Here, calling this method on the Base medium object with Diff_2 as an argument will be a forward merge; calling it on Diff_2 with Base as an argument will be a backward merge. Note that in both cases the contents of the resulting medium will be the same, the only difference is the medium object that takes the result of the merge operation. In case of the forward merge in the above example, the result will be written to Diff_2; in case of the backward merge, the result will be written to Base. In other words, the result of the operation is always stored in the target medium.

Upon successful operation completion, the storage units of all media in the chain between this (source) medium and the target medium, including the source medium itself, will be automatically deleted and the relevant medium objects (including this medium) will become uninitialized. This means that any attempt to call any of their methods or attributes will fail with the "Object not ready" (E_ACCESSDENIED) error. Applied to the above example, the forward merge of Base to Diff_2 will delete and uninitialize both Base and Diff_1 media. Note that Diff_2 in this case will become a base medium itself since it will no longer be based on any other medium.

Considering the above, all of the following conditions must be met in order for the merge operation to succeed:

Neither this (source) medium nor any intermediate differencing medium in the chain between it and the target medium is attached to any virtual machine.

Neither the source medium nor the target medium is an *MediumType.immutable* medium.

The part of the medium tree from the source medium to the target medium is a linear chain, i.e. all medium in this chain have exactly one child which is the next medium in this chain. The only exception from this rule is the target medium in the forward merge operation; it is allowed to have any number of child media because the merge operation will not change its logical contents (as it is seen by the guest OS or by children).

None of the involved media are in *MediumState.locked_read* or *MediumState.locked_write* state.

This (source) medium and all intermediates will be placed to *MediumState.deleting* state and the target medium will be placed to *MediumState.locked_write* state and for the duration of this operation.
in target of type *IMedium* Target medium.
return progress of type *IProgress* Progress object to track the operation completion.

clone_to (*target*, *variant*, *parent*)
Starts creating a clone of this medium in the format and at the location defined by the @a target argument.

The target medium must be either in *MediumState.not_created* state (i.e. must not have an existing storage unit) or in *MediumState.created* state (i.e. created and not locked, and big enough to hold the data or else the copy will be partial). Upon successful completion, the cloned medium will contain exactly the same sector data as the medium being cloned, except that in the first case a new UUID for the clone will be randomly generated, and in the second case the UUID will remain unchanged.

The @a parent argument defines which medium will be the parent of the clone. Passing a @c null reference indicates that the clone will be a base image, i.e. completely independent. It is possible to specify an arbitrary medium for this parameter, including the parent of the medium which is being cloned. Even cloning to a child of the source medium is possible. Note that when cloning to an existing image, the @a parent argument is ignored.

After the returned progress object reports that the operation is successfully complete, the target medium gets remembered by this VirtualBox installation and may be attached to virtual machines.

This medium will be placed to *MediumState.locked_read* state for the duration of this operation.
in target of type *IMedium* Target medium.
in variant of type *MediumVariant* Exact image variant which should be created (as a combination of *MediumVariant* flags).
in parent of type *IMedium* Parent of the cloned medium.
return progress of type *IProgress* Progress object to track the operation completion.
raises *OleErrorNotimpl* The specified cloning variant is not supported at the moment.

clone_to_base (*target*, *variant*)
Starts creating a clone of this medium in the format and at the location defined by the @a target argument.

The target medium must be either in *MediumState.not_created* state (i.e. must not have an existing storage unit) or in *MediumState.created* state (i.e. created and not locked, and big enough to hold the data or else the copy will be partial). Upon successful completion, the cloned medium will contain exactly the same sector data as the medium being cloned, except

that in the first case a new UUID for the clone will be randomly generated, and in the second case the UUID will remain unchanged.

The @a parent argument defines which medium will be the parent of the clone. In this case the clone will be a base image, i.e. completely independent. It is possible to specify an arbitrary medium for this parameter, including the parent of the medium which is being cloned. Even cloning to a child of the source medium is possible. Note that when cloning to an existing image, the @a parent argument is ignored.

After the returned progress object reports that the operation is successfully complete, the target medium gets remembered by this VirtualBox installation and may be attached to virtual machines.

This medium will be placed to *MediumState.locked_read* state for the duration of this operation.
in target of type *IMedium* Target medium.
in variant of type *MediumVariant* *MediumVariant* flags).
return progress of type *IProgress* Progress object to track the operation completion.
raises *OleErrorNotimpl* The specified cloning variant is not supported at the moment.

set_location(*location*)
Changes the location of this medium. Some medium types may support changing the storage unit location by simply changing the value of the associated property. In this case the operation is performed immediately, and @a progress is returning a @c null reference. Otherwise on success there is a progress object returned, which signals progress and completion of the operation. This distinction is necessary because for some formats the operation is very fast, while for others it can be very slow (moving the image file by copying all data), and in the former case it'd be a waste of resources to create a progress object which will immediately signal completion.

When setting a location for a medium which corresponds to a/several regular file(s) in the host's file system, the given file name may be either relative to the *IVirtualBox.home_folder()* VirtualBox home folder or absolute. Note that if the given location specification does not contain the file extension part then a proper default extension will be automatically appended by the implementation depending on the medium type.
in location of type str New location.
return progress of type *IProgress* Progress object to track the operation completion.
raises *OleErrorNotimpl* The operation is not implemented yet.
raises *VBoxErrorNotSupported* Medium format does not support changing the location.

compact()
Starts compacting of this medium. This means that the medium is transformed into a possibly more compact storage representation. This potentially creates temporary images, which can require a substantial amount of additional disk space.

This medium will be placed to *MediumState.locked_write* state and all its parent media (if any) will be placed to *MediumState.locked_read* state for the duration of this operation.

Please note that the results can be either returned straight away, or later as the result of the background operation via the object returned via the @a progress parameter.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorNotSupported* Medium format does not support compacting (but potentially
needs it).

resize(*logical_size*)
Starts resizing this medium. This means that the nominal size of the medium is set to the new value. Both increasing and decreasing the size is possible, and there are no safety checks, since

VirtualBox does not make any assumptions about the medium contents.

Resizing usually needs additional disk space, and possibly also some temporary disk space. Note that resize does not create a full temporary copy of the medium, so the additional disk space requirement is usually much lower than using the clone operation.

This medium will be placed to *MediumState.locked_write* state for the duration of this operation.

Please note that the results can be either returned straight away, or later as the result of the background operation via the object returned via the @a progress parameter.
in logical_size of type int New nominal capacity of the medium in bytes.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorNotSupported* Medium format does not support resizing.

reset()
Starts erasing the contents of this differencing medium.

This operation will reset the differencing medium to its initial state when it does not contain any sector data and any read operation is redirected to its parent medium. This automatically gets called during VM power-up for every medium whose *auto_reset()* attribute is @c true.

The medium will be write-locked for the duration of this operation (see *lock_write()*).
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorNotSupported* This is not a differencing medium.
raises *VBoxErrorInvalidObjectState* Medium is not in

change_encryption(*current_password*, *cipher*, *new_password*, *new_password_id*)
Starts encryption of this medium. This means that the stored data in the medium is encrypted.

This medium will be placed to *MediumState.locked_write* state.

Please note that the results can be either returned straight away, or later as the result of the background operation via the object returned via the @a progress parameter.
in current_password of type str The current password the medium is protected with. Use an empty string to indicate that the medium isn't encrypted.
in cipher of type str The cipher to use for encryption. An empty string indicates no encryption for the result.
in new_password of type str The new password the medium should be protected with. An empty password and password ID will result in the medium being encrypted with the current password.
in new_password_id of type str The ID of the new password when unlocking the medium.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorNotSupported* Encryption is not supported for this medium because it is attached to more than one VM
or has children.

get_encryption_settings()
Returns the encryption settings for this medium.
out cipher of type str The cipher used for encryption.
return password_id of type str The ID of the password when unlocking the medium.
raises *VBoxErrorNotSupported* Encryption is not configured for this medium.

check_encryption_password(*password*)
Checks whether the supplied password is correct for the medium.
in password of type str The password to check.
raises *VBoxErrorNotSupported* Encryption is not configured for this medium.

raises *VBoxErrorPasswordIncorrect* The given password is incorrect.

class virtualbox.library.**IMediumFormat**(*interface=None*)

The IMediumFormat interface represents a medium format.

Each medium format has an associated backend which is used to handle media stored in this format. This interface provides information about the properties of the associated backend.

Each medium format is identified by a string represented by the *id_p()* attribute. This string is used in calls like *IVirtualBox.create_medium()* to specify the desired format.

The list of all supported medium formats can be obtained using *ISystemProperties.medium_formats()* .

IMedium

id_p

Get str value for 'id' Identifier of this format.

The format identifier is a non-@c null non-empty ASCII string. Note that this string is case-insensitive. This means that, for example, all of the following strings:

```
"VDI"
"vdi"
"VdI"
```

refer to the same medium format.

This string is used in methods of other interfaces where it is necessary to specify a medium format, such as *IVirtualBox.create_medium()* .

name

Get str value for 'name' Human readable description of this format.

Mainly for use in file open dialogs.

capabilities

Get MediumFormatCapabilities value for 'capabilities' Capabilities of the format as an array of the flags.

For the meaning of individual capability flags see *MediumFormatCapabilities* .

describe_file_extensions()

Returns two arrays describing the supported file extensions.

The first array contains the supported extensions and the seconds one the type each extension supports. Both have the same size.

Note that some backends do not work on files, so this array may be empty.

IMediumFormat.capabilities()

out extensions of type str The array of supported extensions.

out types of type *DeviceType* The array which indicates the device type for every given extension.

describe_properties()

Returns several arrays describing the properties supported by this format.

An element with the given index in each array describes one property. Thus, the number of elements in each returned array is the same and corresponds to the number of supported properties.

The returned arrays are filled in only if the *MediumFormatCapabilities.properties* flag is set. All arguments must be non-@c null.

DataType, *DataFlags*
out names of type str Array of property names.
out descriptions of type str Array of property descriptions.
out types of type *DataType* Array of property types.
out flags of type int Array of property flags.
out defaults of type str Array of default property values.

class virtualbox.library.**IToken**(*interface=None*)
The IToken interface represents a token passed to an API client, which triggers cleanup actions when it is explicitly released by calling the *abandon()* method (preferred, as it is accurately defined when the release happens), or when the object reference count drops to 0. The latter way is implicitly used when an API client crashes, however the discovery that there was a crash can take rather long, depending on the platform (COM needs 6 minutes). So better don't rely on the crash behavior too much.

abandon()
Releases this token. Cannot be undone in any way, and makes the token object unusable (even the *dummy()* method will return an error), ready for releasing. It is a more defined way than just letting the reference count drop to 0, because the latter (depending on the platform) can trigger asynchronous cleanup activity.

dummy()
Purely a NOOP. Useful when using proxy type API bindings (e.g. the webservice) which manage objects on behalf of the actual client, using an object reference expiration time based garbage collector.

class virtualbox.library.**IMousePointerShape**(*interface=None*)
The guest mouse pointer description.

visible
Get bool value for 'visible' Flag whether the pointer is visible.

alpha
Get bool value for 'alpha' Flag whether the pointer has an alpha channel.

hot_x
Get int value for 'hotX' The pointer hot spot X coordinate.

hot_y
Get int value for 'hotY' The pointer hot spot Y coordinate.

width
Get int value for 'width' Width of the pointer shape in pixels.

height
Get int value for 'height' Height of the pointer shape in pixels.

shape
Get str value for 'shape' Shape bitmaps.

The @a shape buffer contains a 1bpp (bits per pixel) AND mask followed by a 32bpp XOR (color) mask.

For pointers without alpha channel the XOR mask pixels are 32 bit values: (lsb)BGR0(msb). For pointers with alpha channel the XOR mask consists of (lsb)BGRA(msb) 32 bit values.

An AND mask is provided for pointers with alpha channel, so if the client does not support alpha, the pointer could be displayed as a normal color pointer.

The AND mask is a 1bpp bitmap with byte aligned scanlines. The size of the AND mask therefore is cbAnd = (width + 7) / 8 * height. The padding bits at the end of each scanline are

undefined.

The XOR mask follows the AND mask on the next 4-byte aligned offset: uint8_t *pu8Xor = pu8And + (cbAnd + 3) & ~3. Bytes in the gap between the AND and the XOR mask are undefined. The XOR mask scanlines have no gap between them and the size of the XOR mask is: cbXor = width * 4 * height.

If @a shape size is 0, then the shape is not known or did not change. This can happen if only the pointer visibility is changed.

class virtualbox.library.**IDisplaySourceBitmap**(*interface=None*)

Information about the screen bitmap.

screen_id

Get int value for 'screenId'

query_bitmap_info()

Information about the screen bitmap.

out address of type str

out width of type int

out height of type int

out bits_per_pixel of type int

out bytes_per_line of type int

out bitmap_format of type *BitmapFormat*

class virtualbox.library.**IFramebuffer**(*interface=None*)

Frame buffer width, in pixels.

width

Get int value for 'width' Frame buffer width, in pixels.

height

Get int value for 'height' Frame buffer height, in pixels.

bits_per_pixel

Get int value for 'bitsPerPixel' Color depth, in bits per pixel.

bytes_per_line

Get int value for 'bytesPerLine' Scan line size, in bytes.

pixel_format

Get BitmapFormat value for 'pixelFormat' Frame buffer pixel format. It's one of the values defined by *BitmapFormat* .

This attribute must never (and will never) return *BitmapFormat.opaque* – the format of the frame buffer must be always known.

height_reduction

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

overlay

Get IFramebufferOverlay value for 'overlay' An alpha-blended overlay which is superposed over the frame buffer. The initial purpose is to allow the display of icons providing information about the VM state, including disk activity, in front ends which do not have other means of doing that. The overlay is designed to controlled exclusively by IDisplay. It has no locking of

its own, and any changes made to it are not guaranteed to be visible until the affected portion of IFramebuffer is updated. The overlay can be created lazily the first time it is requested. This attribute can also return @c null to signal that the overlay is not implemented.

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

capabilities
Get FramebufferCapabilities value for 'capabilities' Capabilities of the framebuffer instance.

For the meaning of individual capability flags see *FramebufferCapabilities*.

notify_update(*x*, *y*, *width*, *height*)
Informs about an update. Gets called by the display object where this buffer is registered.

in x of type int

in y of type int

in width of type int

in height of type int

notify_update_image(*x*, *y*, *width*, *height*, *image*)
Informs about an update and provides 32bpp bitmap.

in x of type int

in y of type int

in width of type int

in height of type int
in image of type str Array with 32BPP image data.

notify_change(*screen_id*, *x_origin*, *y_origin*, *width*, *height*)
Requests a size change.
in screen_id of type int Logical guest screen number.
in x_origin of type int Location of the screen in the guest.
in y_origin of type int Location of the screen in the guest.
in width of type int Width of the guest display, in pixels.
in height of type int Height of the guest display, in pixels.

video_mode_supported(*width*, *height*, *bpp*)
Returns whether the frame buffer implementation is willing to support a given video mode. In case it is not able to render the video mode (or for some reason not willing), it should return @c 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.

in width of type int

in height of type int

in bpp of type int

return supported of type bool

get_visible_region(*rectangles*, *count*)
Returns the visible region of this frame buffer.

If the @a rectangles parameter is @c 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 @c 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.

The address of the provided array must be in the process space of this IFramebuffer object.

Method not yet implemented.
in rectangles of type str Pointer to the @c RTRECT array to receive region data.
in count of type int Number of @c RTRECT elements in the @a rectangles array.
return count_copied of type int Number of elements copied to the @a rectangles array.

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

The address of the provided array must be in the process space of this IFramebuffer object.

The IFramebuffer implementation must make a copy of the provided array of rectangles.

Method not yet implemented.
in rectangles of type str Pointer to the @c RTRECT array.
in count of type int Number of @c RTRECT elements in the @a rectangles array.

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

The address of the provided command must be in the process space of this IFramebuffer object.
in command of type str Pointer to VBOXVHWACMD containing the command to execute.

notify3_d_event(*type_p*, *data*)
Notifies framebuffer about 3D backend event.
in type_p of type int event type. Currently only VBOX3D_NOTIFY_EVENT_TYPE_VISIBLE_3DDATA is supported.
in data of type str event-specific data, depends on the supplied event type

class virtualbox.library.**IFramebufferOverlay**(*interface=None*)
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.

x
Get int value for 'x' X position of the overlay, relative to the frame buffer.

y
Get int value for 'y' Y position of the overlay, relative to the frame buffer.

visible
Get or set bool value for 'visible' Whether the overlay is currently visible.

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

move(*x*, *y*)
Changes the overlay's position relative to the IFramebuffer.

in x of type int

in y of type int

class virtualbox.library.**IDisplay**(*interface=None*)
The IDisplay interface represents the virtual machine's display.

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

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

guest_screen_layout
Get IGuestScreenInfo value for 'guestScreenLayout' Layout of the guest screens.

get_screen_resolution(*screen_id*)
Queries certain attributes such as display width, height, color depth and the X and Y origin for a given guest screen.

The parameters @a xOrigin and @a yOrigin return the X and Y coordinates of the framebuffer's origin.

All return parameters are optional.

in screen_id of type int

out width of type int

out height of type int

out bits_per_pixel of type int

out x_origin of type int

out y_origin of type int

out guest_monitor_status of type *GuestMonitorStatus*

attach_framebuffer(*screen_id*, *framebuffer*)
Sets the graphics update target for a screen.

in screen_id of type int

in framebuffer of type *IFramebuffer*

return id_p of type str

detach_framebuffer(*screen_id*, *id_p*)
Removes the graphics updates target for a screen.

in screen_id of type int

in id_p of type str

query_framebuffer(*screen_id*)
Queries the graphics updates targets for a screen.

in screen_id of type int

return framebuffer of type *IFramebuffer*

set_video_mode_hint(*display*, *enabled*, *change_origin*, *origin_x*, *origin_y*, *width*, *height*, *bits_per_pixel*)

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 @c 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: @c 0 is the primary display, @c 1 is the first secondary and so on. If the multi-monitor configuration is not supported, @a display must be @c 0.

in display of type int The number of the guest display to send the hint to.
in enabled of type bool @c True, if this guest screen is enabled, @c False otherwise.
in change_origin of type bool @c True, if the origin of the guest screen should be changed, @c False otherwise.
in origin_x of type int The X origin of the guest screen.
in origin_y of type int The Y origin of the guest screen.
in width of type int The width of the guest screen.
in height of type int The height of the guest screen.
in bits_per_pixel of type int The number of bits per pixel of the guest screen.
raises ***OleErrorInvalidarg*** The @a display is not associated with any monitor.

set_seamless_mode(*enabled*)

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

Calling this method has no effect if *IGuest.get_facility_status()* with facility @c Seamless does not return @c Active.

in enabled of type bool

take_screen_shot(*screen_id*, *address*, *width*, *height*, *bitmap_format*)

Takes a screen shot of the requested size and format and copies it to the buffer allocated by the caller and pointed to by @a address. The buffer size must be enough for a 32 bits per pixel bitmap, i.e. width * height * 4 bytes.

This API can be used only locally by a VM process through the COM/XPCOM C++ API as it requires pointer support. It is not available for scripting languages, Java or any webservice clients. Unless you are writing a new VM frontend use *take_screen_shot_to_array()* .

in screen_id of type int

in address of type str

in width of type int

in height of type int

in bitmap_format of type *BitmapFormat*

take_screen_shot_to_array(*screen_id*, *width*, *height*, *bitmap_format*)

Takes a guest screen shot of the requested size and format and returns it as an array of bytes.
in screen_id of type int The guest monitor to take screenshot from.
in width of type int Desired image width.
in height of type int Desired image height.
in bitmap_format of type ***BitmapFormat*** The requested format.

return screen_data of type str Array with resulting screen data.

draw_to_screen(*screen_id*, *address*, *x*, *y*, *width*, *height*)
Draws a 32-bpp image of the specified size from the given buffer to the given point on the VM display.
in screen_id of type int Monitor to take the screenshot from.
in address of type str Address to store the screenshot to
in x of type int Relative to the screen top left corner.
in y of type int Relative to the screen top left corner.
in width of type int Desired image width.
in height of type int Desired image height.
raises ***OleErrorNotimpl*** Feature not implemented.
raises ***VBoxErrorIprtError*** Could not draw to screen.

invalidate_and_update()
Does a full invalidation of the VM display and instructs the VM to update it.
raises ***VBoxErrorIprtError*** Could not invalidate and update screen.

invalidate_and_update_screen(*screen_id*)
Redraw the specified VM screen.
in screen_id of type int The guest screen to redraw.

complete_vhwa_command(*command*)
Signals that the Video HW Acceleration command has completed.
in command of type str Pointer to VBOXVHWACMD containing the completed command.

viewport_changed(*screen_id*, *x*, *y*, *width*, *height*)
Signals that framebuffer window viewport has changed.
in screen_id of type int Monitor to take the screenshot from.
in x of type int Framebuffer x offset.
in y of type int Framebuffer y offset.
in width of type int Viewport width.
in height of type int Viewport height.
raises ***OleErrorInvalidarg*** The specified viewport data is invalid.

query_source_bitmap(*screen_id*)
Obtains the guest screen bitmap parameters.

in screen_id of type int

out display_source_bitmap of type *IDisplaySourceBitmap*

notify_scale_factor_change(*screen_id*, *u32_scale_factor_w_multiplied*, *u32_scale_factor_h_multiplied*)
Notify OpenGL HGCM host service about graphics content scaling factor change.

in screen_id of type int

in u32_scale_factor_w_multiplied of type int

in u32_scale_factor_h_multiplied of type int

notify_hi_dpi_output_policy_change(*f_unscaled_hi_dpi*)
Notify OpenGL HGCM host service about HiDPI monitor scaling policy change.

in f_unscaled_hi_dpi of type bool

set_screen_layout(*screen_layout_mode*, *guest_screen_info*)
Set video modes for the guest screens.

in screen_layout_mode of type *ScreenLayoutMode*

in guest_screen_info of type `IGuestScreenInfo`

class virtualbox.library.**INetworkAdapter**(*interface=None*)

Represents a virtual network adapter that is attached to a virtual machine. Each virtual machine has a fixed number of network adapter slots with one instance of this attached to each of them. Call *IMachine.get_network_adapter()* to get the network adapter that is attached to a given slot in a given machine.

Each network adapter can be in one of five attachment modes, which are represented by the *NetworkAttachmentType* enumeration; see the *attachment_type()* attribute.

adapter_type

Get or set NetworkAdapterType value for 'adapterType' Type of the virtual network adapter. Depending on this value, VirtualBox will provide a different virtual network hardware to the guest.

slot

Get int value for 'slot' Slot number this adapter is plugged into. Corresponds to the value you pass to *IMachine.get_network_adapter()* to obtain this instance.

enabled

Get or set bool value for '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.

mac_address

Get or set str value for 'MACAddress' Ethernet MAC address of the adapter, 12 hexadecimal characters. When setting it to @c null or an empty string for an enabled adapter, VirtualBox will generate a unique MAC address. Disabled adapters can have an empty MAC address.

attachment_type

Get or set NetworkAttachmentType value for 'attachmentType' Sets/Gets network attachment type of this network adapter.

bridged_interface

Get or set str value for 'bridgedInterface' Name of the network interface the VM should be bridged to.

host_only_interface

Get or set str value for 'hostOnlyInterface' Name of the host only network interface the VM is attached to.

internal_network

Get or set str value for 'internalNetwork' Name of the internal network the VM is attached to.

nat_network

Get or set str value for 'NATNetwork' Name of the NAT network the VM is attached to.

generic_driver

Get or set str value for 'genericDriver' Name of the driver to use for the "Generic" network attachment type.

cable_connected

Get or set bool value for 'cableConnected' Flag whether the adapter reports the cable as connected or not. It can be used to report offline situations to a VM.

line_speed

Get or set int value for 'lineSpeed' Line speed reported by custom drivers, in units of 1 kbps.

promisc_mode_policy

Get or set NetworkAdapterPromiscModePolicy value for 'promiscModePolicy' The promiscuous mode policy of the network adapter when attached to an internal network, host only network or a bridge.

trace_enabled
Get or set bool value for 'traceEnabled' Flag whether network traffic from/to the network card should be traced. Can only be toggled when the VM is turned off.

trace_file
Get or set str value for 'traceFile' Filename where a network trace will be stored. If not set, VBox-pid.pcap will be used.

nat_engine
Get INATEngine value for 'NATEngine' Points to the NAT engine which handles the network address translation for this interface. This is active only when the interface actually uses NAT.

boot_priority
Get or set int value for 'bootPriority' Network boot priority of the adapter. Priority 1 is highest. If not set, the priority is considered to be at the lowest possible setting.

bandwidth_group
Get or set IBandwidthGroup value for 'bandwidthGroup' The bandwidth group this network adapter is assigned to.

get_property(*key*)
Returns the value of the network attachment property with the given name.

If the requested data @a key does not exist, this function will succeed and return an empty string in the @a value argument.
in key of type str Name of the property to get.
return value of type str Current property value.
raises ***OleErrorInvalidarg*** @a name is @c null or empty.

set_property(*key*, *value*)
Sets the value of the network attachment property with the given name.

Setting the property value to @c null or an empty string is equivalent to deleting the existing value.
in key of type str Name of the property to set.
in value of type str Property value to set.
raises ***OleErrorInvalidarg*** @a name is @c null or empty.

get_properties(*names*)
Returns values for a group of properties in one call.

The names of the properties to get are specified using the @a names argument which is a list of comma-separated property names or an empty string if all properties are to be returned. Currently the value of this argument is ignored and the method always returns all existing properties.

The method returns two arrays, the array of property names corresponding to the @a names argument and the current values of these properties. Both arrays have the same number of elements with each element at the given index in the first array corresponds to an element at the same index in the second array.
in names of type str Names of properties to get.
out return_names of type str Names of returned properties.
return return_values of type str Values of returned properties.

class virtualbox.library.**ISerialPort**(*interface=None*)
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.

IMachine.get_serial_port()

slot
Get int value for 'slot' Slot number this serial port is plugged into. Corresponds to the value you pass to *IMachine.get_serial_port()* to obtain this instance.

enabled
Get or set bool value for 'enabled' Flag whether the serial port is enabled. If disabled, the serial port will not be reported to the guest OS.

io_base
Get or set int value for 'IOBase' Base I/O address of the serial port.

irq
Get or set int value for 'IRQ' IRQ number of the serial port.

host_mode
Get or set PortMode value for 'hostMode' How is this port connected to the host.

Changing this attribute may fail if the conditions for *path()* are not met.

server
Get or set bool value for '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 *host_mode()* is PortMode_HostPipe or PortMode_TCP.

path
Get or set str value for 'path' Path to the serial port's pipe on the host when *ISerialPort.host_mode()* is PortMode_HostPipe, the host serial device name when *ISerialPort.host_mode()* is PortMode_HostDevice or the TCP **port** (server) or **hostname:port** (client) when *ISerialPort.host_mode()* is PortMode_TCP. For those cases, setting a @c null or empty string as the attribute's value is an error. Otherwise, the value of this property is ignored.

class virtualbox.library.**IParallelPort**(*interface=None*)
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.

IMachine.get_parallel_port()

slot
Get int value for 'slot' Slot number this parallel port is plugged into. Corresponds to the value you pass to *IMachine.get_parallel_port()* to obtain this instance.

enabled
Get or set bool value for 'enabled' Flag whether the parallel port is enabled. If disabled, the parallel port will not be reported to the guest OS.

io_base
Get or set int value for 'IOBase' Base I/O address of the parallel port.

irq
Get or set int value for 'IRQ' IRQ number of the parallel port.

path
Get or set str value for '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 in the parallel port behaving as if not connected to any device.

class virtualbox.library.**IMachineDebugger**(*interface=None*)
Takes a core dump of the guest.

See include/VBox/dbgfcorefmt.h for details on the file format.

dump_guest_core(*filename*, *compression*)
Takes a core dump of the guest.

See include/VBox/dbgfcorefmt.h for details on the file format.
in filename of type str The name of the output file. The file must not exist.
in compression of type str Reserved for future compression method indicator.

dump_host_process_core(*filename*, *compression*)
Takes a core dump of the VM process on the host.

This feature is not implemented in the 4.0.0 release but it may show up in a dot release.
in filename of type str The name of the output file. The file must not exist.
in compression of type str Reserved for future compression method indicator.

info(*name*, *args*)
Interfaces with the info dumpers (DBGFInfo).

This feature is not implemented in the 4.0.0 release but it may show up in a dot release.
in name of type str The name of the info item.
in args of type str Arguments to the info dumper.
return info of type str The into string.

inject_nmi()
Inject an NMI into a running VT-x/AMD-V VM.

modify_log_groups(*settings*)
Modifies the group settings of the debug or release logger.
in settings of type str The group settings string. See iprt/log.h for details. To target the release logger, prefix the string with "release:".

modify_log_flags(*settings*)
Modifies the debug or release logger flags.
in settings of type str The flags settings string. See iprt/log.h for details. To target the release logger, prefix the string with "release:".

modify_log_destinations(*settings*)
Modifies the debug or release logger destinations.
in settings of type str The destination settings string. See iprt/log.h for details. To target the release logger, prefix the string with "release:".

read_physical_memory(*address*, *size*)
Reads guest physical memory, no side effects (MMIO++).

This feature is not implemented in the 4.0.0 release but may show up in a dot release.
in address of type int The guest physical address.
in size of type int The number of bytes to read.
return bytes_p of type str The bytes read.

write_physical_memory(*address*, *size*, *bytes_p*)
Writes guest physical memory, access handles (MMIO++) are ignored.

This feature is not implemented in the 4.0.0 release but may show up in a dot release.
in address of type int The guest physical address.
in size of type int The number of bytes to read.
in bytes_p of type str The bytes to write.

read_virtual_memory(*cpu_id*, *address*, *size*)
Reads guest virtual memory, no side effects (MMIO++).

This feature is not implemented in the 4.0.0 release but may show up in a dot release.
in cpu_id of type int The identifier of the Virtual CPU.
in address of type int The guest virtual address.
in size of type int The number of bytes to read.
return bytes_p of type str The bytes read.

write_virtual_memory(*cpu_id*, *address*, *size*, *bytes_p*)
Writes guest virtual memory, access handles (MMIO++) are ignored.

This feature is not implemented in the 4.0.0 release but may show up in a dot release.
in cpu_id of type int The identifier of the Virtual CPU.
in address of type int The guest virtual address.
in size of type int The number of bytes to read.
in bytes_p of type str The bytes to write.

load_plug_in(*name*)
Loads a DBGF plug-in.
in name of type str The plug-in name or DLL. Special name 'all' loads all installed plug-ins.
return plug_in_name of type str The name of the loaded plug-in.

unload_plug_in(*name*)
Unloads a DBGF plug-in.
in name of type str The plug-in name or DLL. Special name 'all' unloads all plug-ins.

detect_os()
Tries to (re-)detect the guest OS kernel.

This feature is not implemented in the 4.0.0 release but may show up in a dot release.
return os of type str The detected OS kernel on success.

query_os_kernel_log(*max_messages*)
Tries to get the kernel log (dmesg) of the guest OS.
in max_messages of type int Max number of messages to return, counting from the end of the log. If 0, there is no limit.
return dmesg of type str The kernel log.

get_register(*cpu_id*, *name*)
Gets one register.
in cpu_id of type int The identifier of the Virtual CPU.
in name of type str The register name, case is ignored.
return value of type str The register value. This is usually a hex value (always 0x prefixed) but other format may be used for floating point registers (TBD).

get_registers(*cpu_id*)
Gets all the registers for the given CPU.
in cpu_id of type int The identifier of the Virtual CPU.
out names of type str Array containing the lowercase register names.
out values of type str Array parallel to the names holding the register values as if the register was returned by *IMachineDebugger.get_register()* .

set_register(*cpu_id*, *name*, *value*)
Gets one register.

This feature is not implemented in the 4.0.0 release but may show up in a dot release.
in cpu_id of type int The identifier of the Virtual CPU.
in name of type str The register name, case is ignored.
in value of type str The new register value. Hexadecimal, decimal and octal formattings are supported in addition to any special formattings returned by the getters.

set_registers(*cpu_id*, *names*, *values*)
Sets zero or more registers atomically.

This feature is not implemented in the 4.0.0 release but may show up in a dot release.
in cpu_id of type int The identifier of the Virtual CPU.
in names of type str Array containing the register names, case ignored.
in values of type str Array paralell to the names holding the register values. See *IMachineDebugger.set_register()* for formatting guidelines.

dump_guest_stack(*cpu_id*)
Produce a simple stack dump using the current guest state.

This feature is not implemented in the 4.0.0 release but may show up in a dot release.
in cpu_id of type int The identifier of the Virtual CPU.
return stack of type str String containing the formatted stack dump.

reset_stats(*pattern*)
Reset VM statistics.
in pattern of type str The selection pattern. A bit similar to filename globbing.

dump_stats(*pattern*)
Dumps VM statistics.
in pattern of type str The selection pattern. A bit similar to filename globbing.

get_stats(*pattern*, *with_descriptions*)
Get the VM statistics in a XMLish format.
in pattern of type str The selection pattern. A bit similar to filename globbing.
in with_descriptions of type bool Whether to include the descriptions.
return stats of type str The XML document containing the statistics.

single_step
Get or set bool value for 'singleStep' Switch for enabling single-stepping.

recompile_user
Get or set bool value for 'recompileUser' Switch for forcing code recompilation for user mode code.

recompile_supervisor
Get or set bool value for 'recompileSupervisor' Switch for forcing code recompilation for supervisor mode code.

execute_all_in_iem
Get or set bool value for 'executeAllInIEM' Whether to execute all the code in the instruction

interpreter. This is mainly for testing the interpreter and not an execution mode intended for general consumption.

patm_enabled
Get or set bool value for 'PATMEnabled' Switch for enabling and disabling the PATM component.

csam_enabled
Get or set bool value for 'CSAMEnabled' Switch for enabling and disabling the CSAM component.

log_enabled
Get or set bool value for 'logEnabled' Switch for enabling and disabling the debug logger.

log_dbg_flags
Get str value for 'logDbgFlags' The debug logger flags.

log_dbg_groups
Get str value for 'logDbgGroups' The debug logger's group settings.

log_dbg_destinations
Get str value for 'logDbgDestinations' The debug logger's destination settings.

log_rel_flags
Get str value for 'logRelFlags' The release logger flags.

log_rel_groups
Get str value for 'logRelGroups' The release logger's group settings.

log_rel_destinations
Get str value for 'logRelDestinations' The relase logger's destination settings.

hw_virt_ex_enabled
Get bool value for 'HWVirtExEnabled' Flag indicating whether the VM is currently making use of CPU hardware virtualization extensions.

hw_virt_ex_nested_paging_enabled
Get bool value for 'HWVirtExNestedPagingEnabled' Flag indicating whether the VM is currently making use of the nested paging CPU hardware virtualization extension.

hw_virt_ex_vpid_enabled
Get bool value for 'HWVirtExVPIDEnabled' Flag indicating whether the VM is currently making use of the VPID VT-x extension.

hw_virt_ex_ux_enabled
Get bool value for 'HWVirtExUXEnabled' Flag indicating whether the VM is currently making use of the unrestricted execution feature of VT-x.

os_name
Get str value for 'OSName' Query the guest OS kernel name as detected by the DBGF.

This feature is not implemented in the 4.0.0 release but may show up in a dot release.

os_version
Get str value for 'OSVersion' Query the guest OS kernel version string as detected by the DBGF.

This feature is not implemented in the 4.0.0 release but may show up in a dot release.

pae_enabled
Get bool value for 'PAEEnabled' Flag indicating whether the VM is currently making use of the Physical Address Extension CPU feature.

virtual_time_rate
Get or set int value for 'virtualTimeRate' The rate at which the virtual time runs expressed as a percentage. The accepted range is 2% to 20000%.

vm
Get int value for 'VM' Gets the user-mode VM handle, with a reference. Must be passed to VMR3ReleaseUVM when done. This is only for internal use while we carve the details of this interface.

uptime
Get int value for 'uptime' VM uptime in milliseconds, i.e. time in which it could have been executing guest code. Excludes the time when the VM was paused.

class virtualbox.library.**IUSBDeviceFilters**(*interface=None*)
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.usb_device_filters()`).

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 *USBDeviceState.available*, *USBDeviceState.busy*, *USBDeviceState.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.

IUSBDeviceFilter, *IUSBController*

device_filters
Get IUSBDeviceFilter value for '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.usb_device_filters()`).

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 *USBDeviceState.available*, *USBDeviceState.busy*, *USBDeviceState.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.

IUSBDeviceFilter, *IUSBController*

create_device_filter(*name*)
Creates a new USB device filter. All attributes except the filter name are set to empty (any match), *active* is @c false (the filter is not active).

The created filter can then be added to the list of filters using *insert_device_filter()*.

device_filters()
in name of type str Filter name. See *IUSBDeviceFilter.name()* for more info.
return filter_p of type ***IUSBDeviceFilter*** Created filter object.
raises ***VBoxErrorInvalidVmState*** The virtual machine is not mutable.

insert_device_filter(*position*, *filter_p*)
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.

Duplicates are not allowed, so an attempt to insert a filter that is already in the collection, will return an error.

device_filters()
in position of type int Position to insert the filter to.
in filter_p of type ***IUSBDeviceFilter*** USB device filter to insert.
raises ***VBoxErrorInvalidVmState*** Virtual machine is not mutable.
raises ***OleErrorInvalidarg*** USB device filter not created within this VirtualBox instance.
raises ***VBoxErrorInvalidObjectState*** USB device filter already in list.

remove_device_filter(*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.

device_filters()
in position of type int Position to remove the filter from.
return filter_p of type ***IUSBDeviceFilter*** Removed USB device filter.
raises ***VBoxErrorInvalidVmState*** Virtual machine is not mutable.
raises ***OleErrorInvalidarg*** USB device filter list empty or invalid @a position.

class virtualbox.library.**IUSBController**(*interface=None*)
The USB Controller name.

name
Get or set str value for 'name' The USB Controller name.

type_p
Get or set USBControllerType value for 'type' The USB Controller type.

usb_standard
Get int value for '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.

class virtualbox.library.**IUSBDevice**(*interface=None*)
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.usb_devices()* attribute which lists all USB devices attached to a running virtual machine's USB controller.

id_p
Get str value for 'id' Unique USB device ID. This ID is built from #vendorId, #productId, #revision and #serialNumber.

vendor_id
Get int value for 'vendorId' Vendor ID.

product_id
Get int value for 'productId' Product ID.

revision
Get int value for '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.

manufacturer
Get str value for 'manufacturer' Manufacturer string.

product
Get str value for 'product' Product string.

serial_number
Get str value for 'serialNumber' Serial number string.

address
Get str value for 'address' Host specific address of the device.

port
Get int value for 'port' Host USB port number the device is physically connected to.

version
Get int value for 'version' The major USB version of the device - 1, 2 or 3.

port_version
Get int value for 'portVersion' The major USB version of the host USB port the device is physically connected to - 1, 2 or 3. For devices not connected to anything this will have the same value as the version attribute.

speed
Get USBConnectionSpeed value for 'speed' The speed at which the device is currently communicating.

remote
Get bool value for 'remote' Whether the device is physically connected to a remote VRDE client or to a local host machine.

device_info
Get str value for 'deviceInfo' Array of device attributes as single strings.

So far the following are used: 0: The manufacturer string, if the device doesn't expose the ID one is taken from an internal database or an empty string if none is found. 1: The product string, if the device doesn't expose the ID one is taken from an internal database or an empty string if none is found.

backend
Get str value for 'backend' The backend which will be used to communicate with this device.

class virtualbox.library.**IUSBDeviceFilter**(*interface=None*)
The IUSBDeviceFilter interface represents an USB device filter used to perform actions on a group of USB devices.

This type of filters is used by running virtual machines to automatically capture selected USB devices once they are physically attached to the host computer.

A USB device is matched to the given device filter if and only if all attributes of the device match the corresponding attributes of the filter (that is, attributes are joined together using the logical AND operation). On the other hand, all together, filters in the list of filters carry the semantics of the logical OR operation. So if it is desirable to create a match like "this vendor id OR this product id", one needs to create two filters and specify "any match" (see below) for unused attributes.

All filter attributes used for matching are strings. Each string is an expression representing a set of values of the corresponding device attribute, that will match the given filter. Currently, the following filtering expressions are supported:

Interval filters. Used to specify valid intervals for integer device attributes (Vendor ID, Product ID and Revision). The format of the string is:

int:((m)|([m]-[n]))(,(m)|([m]-[n]))*

where m and n are integer numbers, either in octal (starting from 0), hexadecimal (starting from 0x) or decimal (otherwise) form, so that m < n. If m is omitted before a dash (-), the minimum possible integer is assumed; if n is omitted after a dash, the maximum possible integer is assumed.

Boolean filters. Used to specify acceptable values for boolean device attributes. The format of the string is:

true|false|yes|no|0|1

Exact match. Used to specify a single value for the given device attribute. Any string that doesn't start with int: represents the exact match. String device attributes are compared to this string including case of symbols. Integer attributes are first converted to a string (see individual filter attributes) and then compared ignoring case.

Any match. Any value of the corresponding device attribute will match the given filter. An empty or @c null string is used to construct this type of filtering expressions.

On the Windows host platform, interval filters are not currently available. Also all string filter attributes (*manufacturer()* , *product()* , *serial_number()*) are ignored, so they behave as *any match* no matter what string expression is specified.

IUSBDeviceFilters.device_filters() , *IHostUSBDeviceFilter*

name
Get or set str value for 'name' Visible name for this filter. This name is used to visually distinguish one filter from another, so it can neither be @c null nor an empty string.

active
Get or set bool value for 'active' Whether this filter active or has been temporarily disabled.

vendor_id
Get or set str value for 'vendorId' *IUSBDevice.vendor_id()* Vendor ID filter. The string representation for the *exact matching* has the form XXXX, where X is the hex digit (including leading zeroes).

product_id
Get or set str value for 'productId' *IUSBDevice.product_id()* Product ID filter. The string representation for the *exact matching* has the form XXXX, where X is the hex digit (including leading zeroes).

revision
Get or set str value for 'revision' *IUSBDevice.product_id()* Product revision number filter. The string representation for the *exact matching* has the form IIFF, where I is the decimal digit of the integer part of the revision, and F is the decimal digit of its fractional part (including leading and trailing zeros). Note that for interval filters, it's best to use the hexadecimal form, because the revision is stored as a 16 bit packed BCD value; so the expression int:0x0100-0x0199 will match any revision from 1.0 to 1.99.

manufacturer
Get or set str value for 'manufacturer' *IUSBDevice.manufacturer()* Manufacturer filter.

product
Get or set str value for 'product' *IUSBDevice.product()* Product filter.

serial_number
Get or set str value for 'serialNumber' *IUSBDevice.serial_number()* Serial number filter.

port
Get or set str value for 'port' *IUSBDevice.port()* Host USB port filter.

remote
Get or set str value for 'remote' *IUSBDevice.remote()* Remote state filter.

This filter makes sense only for machine USB filters, i.e. it is ignored by IHostUSBDeviceFilter objects.

masked_interfaces
Get or set int value for 'maskedInterfaces' This is an advanced option for hiding one or more USB interfaces from the guest. The value is a bit mask where the bits that are set means the corresponding USB interface should be hidden, masked off if you like. This feature only works on Linux hosts.

class virtualbox.library.**IHostUSBDevice**(*interface=None*)
The IHostUSBDevice interface represents a physical USB device attached to the host computer.

Besides properties inherited from IUSBDevice, this interface adds the *state()* property that holds the current state of the USB device.

`IHost.usb_devices()` , `IHost.usb_device_filters()`

state
Get USBDeviceState value for 'state' Current state of the device.

class virtualbox.library.**IHostUSBDeviceFilter**(*interface=None*)
The IHostUSBDeviceFilter interface represents a global filter for a physical USB device used by the host computer. Used indirectly in `IHost.usb_device_filters()` .

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.

The *IUSBDeviceFilter.remote()* attribute is ignored by this type of filters, because it makes sense only for *IUSBDeviceFilters.device_filters()* machine USB filters.

`IHost.usb_device_filters()`

action
Get or set USBDeviceFilterAction value for 'action' Action performed by the host when an attached USB device matches this filter.

class virtualbox.library.**IUSBProxyBackend**(*interface=None*)
The USBProxyBackend interface represents a source for USB devices available to the host for attaching to the VM.

name
Get str value for 'name' The unique name of the proxy backend.

type_p
Get str value for 'type' The type of the backend.

class virtualbox.library.**IAudioAdapter**(*interface=None*)
The IAudioAdapter interface represents the virtual audio adapter of the virtual machine. Used in *IMachine.audio_adapter()* .

enabled
Get or set bool value for '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.

enabled_in
Get or set bool value for 'enabledIn' Flag whether the audio adapter is enabled for audio input. Only relevant if the adapter is enabled.

enabled_out
Get or set bool value for 'enabledOut' Flag whether the audio adapter is enabled for audio output. Only relevant if the adapter is enabled.

audio_controller
Get or set AudioControllerType value for 'audioController' The emulated audio controller.

audio_codec
Get or set AudioCodecType value for 'audioCodec' The exact variant of audio codec hardware presented to the guest. For HDA and SB16, only one variant is available, but for AC'97, there are several.

audio_driver
Get or set AudioDriverType value for 'audioDriver' Audio driver the adapter is connected to. This setting can only be changed when the VM is not running.

properties_list
Get str value for 'propertiesList' Array of names of tunable properties, which can be supported by audio driver.

set_property(*key*, *value*)
Sets an audio specific property string.

If you pass @c null or empty string as a key @a value, the given @a key will be deleted.
in key of type str Name of the key to set.
in value of type str Value to assign to the key.

get_property(*key*)
Returns an audio specific property string.

If the requested data @a key does not exist, this function will succeed and return an empty string in the @a value argument.
in key of type str Name of the key to get.
return value of type str Value of the requested key.

class virtualbox.library.**IVRDEServer**(*interface=None*)
Flag if VRDE server is enabled.

enabled
Get or set bool value for 'enabled' Flag if VRDE server is enabled.

auth_type
Get or set AuthType value for 'authType' VRDE authentication method.

auth_timeout
Get or set int value for 'authTimeout' Timeout for guest authentication. Milliseconds.

allow_multi_connection
Get or set bool value for '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.

reuse_single_connection
Get or set bool value for 'reuseSingleConnection' Flag whether the existing connection must be dropped and a new connection must be established by the VRDE server, when a new client connects in single connection mode.

vrde_ext_pack
Get or set str value for 'VRDEExtPack' The name of Extension Pack providing VRDE for this VM. Overrides *ISystemProperties.default_vrde_ext_pack()* .

auth_library
Get or set str value for 'authLibrary' Library used for authentication of RDP clients by this VM. Overrides *ISystemProperties.vrde_auth_library()* .

vrde_properties
Get str value for 'VRDEProperties' Array of names of properties, which are supported by this VRDE server.

set_vrde_property(*key*, *value*)
Sets a VRDE specific property string.

If you pass @c null or empty string as a key @a value, the given @a key will be deleted.
in key of type str Name of the key to set.
in value of type str Value to assign to the key.

get_vrde_property(*key*)
Returns a VRDE specific property string.

If the requested data @a key does not exist, this function will succeed and return an empty string in the @a value argument.
in key of type str Name of the key to get.
return value of type str Value of the requested key.

class virtualbox.library.**ISharedFolder**(*interface=None*)
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.shared_folders()*), shared folders available to all virtual machines. *Permanent* (*IMachine.shared_folders()*), VM-specific shared folders available to the given virtual machine at startup. *Transient* (*IConsole.shared_folders()*), VM-specific shared folders created in the session context (for example, when the virtual machine is running) and automatically discarded when the session is closed (the VM is powered off).

Logical names of shared folders must be unique within the given scope (global, permanent or transient). However, they do not need to be unique across scopes. In this case, the definition of the shared folder in a more specific scope takes precedence over definitions in all other scopes. The order of precedence is (more specific to more general):

Transient definitions Permanent definitions Global definitions

For example, if MyMachine has a shared folder named C_DRIVE (that points to C:), then creating a transient shared folder named C_DRIVE (that points to C:\WINDOWS) will change the definition of C_DRIVE in the guest OS so that \VBOXSVRC_DRIVE will give access to C:WINDOWS instead of C:on the host PC. Removing the transient shared folder C_DRIVE will restore the previous (permanent) definition of C_DRIVE that points to C:if it still exists.

Note that permanent and transient shared folders of different machines are in different name spaces, so they don't overlap and don't need to have unique logical names.

Global shared folders are not implemented in the current version of the product.

name
Get str value for 'name' Logical name of the shared folder.

host_path
Get str value for 'hostPath' Full path to the shared folder in the host file system.

accessible
Get bool value for 'accessible' Whether the folder defined by the host path is currently accessi-

ble or not. For example, the folder can be inaccessible if it is placed on the network share that is not available by the time this property is read.

writable
Get bool value for 'writable' Whether the folder defined by the host path is writable or not.

auto_mount
Get bool value for 'autoMount' Whether the folder gets automatically mounted by the guest or not.

last_access_error
Get str value for 'lastAccessError' Text message that represents the result of the last accessibility check.

Accessibility checks are performed each time the *accessible()* attribute is read. An empty string is returned if the last accessibility check was successful. A non-empty string indicates a failure and should normally describe a reason of the failure (for example, a file read error).

class virtualbox.library.**IInternalSessionControl**(*interface=None*)
PID of the process that has created this Session object.

pid
Get int value for 'PID' PID of the process that has created this Session object.

remote_console
Get IConsole value for 'remoteConsole' Returns the console object suitable for remote control.

nominal_state
Get MachineState value for 'nominalState' Returns suitable machine state for the VM execution state. Useful for choosing a sensible machine state after a complex operation which failed or otherwise resulted in an unclear situation.

assign_machine(*machine*, *lock_type*, *token*)
Assigns the machine object associated with this direct-type session or informs the session that it will be a remote one (if @a machine == @c null).

in machine of type *IMachine*

in lock_type of type *LockType*

in token of type *IToken*
raises ***VBoxErrorInvalidVmState*** Session state prevents operation.
raises ***VBoxErrorInvalidObjectState*** Session type prevents operation.

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

in machine of type *IMachine*

in console of type *IConsole*
raises ***VBoxErrorInvalidVmState*** Session state prevents operation.

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

in machine_state of type *MachineState*
raises ***VBoxErrorInvalidVmState*** Session state prevents operation.
raises ***VBoxErrorInvalidObjectState*** Session type prevents operation.

uninitialize()
Uninitializes (closes) this session. Used by VirtualBox to close the corresponding remote session when the direct session dies or gets closed.

raises *VBoxErrorInvalidVmState* Session state prevents operation.

on_network_adapter_change(*network_adapter*, *change_adapter*)
Triggered when settings of a network adapter of the associated virtual machine have changed.

in network_adapter of type *INetworkAdapter*

in change_adapter of type bool
raises *VBoxErrorInvalidVmState* Session state prevents operation.
raises *VBoxErrorInvalidObjectState* Session type prevents operation.

on_serial_port_change(*serial_port*)
Triggered when settings of a serial port of the associated virtual machine have changed.

in serial_port of type *ISerialPort*
raises *VBoxErrorInvalidVmState* Session state prevents operation.
raises *VBoxErrorInvalidObjectState* Session type prevents operation.

on_parallel_port_change(*parallel_port*)
Triggered when settings of a parallel port of the associated virtual machine have changed.

in parallel_port of type *IParallelPort*
raises *VBoxErrorInvalidVmState* Session state prevents operation.
raises *VBoxErrorInvalidObjectState* Session type prevents operation.

on_storage_controller_change()
Triggered when settings of a storage controller of the associated virtual machine have changed.
raises *VBoxErrorInvalidVmState* Session state prevents operation.
raises *VBoxErrorInvalidObjectState* Session type prevents operation.

on_medium_change(*medium_attachment*, *force*)
Triggered when attached media of the associated virtual machine have changed.
in medium_attachment of type *IMediumAttachment* The medium attachment which changed.
in force of type bool If the medium change was forced.
raises *VBoxErrorInvalidVmState* Session state prevents operation.
raises *VBoxErrorInvalidObjectState* Session type prevents operation.

on_storage_device_change(*medium_attachment*, *remove*, *silent*)
Triggered when attached storage devices of the associated virtual machine have changed.
in medium_attachment of type *IMediumAttachment* The medium attachment which changed.
in remove of type bool TRUE if the device is removed, FALSE if it was added.
in silent of type bool TRUE if the device is is silently reconfigured without notifying the guest about it.
raises *VBoxErrorInvalidVmState* Session state prevents operation.
raises *VBoxErrorInvalidObjectState* Session type prevents operation.

on_clipboard_mode_change(*clipboard_mode*)
Notification when the shared clipboard mode changes.
in clipboard_mode of type *ClipboardMode* The new shared clipboard mode.

on_dn_d_mode_change(*dnd_mode*)
Notification when the drag'n drop mode changes.
in dnd_mode of type *DnDMode* The new mode for drag'n drop.

on_cpu_change(*cpu*, *add*)
Notification when a CPU changes.
in cpu of type int The CPU which changed
in add of type bool Flag whether the CPU was added or removed

on_cpu_execution_cap_change(*execution_cap*)
Notification when the CPU execution cap changes.
in execution_cap of type int The new CPU execution cap value. (1-100)

on_vrde_server_change(*restart*)
Triggered when settings of the VRDE server object of the associated virtual machine have changed.
in restart of type bool Flag whether the server must be restarted
raises ***VBoxErrorInvalidVmState*** Session state prevents operation.
raises ***VBoxErrorInvalidObjectState*** Session type prevents operation.

on_video_capture_change()
Triggered when video capture settings have changed.

on_usb_controller_change()
Triggered when settings of the USB controller object of the associated virtual machine have changed.
raises ***VBoxErrorInvalidVmState*** Session state prevents operation.
raises ***VBoxErrorInvalidObjectState*** Session type prevents operation.

on_shared_folder_change(*global_p*)
Triggered when a permanent (global or machine) shared folder has been created or removed.

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.

in global_p of type bool
raises ***VBoxErrorInvalidVmState*** Session state prevents operation.
raises ***VBoxErrorInvalidObjectState*** Session type prevents operation.

on_usb_device_attach(*device*, *error*, *masked_interfaces*, *capture_filename*)
Triggered when a request to capture a USB device (as a result of matched USB filters or direct call to *IConsole.attach_usb_device()*) has completed. A @c null @a error object means success, otherwise it describes a failure.

in device of type *IUSBDevice*

in error of type *IVirtualBoxErrorInfo*

in masked_interfaces of type int

in capture_filename of type str
raises ***VBoxErrorInvalidVmState*** Session state prevents operation.
raises ***VBoxErrorInvalidObjectState*** Session type prevents operation.

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

in id_p of type str

in error of type *IVirtualBoxErrorInfo*
raises ***VBoxErrorInvalidVmState*** Session state prevents operation.
raises ***VBoxErrorInvalidObjectState*** Session type prevents operation.

on_show_window(*check*)
Called by *IMachine.can_show_console_window()* and by *IMachine.show_console_window()* in order to notify console listeners *ICanShowWindowEvent* and *IShowWindowEvent* .

in check of type bool

out can_show of type bool

out win_id of type int
raises *VBoxErrorInvalidObjectState* Session type prevents operation.

on_bandwidth_group_change(*bandwidth_group*)
Notification when one of the bandwidth groups change.
in bandwidth_group of type *IBandwidthGroup* The bandwidth group which changed.

access_guest_property(*name*, *value*, *flags*, *access_mode*)
Called by *IMachine.get_guest_property()* and by *IMachine.set_guest_property()* in order to read and modify guest properties.

in name of type str

in value of type str

in flags of type str
in access_mode of type int 0 = get, 1 = set, 2 = delete.
out ret_value of type str

out ret_timestamp of type int

out ret_flags of type str
raises *VBoxErrorInvalidVmState* Machine session is not open.
raises *VBoxErrorInvalidObjectState* Session type is not direct.

enumerate_guest_properties(*patterns*)
Return a list of the guest properties matching a set of patterns along with their values, time stamps and flags.
in patterns of type str The patterns to match the properties against as a comma-separated string. If this is empty, all properties currently set will be returned.
out keys of type str The key names of the properties returned.
out values of type str The values of the properties returned. The array entries match the corresponding entries in the @a key array.
out timestamps of type int The time stamps of the properties returned. The array entries match the corresponding entries in the @a key array.
out flags of type str The flags of the properties returned. The array entries match the corresponding entries in the @a key array.
raises *VBoxErrorInvalidVmState* Machine session is not open.
raises *VBoxErrorInvalidObjectState* Session type is not direct.

online_merge_medium(*medium_attachment*, *source_idx*, *target_idx*, *progress*)
Triggers online merging of a hard disk. Used internally when deleting a snapshot while a VM referring to the same hard disk chain is running.
in medium_attachment of type *IMediumAttachment* The medium attachment to identify the medium chain.
in source_idx of type int The index of the source image in the chain. Redundant, but drastically reduces IPC.
in target_idx of type int The index of the target image in the chain. Redundant, but drastically reduces IPC.
in progress of type *IProgress* Progress object for this operation.
raises *VBoxErrorInvalidVmState* Machine session is not open.
raises *VBoxErrorInvalidObjectState* Session type is not direct.

reconfigure_medium_attachments(*attachments*)
Reconfigure all specified medium attachments in one go, making sure the current state corresponds to the specified medium.

in attachments of type ***IMediumAttachment*** Array containing the medium attachments which need to be reconfigured.
raises ***VBoxErrorInvalidVmState*** Machine session is not open.
raises ***VBoxErrorInvalidObjectState*** Session type is not direct.

enable_vmm_statistics(*enable*)
Enables or disables collection of VMM RAM statistics.
in enable of type bool True enables statistics collection.
raises ***VBoxErrorInvalidVmState*** Machine session is not open.
raises ***VBoxErrorInvalidObjectState*** Session type is not direct.

pause_with_reason(*reason*)
Internal method for triggering a VM pause with a specified reason code. The reason code can be interpreted by device/drivers and thus it might behave slightly differently than a normal VM pause.

IConsole.pause()
in reason of type ***Reason*** Specify the best matching reason code please.
raises ***VBoxErrorInvalidVmState*** Virtual machine not in Running state.
raises ***VBoxErrorVmError*** Virtual machine error in suspend operation.

resume_with_reason(*reason*)
Internal method for triggering a VM resume with a specified reason code. The reason code can be interpreted by device/drivers and thus it might behave slightly differently than a normal VM resume.

IConsole.resume()
in reason of type ***Reason*** Specify the best matching reason code please.
raises ***VBoxErrorInvalidVmState*** Virtual machine not in Paused state.
raises ***VBoxErrorVmError*** Virtual machine error in resume operation.

save_state_with_reason(*reason*, *progress*, *state_file_path*, *pause_vm*)
Internal method for triggering a VM save state with a specified reason code. The reason code can be interpreted by device/drivers and thus it might behave slightly differently than a normal VM save state.

This call is fully synchronous, and the caller is expected to have set the machine state appropriately (and has to set the follow-up machine state if this call failed).

IMachine.save_state()
in reason of type ***Reason*** Specify the best matching reason code please.
in progress of type ***IProgress*** Progress object to track the operation completion.
in state_file_path of type str File path the VM process must save the execution state to.
in pause_vm of type bool The VM should be paused before saving state. It is automatically unpaused on error in the "vanilla save state" case.
return left_paused of type bool Returns if the VM was left in paused state, which is necessary in many situations (snapshots, teleportation).
raises ***VBoxErrorInvalidVmState*** Virtual machine state is not one of the expected values.
raises ***VBoxErrorFileError*** Failed to create directory for saved state file.

cancel_save_state_with_reason()
Internal method for cancelling a VM save state. *IInternalSessionControl.save_state_with_reason()*

class virtualbox.library.**IStorageController**(*interface=None*)
Represents a storage controller that is attached to a virtual machine (*IMachine*). Just as drives (hard disks, DVDs, FDs) are attached to storage controllers in a real computer, virtual drives (represented by *IMediumAttachment*) are attached to virtual storage controllers, represented by this

interface.

As opposed to physical hardware, VirtualBox has a very generic concept of a storage controller, and for purposes of the Main API, all virtual storage is attached to virtual machines via instances of this interface. There are five types of such virtual storage controllers: IDE, SCSI, SATA, SAS and Floppy (see *bus()*). Depending on which of these four is used, certain sub-types may be available and can be selected in *controller_type()* .

Depending on these settings, the guest operating system might see significantly different virtual hardware.

name
Get or set str value for 'name' Name of the storage controller, as originally specified with *IMachine.add_storage_controller()* . This then uniquely identifies this controller with other method calls such as *IMachine.attach_device()* and *IMachine.mount_medium()* .

max_devices_per_port_count
Get int value for 'maxDevicesPerPortCount' Maximum number of devices which can be attached to one port.

min_port_count
Get int value for 'minPortCount' Minimum number of ports that *IStorageController.port_count()* can be set to.

max_port_count
Get int value for 'maxPortCount' Maximum number of ports that *IStorageController.port_count()* can be set to.

instance
Get or set int value for 'instance' The instance number of the device in the running VM.

port_count
Get or set int value for 'portCount' The number of currently usable ports on the controller. The minimum and maximum number of ports for one controller are stored in *IStorageController.min_port_count()* and *IStorageController.max_port_count()* .

bus
Get StorageBus value for 'bus' The bus type of the storage controller (IDE, SATA, SCSI, SAS or Floppy).

controller_type
Get or set StorageControllerType value for 'controllerType' The exact variant of storage controller hardware presented to the guest. Depending on this value, VirtualBox will provide a different virtual storage controller hardware to the guest. For SATA, SAS and floppy controllers, only one variant is available, but for IDE and SCSI, there are several.

For SCSI controllers, the default type is LsiLogic.

use_host_io_cache
Get or set bool value for 'useHostIOCache' If true, the storage controller emulation will use a dedicated I/O thread, enable the host I/O caches and use synchronous file APIs on the host. This was the only option in the API before VirtualBox 3.2 and is still the default for IDE controllers.

If false, the host I/O cache will be disabled for image files attached to this storage controller. Instead, the storage controller emulation will use asynchronous I/O APIs on the host. This makes it possible to turn off the host I/O caches because the emulation can handle unaligned access to the file. This should be used on OS X and Linux hosts if a high I/O load is expected or many virtual machines are running at the same time to prevent I/O cache related hangs.

This option new with the API of VirtualBox 3.2 and is now the default for non-IDE storage controllers.

bootable
Get bool value for 'bootable' Returns whether it is possible to boot from disks attached to this controller.

class virtualbox.library.**IPerformanceMetric**(*interface=None*)
The IPerformanceMetric interface represents parameters of the given performance metric.

metric_name
Get str value for 'metricName' Name of the metric.

object_p
Get Interface value for 'object' Object this metric belongs to.

description
Get str value for 'description' Textual description of the metric.

period
Get int value for 'period' Time interval between samples, measured in seconds.

count
Get int value for '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.

unit
Get str value for 'unit' Unit of measurement.

minimum_value
Get int value for 'minimumValue' Minimum possible value of this metric.

maximum_value
Get int value for 'maximumValue' Maximum possible value of this metric.

class virtualbox.library.**IPerformanceCollector**(*interface=None*)
The IPerformanceCollector interface represents a service that collects and stores performance metrics data.

Performance metrics are associated with objects of interfaces like IHost and IMachine. Each object has a distinct set of performance metrics. The set can be obtained with *IPerformanceCollector.get_metrics()*.

Metric data is collected at the specified intervals and is retained internally. The interval and the number of retained samples can be set with *IPerformanceCollector.setup_metrics()*. Both metric data and collection settings are not persistent, they are discarded as soon as VBoxSVC process terminates. Moreover, metric settings and data associated with a particular VM only exist while VM is running. They disappear as soon as VM shuts down. It is not possible to set up metrics for machines that are powered off. One needs to start VM first, then set up metric collection parameters.

Metrics are organized hierarchically, with each level separated by a slash (/) character. Generally, the scheme for metric names is like this:

Category/Metric[/SubMetric][:aggregation]

"Category/Metric" together form the base metric name. A base metric is the smallest unit for which a sampling interval and the number of retained samples can be set. Only base metrics can be enabled and disabled. All sub-metrics are collected when their base metric is collected.

Collected values for any set of sub-metrics can be queried with *IPerformanceCollector.query_metrics_data()* .

For example "CPU/Load/User:avg" metric name stands for the "CPU" category, "Load" metric, "User" submetric, "average" aggregate. An aggregate function is computed over all retained data. Valid aggregate functions are:

avg – average min – minimum max – maximum

When setting up metric parameters, querying metric data, enabling or disabling metrics wildcards can be used in metric names to specify a subset of metrics. For example, to select all CPU-related metrics use CPU/*, all averages can be queried using *:avg and so on. To query metric values without aggregates *: can be used.

The valid names for base metrics are:

CPU/Load CPU/MHz RAM/Usage RAM/VMM

The general sequence for collecting and retrieving the metrics is:

Obtain an instance of IPerformanceCollector with *IVirtualBox.performance_collector()*

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.setup_metrics()* . 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.query_metrics_data()* . The data that have been collected so far are returned. Note that the values are still retained internally and data collection continues.

For an example of usage refer to the following files in VirtualBox SDK:

Java: bindings/webservice/java/jax-ws/samples/metrictest.java

Python: bindings/xpcom/python/sample/shellcommon.py

metric_names
Get str value for '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.get_metrics()* can be used to get the list of supported metrics for a particular object.

get_metrics(*metric_names*, *objects*)
Returns parameters of specified metrics for a set of objects.

@c Null metrics array means all metrics. @c Null object array means all existing objects.
in metric_names of type str Metric name filter. Currently, only a comma-separated list of metrics is supported.
in objects of type Interface Set of objects to return metric parameters for.
return metrics of type *IPerformanceMetric* Array of returned metric parameters.

setup_metrics(*metric_names*, *objects*, *period*, *count*)
Sets parameters of specified base metrics for a set of objects. Returns an array of *IPerformanceMetric* describing the metrics have been affected.

@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.

in metric_names of type str Metric name filter. Comma-separated list of metrics with wildcard support.

in objects of type Interface Set of objects to setup metric parameters for.

in period of type int Time interval in seconds between two consecutive samples of performance data.

in count of type int Number of samples to retain in performance data history. Older samples get discarded.

return affected_metrics of type *IPerformanceMetric* Array of metrics that have been modified by the call to this method.

enable_metrics(*metric_names*, *objects*)
Turns on collecting specified base metrics. Returns an array of *IPerformanceMetric* describing the metrics have been affected.

@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.

in metric_names of type str Metric name filter. Comma-separated list of metrics with wildcard support.

in objects of type Interface Set of objects to enable metrics for.

return affected_metrics of type *IPerformanceMetric* Array of metrics that have been modified by the call to this method.

disable_metrics(*metric_names*, *objects*)
Turns off collecting specified base metrics. Returns an array of *IPerformanceMetric* describing the metrics have been affected.

@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.

in metric_names of type str Metric name filter. Comma-separated list of metrics with wildcard support.

in objects of type Interface Set of objects to disable metrics for.

return affected_metrics of type *IPerformanceMetric* Array of metrics that have been modified by the call to this method.

query_metrics_data(*metric_names*, *objects*)
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]].

@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.

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 allows 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.

in metric_names of type str Metric name filter. Comma-separated list of metrics with wildcard support.

in objects of type Interface Set of objects to query metrics for.

out return_metric_names of type str Names of metrics returned in @c returnData.

out return_objects of type Interface Objects associated with metrics returned in @c returnData.

out return_units of type str Units of measurement for each returned metric.

out return_scales of type int Divisor that should be applied to return values in order to get floating point values. For example: (double)returnData[returnDataIndices[0]+i] / returnScales[0] will retrieve the floating point value of i-th sample of the first metric.

out return_sequence_numbers of type int Sequence numbers of the first elements of value sequences of particular metrics returned in @c returnData. For aggregate metrics it is the sequence number of the sample the aggregate started calculation from.

out return_data_indices of type int Indices of the first elements of value sequences of particular metrics returned in @c returnData.

out return_data_lengths of type int Lengths of value sequences of particular metrics.

return return_data of type int Flattened array of all metric data containing sequences of values for each metric.

class virtualbox.library.**INATEngine**(*interface=None*)

Interface for managing a NAT engine which is used with a virtual machine. This allows for changing NAT behavior such as port-forwarding rules. This interface is used in the *INetworkAdapter.nat_engine()* attribute.

network

Get or set str value for 'network' The network attribute of the NAT engine (the same value is used with built-in DHCP server to fill corresponding fields of DHCP leases).

host_ip

Get or set str value for 'hostIP' IP of host interface to bind all opened sockets to. Changing this does not change binding of port forwarding.

tftp_prefix

Get or set str value for 'TFTPPrefix' TFTP prefix attribute which is used with the built-in DHCP server to fill the corresponding fields of DHCP leases.

tftp_boot_file

Get or set str value for 'TFTPBootFile' TFTP boot file attribute which is used with the built-in DHCP server to fill the corresponding fields of DHCP leases.

tftp_next_server

Get or set str value for 'TFTPNextServer' TFTP server attribute which is used with the built-in DHCP server to fill the corresponding fields of DHCP leases. The preferred form is IPv4 addresses.

alias_mode

Get or set int value for 'aliasMode'

dns_pass_domain
Get or set bool value for 'DNSPassDomain' Whether the DHCP server should pass the DNS domain used by the host.

dns_proxy
Get or set bool value for 'DNSProxy' Whether the DHCP server (and the DNS traffic by NAT) should pass the address of the DNS proxy and process traffic using DNS servers registered on the host.

dns_use_host_resolver
Get or set bool value for 'DNSUseHostResolver' Whether the DHCP server (and the DNS traffic by NAT) should pass the address of the DNS proxy and process traffic using the host resolver mechanism.

redirects
Get str value for 'redirects' Array of NAT port-forwarding rules in string representation, in the following format: "name,protocol id,host ip,host port,guest ip,guest port".

set_network_settings(*mtu*, *sock_snd*, *sock_rcv*, *tcp_wnd_snd*, *tcp_wnd_rcv*)
Sets network configuration of the NAT engine.
in mtu of type int MTU (maximum transmission unit) of the NAT engine in bytes.
in sock_snd of type int Capacity of the socket send buffer in bytes when creating a new socket.
in sock_rcv of type int Capacity of the socket receive buffer in bytes when creating a new socket.
in tcp_wnd_snd of type int Initial size of the NAT engine's sending TCP window in bytes when establishing a new TCP connection.
in tcp_wnd_rcv of type int Initial size of the NAT engine's receiving TCP window in bytes when establishing a new TCP connection.

get_network_settings()
Returns network configuration of NAT engine. See *set_network_settings()* for parameter descriptions.

out mtu of type int

out sock_snd of type int

out sock_rcv of type int

out tcp_wnd_snd of type int

out tcp_wnd_rcv of type int

add_redirect(*name*, *proto*, *host_ip*, *host_port*, *guest_ip*, *guest_port*)
Adds a new NAT port-forwarding rule.
in name of type str The name of the rule. An empty name is acceptable, in which case the NAT engine auto-generates one using the other parameters.
in proto of type *NATProtocol* Protocol handled with the rule.
in host_ip of type str IP of the host interface to which the rule should apply. An empty ip address is acceptable, in which case the NAT engine binds the handling socket to any interface.
in host_port of type int The port number to listen on.
in guest_ip of type str The IP address of the guest which the NAT engine will forward matching packets to. An empty IP address is acceptable, in which case the NAT engine will forward packets to the first DHCP lease (x.x.x.15).
in guest_port of type int The port number to forward.

remove_redirect(*name*)
Removes a port-forwarding rule that was previously registered.
in name of type str The name of the rule to delete.

class virtualbox.library.**IExtPackPlugIn**(*interface=None*)
Interface for keeping information about a plug-in that ships with an extension pack.

name
Get str value for 'name' The plug-in name.

description
Get str value for 'description' The plug-in description.

frontend
Get str value for 'frontend' The name of the frontend or component name this plug-in plugs into.

module_path
Get str value for 'modulePath' The module path.

class virtualbox.library.**IExtPackBase**(*interface=None*)
Interface for querying information about an extension pack as well as accessing COM objects within it.

name
Get str value for 'name' The extension pack name. This is unique.

description
Get str value for 'description' The extension pack description.

version
Get str value for 'version' The extension pack version string. This is restricted to the dotted version number and optionally a build indicator. No tree revision or tag will be included in the string as those things are available as separate properties. An optional publisher tag may be present like for *IVirtualBox.version()* .

Examples: "1.2.3", "1.2.3_BETA1" and "1.2.3_RC2".

revision
Get int value for 'revision' The extension pack internal revision number.

edition
Get str value for 'edition' Edition indicator. This is usually empty.

Can for instance be used to help distinguishing between two editions of the same extension pack where only the license, service contract or something differs.

vrde_module
Get str value for 'VRDEModule' The name of the VRDE module if the extension pack sports one.

plug_ins
Get IExtPackPlugIn value for 'plugIns' Plug-ins provided by this extension pack.

usable
Get bool value for 'usable' Indicates whether the extension pack is usable or not.

There are a number of reasons why an extension pack might be unusable, typical examples would be broken installation/file or that it is incompatible with the current VirtualBox version.

why_unusable
Get str value for 'whyUnusable' String indicating why the extension pack is not usable. This is an empty string if usable and always a non-empty string if not usable.

show_license
Get bool value for 'showLicense' Whether to show the license before installation

license_p
Get str value for 'license' The default HTML license text for the extension pack. Same as calling *query_license()* queryLicense with preferredLocale and preferredLanguage as empty strings and format set to html.

query_license(*preferred_locale*, *preferred_language*, *format_p*)
Full feature version of the license attribute.
in preferred_locale of type str The preferred license locale. Pass an empty string to get the default license.
in preferred_language of type str The preferred license language. Pass an empty string to get the default language for the locale.
in format_p of type str The license format: html, rtf or txt. If a license is present there will always be an HTML of it, the rich text format (RTF) and plain text (txt) versions are optional. If
return license_text of type str The license text.

class virtualbox.library.**IExtPack**(*interface=None*)
Interface for querying information about an extension pack as well as accessing COM objects within it.

query_object(*obj_uuid*)
Queries the IUnknown interface to an object in the extension pack main module. This allows plug-ins and others to talk directly to an extension pack.
in obj_uuid of type str The object ID. What exactly this is
return return_interface of type Interface The queried interface.

class virtualbox.library.**IExtPackFile**(*interface=None*)
Extension pack file (aka tarball, .vbox-extpack) representation returned by *IExtPackManager.open_ext_pack_file()* . This provides the base extension pack information with the addition of the file name.

file_path
Get str value for 'filePath' The path to the extension pack file.

install(*replace*, *display_info*)
Install the extension pack.
in replace of type bool Set this to automatically uninstall any existing extension pack with the same name as the one being installed.
in display_info of type str Platform specific display information. Reserved for future hacks.
return progess of type *IProgress* Progress object for the operation.

class virtualbox.library.**IExtPackManager**(*interface=None*)
Interface for managing VirtualBox Extension Packs.

@todo Describe extension packs, how they are managed and how to create one.

installed_ext_packs
Get IExtPack value for 'installedExtPacks' List of the installed extension packs.

find(*name*)
Returns the extension pack with the specified name if found.
in name of type str The name of the extension pack to locate.
return return_data of type *IExtPack* The extension pack if found.
raises *VBoxErrorObjectNotFound* No extension pack matching @a name was found.

open_ext_pack_file(*path*)
Attempts to open an extension pack file in preparation for installation.
in path of type str The path of the extension pack tarball. This can optionally be followed by a "::SHA-256=hex-digit" of the tarball.

return file_p of type *IExtPackFile* The interface of the extension pack file object.

uninstall(*name*, *forced_removal*, *display_info*)
Uninstalls an extension pack, removing all related files.
in name of type str The name of the extension pack to uninstall.
in forced_removal of type bool Forced removal of the extension pack. This means that the uninstall hook will not be called.
in display_info of type str Platform specific display information. Reserved for future hacks.
return progess of type *IProgress* Progress object for the operation.

cleanup()
Cleans up failed installs and uninstalls

query_all_plug_ins_for_frontend(*frontend_name*)
Gets the path to all the plug-in modules for a given frontend.

This is a convenience method that is intended to simplify the plug-in loading process for a frontend.
in frontend_name of type str The name of the frontend or component.
return plug_in_modules of type str Array containing the plug-in modules (full paths).

is_ext_pack_usable(*name*)
Check if the given extension pack is loaded and usable.
in name of type str The name of the extension pack to check for.
return usable of type bool Is the given extension pack loaded and usable.

class virtualbox.library.**IBandwidthGroup**(*interface=None*)
Represents one bandwidth group.

name
Get str value for 'name' Name of the group.

type_p
Get BandwidthGroupType value for 'type' Type of the group.

reference
Get int value for 'reference' How many devices/medium attachments use this group.

max_bytes_per_sec
Get or set int value for 'maxBytesPerSec' The maximum number of bytes which can be transfered by all entities attached to this group during one second.

class virtualbox.library.**IBandwidthControl**(*interface=None*)
Controls the bandwidth groups of one machine used to cap I/O done by a VM. This includes network and disk I/O.

num_groups
Get int value for 'numGroups' The current number of existing bandwidth groups managed.

create_bandwidth_group(*name*, *type_p*, *max_bytes_per_sec*)
Creates a new bandwidth group.
in name of type str Name of the bandwidth group.
in type_p of type *BandwidthGroupType* The type of the bandwidth group (network or disk).
in max_bytes_per_sec of type int The maximum number of bytes which can be transfered by all entities attached to this group during one second.

delete_bandwidth_group(*name*)
Deletes a new bandwidth group.
in name of type str Name of the bandwidth group to delete.

get_bandwidth_group(*name*)
Get a bandwidth group by name.
in name of type str Name of the bandwidth group to get.
return bandwidth_group of type ***IBandwidthGroup*** Where to store the bandwidth group on success.

get_all_bandwidth_groups()
Get all managed bandwidth groups.
return bandwidth_groups of type ***IBandwidthGroup*** The array of managed bandwidth groups.

class virtualbox.library.**IVirtualBoxClient**(*interface=None*)
Convenience interface for client applications. Treat this as a singleton, i.e. never create more than one instance of this interface.

At the moment only available for clients of the local API (not usable via the webservice). Once the session logic is redesigned this might change.

Error information handling is a bit special with IVirtualBoxClient: creating an instance will always succeed. The return of the actual error code/information is postponed to any attribute or method call. The reason for this is that COM likes to mutilate the error code and lose the detailed error information returned by instance creation.

virtual_box
Get IVirtualBox value for 'virtualBox' Reference to the server-side API root object.

session
Get ISession value for 'session' Create a new session object and return the reference to it.

event_source
Get IEventSource value for 'eventSource' Event source for VirtualBoxClient events.

check_machine_error(*machine*)
Perform error checking before using an *IMachine* object. Generally useful before starting a VM and all other uses. If anything is not as it should be then this method will return an appropriate error.
in machine of type ***IMachine*** The machine object to check.

class virtualbox.library.**IEventListener**(*interface=None*)
Event listener. An event listener can work in either active or passive mode, depending on the way it was registered. See *IEvent* for an introduction to VirtualBox event handling.

handle_event(*event*)
Handle event callback for active listeners. It is not called for passive listeners. After calling *handle_event()* on all active listeners and having received acknowledgement from all passive listeners via *IEventSource.event_processed()* , the event is marked as processed and *IEvent.wait_processed()* will return immediately.
in event of type ***IEvent*** Event available.

class virtualbox.library.**IEvent**(*interface=None*)
Abstract parent interface for VirtualBox events. Actual events will typically implement a more specific interface which derives from this (see below).

Introduction to VirtualBox events

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

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

An active listener is very similar to a callback: it is a function invoked by the API. As opposed to the callbacks that were used in the API before VirtualBox 4.0 however, events are now objects with an interface hierarchy.

Passive listeners are somewhat trickier to implement, but do not require a client function to be callable, which is not an option with scripting languages or web service clients. Internally the *IEventSource* implementation maintains an event queue for each passive listener, and newly arrived events are put in this queue. When the listener calls *IEventSource.get_event()* , first element from its internal event queue is returned. When the client completes processing of an event, the *IEventSource.event_processed()* function must be called, acknowledging that the event was processed. It supports implementing waitable events. On passive listener unregistration, all events from its queue are auto-acknowledged.

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

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

Using events

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

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

After registration, the callback's *IEventListener.handle_event()* method is called automatically when the event is triggered, while passive listeners have to call *IEventSource.get_event()* and *IEventSource.event_processed()* in an event processing loop.

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

type_p
Get VBoxEventType value for 'type' Event type.

source
Get IEventSource value for 'source' Source of this event.

waitable
Get bool value for 'waitable' If we can wait for this event being processed. If false, *wait_processed()* returns immediately, and *set_processed()* doesn't make sense. Non-waitable events are generally better performing, as no additional overhead associated with waitability imposed. Waitable events are needed when one need to be able to wait for particular

event processed, for example for vetoable changes, or if event refers to some resource which need to be kept immutable until all consumers confirmed events.

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

wait_processed(*timeout*)
Wait until time outs, or this event is processed. Event must be waitable for this operation to have described semantics, for non-waitable returns true immediately.
in timeout of type int Maximum time to wait for event processing, in ms; 0 = no wait, -1 = indefinite wait.
return result of type bool If this event was processed before timeout.

class virtualbox.library.**IReusableEvent**(*interface=None*)
Base abstract interface for all reusable events.

generation
Get int value for 'generation' Current generation of event, incremented on reuse.

reuse()
Marks an event as reused, increments 'generation', fields shall no longer be considered valid.

class virtualbox.library.**IMachineEvent**(*interface=None*)
Base abstract interface for all machine events.

id = VBoxEventType(3)

machine_id
Get str value for 'machineId' ID of the machine this event relates to.

class virtualbox.library.**IMachineStateChangedEvent**(*interface=None*)
Machine state change event.

id = VBoxEventType(32)

state
Get MachineState value for 'state' New execution state.

class virtualbox.library.**IMachineDataChangedEvent**(*interface=None*)
Any of the settings of the given machine has changed.

id = VBoxEventType(33)

temporary
Get bool value for 'temporary' @c true if the settings change is temporary. All permanent settings changes will trigger an event, and only temporary settings changes for running VMs will trigger an event. Note: sending events for temporary changes is NOT IMPLEMENTED.

class virtualbox.library.**IMediumRegisteredEvent**(*interface=None*)
The given medium was registered or unregistered within this VirtualBox installation. This event is not yet implemented.

id = VBoxEventType(36)

medium_id
Get str value for 'mediumId' ID of the medium this event relates to.

medium_type
Get DeviceType value for 'mediumType' Type of the medium this event relates to.

registered
Get bool value for 'registered' If @c true, the medium was registered, otherwise it was unregistered.

class virtualbox.library.**IMediumConfigChangedEvent**(*interface=None*)
The configuration of the given medium was changed (location, properties, child/parent or anything else). This event is not yet implemented.

id = VBoxEventType(96)

medium
Get IMedium value for 'medium' ID of the medium this event relates to.

class virtualbox.library.**IMachineRegisteredEvent**(*interface=None*)
The given machine was registered or unregistered within this VirtualBox installation.

id = VBoxEventType(37)

registered
Get bool value for 'registered' If @c true, the machine was registered, otherwise it was unregistered.

class virtualbox.library.**ISessionStateChangedEvent**(*interface=None*)
The state of the session for the given machine was changed. *IMachine.session_state()*

id = VBoxEventType(38)

state
Get SessionState value for 'state' New session state.

class virtualbox.library.**IGuestPropertyChangedEvent**(*interface=None*)
Notification when a guest property has changed.

id = VBoxEventType(42)

name
Get str value for 'name' The name of the property that has changed.

value
Get str value for 'value' The new property value.

flags
Get str value for 'flags' The new property flags.

class virtualbox.library.**ISnapshotEvent**(*interface=None*)
Base interface for all snapshot events.

id = VBoxEventType(4)

snapshot_id
Get str value for 'snapshotId' ID of the snapshot this event relates to.

class virtualbox.library.**ISnapshotTakenEvent**(*interface=None*)
A new snapshot of the machine has been taken. *ISnapshot*

id = VBoxEventType(39)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**ISnapshotDeletedEvent**(*interface=None*)
Snapshot of the given machine has been deleted.

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).

ISnapshot

id = VBoxEventType(40)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**ISnapshotRestoredEvent**(*interface=None*)
Snapshot of the given machine has been restored. *ISnapshot*

id = VBoxEventType(95)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**ISnapshotChangedEvent**(*interface=None*)
Snapshot properties (name and/or description) have been changed. *ISnapshot*

id = VBoxEventType(41)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IMousePointerShapeChangedEvent**(*interface=None*)
Notification when the guest mouse pointer shape has changed. The new shape data is given.

id = VBoxEventType(43)

visible
Get bool value for 'visible' Flag whether the pointer is visible.

alpha
Get bool value for 'alpha' Flag whether the pointer has an alpha channel.

xhot
Get int value for 'xhot' The pointer hot spot X coordinate.

yhot
Get int value for 'yhot' The pointer hot spot Y coordinate.

width
Get int value for 'width' Width of the pointer shape in pixels.

height
Get int value for 'height' Height of the pointer shape in pixels.

shape
Get str value for 'shape' Shape buffer arrays.

The @a shape buffer contains a 1-bpp (bits per pixel) AND mask followed by a 32-bpp XOR (color) mask.

For pointers without alpha channel the XOR mask pixels are 32-bit values: (lsb)BGR0(msb). For pointers with alpha channel the XOR mask consists of (lsb)BGRA(msb) 32-bit values.

An AND mask is used for pointers with alpha channel, so if the callback does not support alpha, the pointer could be displayed as a normal color pointer.

The AND mask is a 1-bpp bitmap with byte aligned scanlines. The size of the AND mask therefore is cbAnd = (width + 7) / 8 * height. The padding bits at the end of each scanline are undefined.

The XOR mask follows the AND mask on the next 4-byte aligned offset: uint8_t *pXor = pAnd + (cbAnd + 3) & ~3. Bytes in the gap between the AND and the XOR mask are undefined. The

XOR mask scanlines have no gap between them and the size of the XOR mask is: cXor = width * 4 * height.

If @a shape is 0, only the pointer visibility is changed.

class virtualbox.library.**IMouseCapabilityChangedEvent**(*interface=None*)
Notification when the mouse capabilities reported by the guest have changed. The new capabilities are passed.

id = VBoxEventType(44)

supports_absolute
Get bool value for 'supportsAbsolute' Supports absolute coordinates.

supports_relative
Get bool value for 'supportsRelative' Supports relative coordinates.

supports_multi_touch
Get bool value for 'supportsMultiTouch' Supports multi-touch events coordinates.

needs_host_cursor
Get bool value for 'needsHostCursor' If host cursor is needed.

class virtualbox.library.**IKeyboardLedsChangedEvent**(*interface=None*)
Notification when the guest OS executes the KBD_CMD_SET_LEDS command to alter the state of the keyboard LEDs.

id = VBoxEventType(45)

num_lock
Get bool value for 'numLock' NumLock status.

caps_lock
Get bool value for 'capsLock' CapsLock status.

scroll_lock
Get bool value for 'scrollLock' ScrollLock status.

class virtualbox.library.**IStateChangedEvent**(*interface=None*)
Notification when the execution state of the machine has changed. The new state is given.

id = VBoxEventType(46)

state
Get MachineState value for 'state' New machine state.

class virtualbox.library.**IAdditionsStateChangedEvent**(*interface=None*)
Notification when a Guest Additions property changes. Interested callees should query IGuest attributes to find out what has changed.

id = VBoxEventType(47)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**INetworkAdapterChangedEvent**(*interface=None*)
Notification when a property of one of the virtual *IMachine.get_network_adapter()* network adapters changes. Interested callees should use INetworkAdapter methods and attributes to find out what has changed.

id = VBoxEventType(48)

network_adapter
Get INetworkAdapter value for 'networkAdapter' Network adapter that is subject to change.

class virtualbox.library.**ISerialPortChangedEvent**(*interface=None*)
Notification when a property of one of the virtual *IMachine.get_serial_port()* serial ports changes. Interested callees should use ISerialPort methods and attributes to find out what has changed.

id = VBoxEventType(49)

serial_port
Get ISerialPort value for 'serialPort' Serial port that is subject to change.

class virtualbox.library.**IParallelPortChangedEvent**(*interface=None*)
Notification when a property of one of the virtual *IMachine.get_parallel_port()* parallel ports changes. Interested callees should use ISerialPort methods and attributes to find out what has changed.

id = VBoxEventType(50)

parallel_port
Get IParallelPort value for 'parallelPort' Parallel port that is subject to change.

class virtualbox.library.**IStorageControllerChangedEvent**(*interface=None*)
Notification when a *IMachine.medium_attachments()* medium attachment changes.

id = VBoxEventType(51)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IMediumChangedEvent**(*interface=None*)
Notification when a *IMachine.medium_attachments()* medium attachment changes. This event is not yet implemented.

id = VBoxEventType(52)

medium_attachment
Get IMediumAttachment value for 'mediumAttachment' Medium attachment that is subject to change.

class virtualbox.library.**IClipboardModeChangedEvent**(*interface=None*)
Notification when the shared clipboard mode changes.

id = VBoxEventType(72)

clipboard_mode
Get ClipboardMode value for 'clipboardMode' The new clipboard mode.

class virtualbox.library.**IDnDModeChangedEvent**(*interface=None*)
Notification when the drag'n drop mode changes.

id = VBoxEventType(73)

dnd_mode
Get DnDMode value for 'dndMode' The new drag'n drop mode.

class virtualbox.library.**ICPUChangedEvent**(*interface=None*)
Notification when a CPU changes.

id = VBoxEventType(60)

cpu
Get int value for 'CPU' The CPU which changed.

add
Get bool value for 'add' Flag whether the CPU was added or removed.

class virtualbox.library.**ICPUExecutionCapChangedEvent**(*interface=None*)
Notification when the CPU execution cap changes.

id = VBoxEventType(63)

execution_cap
Get int value for 'executionCap' The new CPU execution cap value. (1-100)

class virtualbox.library.**IGuestKeyboardEvent**(*interface=None*)
Notification when guest keyboard event happens.

id = VBoxEventType(64)

scancodes
Get int value for 'scancodes' Array of scancodes.

class virtualbox.library.**IGuestMouseEvent**(*interface=None*)
Notification when guest mouse event happens.

id = VBoxEventType(65)

mode
Get GuestMouseEventMode value for 'mode' If this event is relative, absolute or multi-touch.

x
Get int value for 'x' New X position, or X delta.

y
Get int value for 'y' New Y position, or Y delta.

z
Get int value for 'z' Z delta.

w
Get int value for 'w' W delta.

buttons
Get int value for 'buttons' Button state bitmask.

class virtualbox.library.**IGuestMultiTouchEvent**(*interface=None*)
Notification when guest touch screen event happens.

id = VBoxEventType(93)

contact_count
Get int value for 'contactCount' Number of contacts in the event.

x_positions
Get int value for 'xPositions' X positions.

y_positions
Get int value for 'yPositions' Y positions.

contact_ids
Get int value for 'contactIds' Contact identifiers.

contact_flags
Get int value for 'contactFlags' Contact state. Bit 0: in contact. Bit 1: in range.

scan_time
Get int value for 'scanTime' Timestamp of the event in milliseconds. Only relative time between events is important.

class virtualbox.library.**IGuestSessionEvent**(*interface=None*)
Base abstract interface for all guest session events.

session
Get IGuestSession value for 'session' Guest session that is subject to change.

class virtualbox.library.**IGuestSessionStateChangedEvent**(*interface=None*)
Notification when a guest session changed its state.

id = VBoxEventType(80)

id_p
Get int value for 'id' Session ID of guest session which was changed.

status
Get GuestSessionStatus value for 'status' New session status.

error
Get IVirtualBoxErrorInfo value for 'error' Error information in case of new session status is indicating an error.

The attribute *IVirtualBoxErrorInfo.result_detail()* will contain the runtime (IPRT) error code from the guest. See include/iprt/err.h and include/VBox/err.h for details.

class virtualbox.library.**IGuestSessionRegisteredEvent**(*interface=None*)
Notification when a guest session was registered or unregistered.

id = VBoxEventType(81)

registered
Get bool value for 'registered' If @c true, the guest session was registered, otherwise it was unregistered.

class virtualbox.library.**IGuestProcessEvent**(*interface=None*)
Base abstract interface for all guest process events.

process
Get IGuestProcess value for 'process' Guest process object which is related to this event.

pid
Get int value for 'pid' Guest process ID (PID).

class virtualbox.library.**IGuestProcessRegisteredEvent**(*interface=None*)
Notification when a guest process was registered or unregistered.

id = VBoxEventType(82)

registered
Get bool value for 'registered' If @c true, the guest process was registered, otherwise it was unregistered.

class virtualbox.library.**IGuestProcessStateChangedEvent**(*interface=None*)
Notification when a guest process changed its state.

id = VBoxEventType(83)

status
Get ProcessStatus value for 'status' New guest process status.

error
Get IVirtualBoxErrorInfo value for 'error' Error information in case of new session status is indicating an error.

The attribute *IVirtualBoxErrorInfo.result_detail()* will contain the runtime (IPRT) error code from the guest. See include/iprt/err.h and include/VBox/err.h for details.

class virtualbox.library.**IGuestProcessIOEvent**(*interface=None*)
Base abstract interface for all guest process input/output (IO) events.

handle
Get int value for 'handle' Input/output (IO) handle involved in this event. Usually 0 is stdin, 1 is stdout and 2 is stderr.

processed
Get int value for 'processed' Processed input or output (in bytes).

class virtualbox.library.**IGuestProcessInputNotifyEvent**(*interface=None*)
Notification when a guest process' stdin became available. This event is right now not implemented!

id = VBoxEventType(84)

status
Get ProcessInputStatus value for 'status' Current process input status.

class virtualbox.library.**IGuestProcessOutputEvent**(*interface=None*)
Notification when there is guest process output available for reading.

id = VBoxEventType(85)

data
Get str value for 'data' Actual output data.

class virtualbox.library.**IGuestFileEvent**(*interface=None*)
Base abstract interface for all guest file events.

file_p
Get IGuestFile value for 'file' Guest file object which is related to this event.

class virtualbox.library.**IGuestFileRegisteredEvent**(*interface=None*)
Notification when a guest file was registered or unregistered.

id = VBoxEventType(86)

registered
Get bool value for 'registered' If @c true, the guest file was registered, otherwise it was unregistered.

class virtualbox.library.**IGuestFileStateChangedEvent**(*interface=None*)
Notification when a guest file changed its state.

id = VBoxEventType(87)

status
Get FileStatus value for 'status' New guest file status.

error
Get IVirtualBoxErrorInfo value for 'error' Error information in case of new session status is indicating an error.

The attribute *IVirtualBoxErrorInfo.result_detail()* will contain the runtime (IPRT) error code from the guest. See include/iprt/err.h and include/VBox/err.h for details.

class virtualbox.library.**IGuestFileIOEvent**(*interface=None*)
Base abstract interface for all guest file input/output (IO) events.

offset
Get int value for 'offset' Current offset (in bytes).

processed
Get int value for 'processed' Processed input or output (in bytes).

class virtualbox.library.**IGuestFileOffsetChangedEvent**(*interface=None*)
Notification when a guest file changed its current offset.

id = VBoxEventType(88)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IGuestFileReadEvent**(*interface=None*)
Notification when data has been read from a guest file.

id = VBoxEventType(89)

data
Get str value for 'data' Actual data read.

class virtualbox.library.**IGuestFileWriteEvent**(*interface=None*)
Notification when data has been written to a guest file.

id = VBoxEventType(90)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IVRDEServerChangedEvent**(*interface=None*)
Notification when a property of the *IMachine.vrde_server()* VRDE server changes. Interested callees should use IVRDEServer methods and attributes to find out what has changed.

id = VBoxEventType(53)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IVRDEServerInfoChangedEvent**(*interface=None*)
Notification when the status of the VRDE server changes. Interested callees should use *IConsole.vrde_server_info()* IVRDEServerInfo attributes to find out what is the current status.

id = VBoxEventType(61)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IVideoCaptureChangedEvent**(*interface=None*)
Notification when video capture settings have changed.

id = VBoxEventType(91)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IUSBControllerChangedEvent**(*interface=None*)
Notification when a property of the virtual *IMachine.usb_controllers()* USB controllers changes. Interested callees should use IUSBController methods and attributes to find out what has changed.

id = VBoxEventType(54)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IUSBDeviceStateChangedEvent**(*interface=None*)
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.attach_usb_device()* or *IConsole.detach_usb_device()* .

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.usb_devices()* . On failure, the collection doesn't change and the @a error parameter represents the error message describing the failure.

id = VBoxEventType(55)

device
Get IUSBDevice value for 'device' Device that is subject to state change.

attached
Get bool value for 'attached' @c true if the device was attached and @c false otherwise.

error
Get IVirtualBoxErrorInfo value for 'error' @c null on success or an error message object on failure.

class virtualbox.library.**ISharedFolderChangedEvent**(*interface=None*)
Notification when a shared folder is added or removed. The @a scope argument defines one of three scopes: *IVirtualBox.shared_folders()* global shared folders (Scope.global_p Global), *IMachine.shared_folders()* permanent shared folders of the machine (Scope.machine Machine) or *IConsole.shared_folders()* transient shared folders of the machine (Scope.session Session). Interested callees should use query the corresponding collections to find out what has changed.

id = VBoxEventType(56)

scope
Get Scope value for 'scope' Scope of the notification.

class virtualbox.library.**IRuntimeErrorEvent**(*interface=None*)
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 @c 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 *IMachine.save_state()* or power it off using *IConsole.power_down()* . Resuming the execution can lead to unpredictable results.

Non-fatal errors and warnings are indicated by the @a fatal parameter set to @c false. If the virtual machine is in the Paused state by the time the error notification is received, it means that the user can *try to resume* the machine execution after attempting to solve the problem that caused the error. In this case, the notification handler is supposed to show an appropriate message to the user (depending on the value of the @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 identifiers are known:

"HostMemoryLow" "HostAudioNotResponding" "VDIStorageFull" "3DSupportIncompatibleAdditions"

id = VBoxEventType(57)

fatal
Get bool value for 'fatal' Whether the error is fatal or not.

id_p
Get str value for 'id' Error identifier.

message
Get str value for 'message' Optional error message.

class virtualbox.library.**IEventSourceChangedEvent**(*interface=None*)
Notification when an event source state changes (listener added or removed).

id = VBoxEventType(62)

listener
Get IEventListener value for 'listener' Event listener which has changed.

add
Get bool value for 'add' Flag whether listener was added or removed.

class virtualbox.library.**IExtraDataChangedEvent**(*interface=None*)
Notification when machine specific or global extra data has changed.

id = VBoxEventType(34)

machine_id
Get str value for 'machineId' ID of the machine this event relates to. Null for global extra data changes.

key
Get str value for 'key' Extra data key that has changed.

value
Get str value for 'value' Extra data value for the given key.

class virtualbox.library.**IVetoEvent**(*interface=None*)
Base abstract interface for veto events.

add_veto(*reason*)
Adds a veto on this event.
in reason of type str Reason for veto, could be null or empty string.

is_vetoed()
If this event was vetoed.
return result of type bool Reason for veto.

get_vetos()
Current veto reason list, if size is 0 - no veto.
return result of type str Array of reasons for veto provided by different event handlers.

add_approval(*reason*)
Adds an approval on this event.
in reason of type str Reason for approval, could be null or empty string.

is_approved()
If this event was approved.

return result of type bool

get_approvals()
Current approval reason list, if size is 0 - no approvals.
return result of type str Array of reasons for approval provided by different event handlers.

class virtualbox.library.**IExtraDataCanChangeEvent**(*interface=None*)
Notification when someone tries to change extra data for either the given machine or (if @c null) global extra data. This gives the chance to veto against changes.

id = VBoxEventType(35)

machine_id
Get str value for 'machineId' ID of the machine this event relates to. Null for global extra data changes.

key
Get str value for 'key' Extra data key that has changed.

value
Get str value for 'value' Extra data value for the given key.

class virtualbox.library.**ICanShowWindowEvent**(*interface=None*)
Notification when a call to *IMachine.can_show_console_window()* is made by a front-end to check if a subsequent call to *IMachine.show_console_window()* can succeed.

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

id = VBoxEventType(58)

midl_does_not_like_empty_interfaces
Get bool value for 'midlDoesNotLikeEmptyInterfaces'

class virtualbox.library.**IShowWindowEvent**(*interface=None*)
Notification when a call to *IMachine.show_console_window()* 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.

id = VBoxEventType(59)

win_id
Get or set int value for 'winId' Platform-dependent identifier of the top-level VM console window, or zero if this method has performed all actions necessary to implement the *show window* semantics for the given platform and/or this VirtualBox front-end.

class virtualbox.library.**INATRedirectEvent**(*interface=None*)
Notification when NAT redirect rule added or removed.

id = VBoxEventType(66)

slot
Get int value for 'slot' Adapter which NAT attached to.

remove
Get bool value for 'remove' Whether rule remove or add.

name
Get str value for 'name' Name of the rule.

proto
Get NATProtocol value for 'proto' Protocol (TCP or UDP) of the redirect rule.

host_ip
Get str value for 'hostIP' Host ip address to bind socket on.

host_port
Get int value for 'hostPort' Host port to bind socket on.

guest_ip
Get str value for 'guestIP' Guest ip address to redirect to.

guest_port
Get int value for 'guestPort' Guest port to redirect to.

class virtualbox.library.**IHostPCIDevicePlugEvent**(*interface=None*)
Notification when host PCI device is plugged/unplugged. Plugging usually takes place on VM startup, unplug - when *IMachine.detach_host_pci_device()* is called.

IMachine.detach_host_pci_device()

id = VBoxEventType(67)

plugged
Get bool value for 'plugged' If device successfully plugged or unplugged.

success
Get bool value for 'success' If operation was successful, if false - 'message' attribute may be of interest.

attachment
Get IPCIDeviceAttachment value for 'attachment' Attachment info for this device.

message
Get str value for 'message' Optional error message.

class virtualbox.library.**IVBoxSVCAvailabilityChangedEvent**(*interface=None*)
Notification when VBoxSVC becomes unavailable (due to a crash or similar unexpected circumstances) or available again.

id = VBoxEventType(68)

available
Get bool value for 'available' Whether VBoxSVC is available now.

class virtualbox.library.**IBandwidthGroupChangedEvent**(*interface=None*)
Notification when one of the bandwidth groups changed

id = VBoxEventType(69)

bandwidth_group
Get IBandwidthGroup value for 'bandwidthGroup' The changed bandwidth group.

class virtualbox.library.**IGuestMonitorChangedEvent**(*interface=None*)
Notification when the guest enables one of its monitors.

id = VBoxEventType(70)

change_type
Get GuestMonitorChangedEventType value for 'changeType' What was changed for this guest monitor.

screen_id
Get int value for 'screenId' The monitor which was changed.

origin_x
Get int value for 'originX' Physical X origin relative to the primary screen. Valid for Enabled and NewOrigin.

origin_y
Get int value for 'originY' Physical Y origin relative to the primary screen. Valid for Enabled and NewOrigin.

width
Get int value for 'width' Width of the screen. Valid for Enabled.

height
Get int value for 'height' Height of the screen. Valid for Enabled.

class virtualbox.library.**IGuestUserStateChangedEvent**(*interface=None*)
Notification when a guest user changed its state.

id = VBoxEventType(92)

name
Get str value for 'name' Name of the guest user whose state changed.

domain
Get str value for 'domain' Name of the FQDN (fully qualified domain name) this user is bound to. Optional.

state
Get GuestUserState value for 'state' What was changed for this guest user. See *GuestUserState* for more information.

state_details
Get str value for 'stateDetails' Optional state details, depending on the *state()* attribute.

class virtualbox.library.**IStorageDeviceChangedEvent**(*interface=None*)
Notification when a *IMachine.medium_attachments()* storage device is attached or removed.

id = VBoxEventType(71)

storage_device
Get IMediumAttachment value for 'storageDevice' Storage device that is subject to change.

removed
Get bool value for 'removed' Flag whether the device was removed or added to the VM.

silent
Get bool value for 'silent' Flag whether the guest should be notified about the change.

class virtualbox.library.**INATNetworkStartStopEvent**(*interface=None*)
IsStartEvent is true when NAT network is started and false on stopping.

id = VBoxEventType(75)

start_event
Get bool value for 'startEvent' IsStartEvent is true when NAT network is started and false on stopping.

class virtualbox.library.**IVirtualBox**(*interface=None*, *manager=None*)
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.

Error information handling is a bit special with IVirtualBox: creating an instance will always succeed. The return of the actual error code/information is postponed to any attribute or method call. The reason for this is that COM likes to mutilate the error code and lose the detailed error information returned by instance creation.

register_on_machine_state_changed(*callback*)
Set the callback function to consume on machine state changed events.

Callback receives a IMachineStateChangedEvent object.

Returns the callback_id

register_on_machine_data_changed(*callback*)
Set the callback function to consume on machine data changed events.

Callback receives a IMachineDataChangedEvent object.

Returns the callback_id

register_on_machine_registered(*callback*)
Set the callback function to consume on machine registered events.

Callback receives a IMachineRegisteredEvent object.

Returns the callback_id

register_on_snapshot_deleted(*callback*)
Set the callback function to consume on snapshot deleted events.

Callback receives a ISnapshotDeletedEvent object.

Returns the callback_id

register_on_snapshot_taken(*callback*)
Set the callback function to consume on snapshot taken events.

Callback receives a ISnapshotTakenEvent object.

Returns the callback_id

register_on_snapshot_changed(*callback*)
Set the callback function to consume on snapshot changed events which occur when snapshot properties have been changed.

Callback receives a ISnapshotChangedEvent object.

Returns the callback_id

register_on_guest_property_changed(*callback*)
Set the callback function to consume on guest property changed events.

Callback receives a IGuestPropertyChangedEvent object.

Returns the callback_id

register_on_session_state_changed(*callback*)
Set the callback function to consume on session state changed events.

Callback receives a ISessionStateChangedEvent object.

Returns the callback_id

register_on_event_source_changed(*callback*)
Set the callback function to consume on event source changed events. This occurs when a listener is added or removed.

Callback receives a IEventSourceChangedEvent object.

Returns the callback_id

register_on_extra_data_changed(*callback*)
Set the callback function to consume on extra data changed events.

Callback receives a IExtraDataChangedEvent object.

Returns the callback_id

api_revision
Get int value for 'APIRevision' To be defined exactly, but we need something that the Validation Kit can use to figure which methods and attributes can safely be used on a continuously changing trunk (and occasional branch).

api_version
Get str value for 'APIVersion' A string representing the VirtualBox API version number. The format is 2 integer numbers divided by an underscore (e.g. 1_0). After the first public release of packages with a particular API version the API will not be changed in an incompatible way. Note that this guarantee does not apply to development builds, and also there is no guarantee that this version is identical to the first two integer numbers of the package version.

check_firmware_present(*firmware_type*, *version*)
Check if this VirtualBox installation has a firmware of the given type available, either system-wide or per-user. Optionally, this may return a hint where this firmware can be downloaded from.
in firmware_type of type *FirmwareType* Type of firmware to check.
in version of type str Expected version number, usually empty string (presently ignored).
out url of type str Suggested URL to download this firmware from.
out file_p of type str Filename of firmware, only valid if result == TRUE.
return result of type bool If firmware of this type and version is available.

compose_machine_filename(*name*, *group*, *create_flags*, *base_folder*)
Returns a recommended full path of the settings file name for a new virtual machine.

This API serves two purposes:

It gets called by *create_machine()* if @c null or empty string (which is recommended) is specified for the @a settingsFile argument there, which means that API should use a recommended default file name.

It can be called manually by a client software before creating a machine, e.g. if that client wants to pre-create the machine directory to create virtual hard disks in that directory together with the new machine settings file. In that case, the file name should be stripped from the full settings file path returned by this function to obtain the machine directory.

See *IMachine.name()* and *create_machine()* for more details about the machine name.

@a groupName defines which additional subdirectory levels should be included. It must be either a valid group name or @c null or empty string which designates that the machine will not be related to a machine group.

If @a baseFolder is a @c null or empty string (which is recommended), the default machine settings folder (see *ISystemProperties.default_machine_folder()*) will be used as a base folder for the created machine, resulting in a file name like "/home/user/VirtualBox VMs/name/name.vbox". Otherwise the given base folder will be used.

This method does not access the host disks. In particular, it does not check for whether a machine with this name already exists.
in name of type str Suggested machine name.
in group of type str Machine group name for the new machine or machine group. It is used to determine the right subdirectory.
in create_flags of type str Machine creation flags, see *create_machine()* (optional).
in base_folder of type str Base machine folder (optional).
return file_p of type str Fully qualified path where the machine would be created.

create_appliance()
Creates a new appliance object, which represents an appliance in the Open Virtual Machine Format (OVF). This can then be used to import an OVF appliance into VirtualBox or to export machines as an OVF appliance; see the documentation for *IAppliance* for details.
return appliance of type *IAppliance* New appliance.

create_dhcp_server(*name*)
Creates a DHCP server settings to be used for the given internal network name
in name of type str server name
return server of type *IDHCPServer* DHCP server settings
raises *OleErrorInvalidarg* Host network interface @a name already exists.

create_machine(*settings_file*, *name*, *groups*, *os_type_id*, *flags*)
Creates a new virtual machine by creating a machine settings file at the given location.

VirtualBox machine settings files use a custom XML dialect. Starting with VirtualBox 4.0, a ".vbox" extension is recommended, but not enforced, and machine files can be created at arbitrary locations.

However, it is recommended that machines are created in the default machine folder (e.g. "/home/user/VirtualBox VMs/name/name.vbox"; see *ISystemProperties.default_machine_folder()*). If you specify @c null or empty string (which is recommended) for the @a settingsFile argument, *compose_machine_filename()* is called automatically to have such a recommended name composed based on the machine name given in the @a name argument and the primary group.

If the resulting settings file already exists, this method will fail, unless the forceOverwrite flag is set.

The new machine is created unregistered, with the initial configuration set according to the specified guest OS type. A typical sequence of actions to create a new virtual machine is as follows:

Call this method to have a new machine created. The returned machine object will be "mutable" allowing to change any machine property.

Configure the machine using the appropriate attributes and methods.

Call *IMachine.save_settings()* to write the settings to the machine's XML settings file. The configuration of the newly created machine will not be saved to disk until this method is called.

Call *register_machine()* to add the machine to the list of machines known to VirtualBox.

The specified guest OS type identifier must match an ID of one of known guest OS types listed in the *IVirtualBox.guest_os_types()* array.

IMachine.settings_modified() will return @c false for the created machine, until any of machine settings are changed.

There is no way to change the name of the settings file or subfolder of the created machine directly.

in settings_file of type str Fully qualified path where the settings file should be created, empty string or @c null for a default folder and file based on the @a name argument and the primary group. (see *compose_machine_filename()*).

in name of type str Machine name.

in groups of type str Array of group names. @c null or an empty array have the same meaning as an array with just the empty string or "/", i.e. create a machine without group association.

in os_type_id of type str Guest OS Type ID.

in flags of type str Additional property parameters, passed as a comma-separated list of "name=value" type entries. The following ones are recognized: forceOverwrite=1 to overwrite an existing machine settings file, UUID=<uuid> to specify a machine UUID and directoryIncludesUUID=1 to switch to a special VM directory naming scheme which should not be used unless necessary.

return machine of type *IMachine* Created machine object.

raises *VBoxErrorObjectNotFound* @a osTypeId is invalid.

raises *VBoxErrorFileError* Resulting settings file name is invalid or the settings file already

exists or could not be created due to an I/O error.

raises *OleErrorInvalidarg* @a name is empty or @c null.

create_medium (*format_p*, *location*, *access_mode*, *a_device_type_type*)

Creates a new base medium object that will use the given storage format and location for medium data.

The actual storage unit is not created by this method. In order to do it, and before you are able to attach the created medium to virtual machines, you must call one of the following methods to allocate a format-specific storage unit at the specified location:

IMedium.create_base_storage() *IMedium.create_diff_storage()*

Some medium attributes, such as *IMedium.id_p()* , may remain uninitialized until the medium storage unit is successfully created by one of the above methods.

Depending on the given device type, the file at the storage location must be in one of the media formats understood by VirtualBox:

With a "HardDisk" device type, the file must be a hard disk image in one of the formats supported by VirtualBox (see *ISystemProperties.medium_formats()*). After the storage unit is successfully created and this method succeeds, if the medium is a base medium, it will be added to the *hard_disks()* array attribute. With a "DVD" device type, the file must

be an ISO 9960 CD/DVD image. After this method succeeds, the medium will be added to the *dvd_images()* array attribute. With a "Floppy" device type, the file must be an RAW floppy image. After this method succeeds, the medium will be added to the *floppy_images()* array attribute.

The list of all storage formats supported by this VirtualBox installation can be obtained using *ISystemProperties.medium_formats()* . If the @a format attribute is empty or @c null then the default storage format specified by *ISystemProperties.default_hard_disk_format()* will be used for disks r creating a storage unit of the medium.

Note that the format of the location string is storage format specific. See *IMedium.location()* and IMedium for more details.

in format_p of type str Identifier of the storage format to use for the new medium.
in location of type str Location of the storage unit for the new medium.
in access_mode of type *AccessMode* Whether to open the image in read/write or read-only mode. For a "DVD" device type, this is ignored and read-only mode is always assumed.
in a_device_type_type of type *DeviceType* Must be one of "HardDisk", "DVD" or "Floppy".
return medium of type *IMedium* Created medium object.
raises *VBoxErrorObjectNotFound* @a format identifier is invalid. See
raises *VBoxErrorFileError* @a location is a not valid file name (for file-based formats only).

create_nat_network(*network_name*)
in network_name of type str

return network of type *INATNetwork*

create_shared_folder(*name*, *host_path*, *writable*, *automount*)
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.

In the current implementation, this operation is not implemented.
in name of type str Unique logical name of the shared folder.
in host_path of type str Full path to the shared folder in the host file system.
in writable of type bool Whether the share is writable or readonly
in automount of type bool Whether the share gets automatically mounted by the guest or not.

dhcp_servers
Get IDHCPServer value for 'DHCPServers' DHCP servers.

dvd_images
Get IMedium value for 'DVDImages' Array of CD/DVD image objects currently in use by this VirtualBox instance.

event_source
Get IEventSource value for 'eventSource' Event source for VirtualBox events.

extension_pack_manager
Get IExtPackManager value for 'extensionPackManager' The extension pack manager.

find_dhcp_server_by_network_name(*name*)
Searches a DHCP server settings to be used for the given internal network name
in name of type str server name
return server of type *IDHCPServer* DHCP server settings
raises *OleErrorInvalidarg* Host network interface @a name already exists.

find_machine(*name_or_id*)
Attempts to find a virtual machine given its name or UUID.

Inaccessible machines cannot be found by name, only by UUID, because their name cannot safely be determined.
in name_or_id of type str What to search for. This can either be the UUID or the name of a virtual machine.
return machine of type *IMachine* Machine object, if found.
raises *VBoxErrorObjectNotFound* Could not find registered machine matching @a nameOrId.

find_nat_network_by_name(*network_name*)
in network_name of type str

return network of type *INATNetwork*

floppy_images
Get IMedium value for 'floppyImages' Array of floppy image objects currently in use by this VirtualBox instance.

generic_network_drivers
Get str value for 'genericNetworkDrivers' Names of all generic network drivers.

get_extra_data(*key*)
Returns associated global extra data.

If the requested data @a key does not exist, this function will succeed and return an empty string in the @a value argument.
in key of type str Name of the data key to get.
return value of type str Value of the requested data key.
raises *VBoxErrorFileError* Settings file not accessible.
raises *VBoxErrorXmlError* Could not parse the settings file.

get_extra_data_keys()
Returns an array representing the global extra data keys which currently have values defined.
return keys of type str Array of extra data keys.

get_guest_os_type(*id_p*)
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.os_type_id()* attribute.

The *IVirtualBox.guest_os_types()* collection contains all available guest OS type objects. Each object has an `IGuestOSType.id_p()` attribute which contains an identifier of the guest OS this object describes.
in id_p of type str Guest OS type ID string.
return type_p of type IGuestOSType Guest OS type object.
raises *OleErrorInvalidarg* @a id is not a valid Guest OS type.

get_machine_states(*machines*)
Gets the state of several machines in a single operation.
in machines of type *IMachine* Array with the machine references.
return states of type *MachineState* Machine states, corresponding to the machines.

get_machines_by_groups(*groups*)
Gets all machine references which are in one of the specified groups.
in groups of type str What groups to match. The usual group list rules apply, i.e. passing an empty list will match VMs in the toplevel group, likewise the empty string.

return machines of type *IMachine* All machines which matched.

guest_os_types
Get IGuestOSType value for 'guestOSTypes'

hard_disks
Get IMedium value for 'hardDisks' Array of medium objects known to this VirtualBox installation.

This array contains only base media. All differencing media of the given base medium can be enumerated using *IMedium.children()* .

home_folder
Get str value for '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).

host
Get IHost value for 'host' Associated host object.

internal_networks
Get str value for 'internalNetworks' Names of all internal networks.

machine_groups
Get str value for 'machineGroups' Array of all machine group names which are used by the machines which are accessible. Each group is only listed once, however they are listed in no particular order and there is no guarantee that there are no gaps in the group hierarchy (i.e. "/", "/group/subgroup" is a valid result).

machines
Get IMachine value for 'machines' Array of machine objects registered within this VirtualBox instance.

nat_networks
Get INATNetwork value for 'NATNetworks'

open_machine(*settings_file*)
Opens a virtual machine from the existing settings file. The opened machine remains unregistered until you call *register_machine()* .

The specified settings file name must be fully qualified. The file must exist and be a valid machine XML settings file whose contents will be used to construct the machine object.

IMachine.settings_modified() will return @c false for the opened machine, until any of machine settings are changed.
in settings_file of type str Name of the machine settings file.
return machine of type *IMachine* Opened machine object.
raises *VBoxErrorFileError* Settings file name invalid, not found or sharing violation.

open_medium(*location*, *device_type*, *access_mode*, *force_new_uuid*)
Finds existing media or opens a medium from an existing storage location.
Once a medium has been opened, it can be passed to other VirtualBox methods, in particular to *IMachine.attach_device()* .

Depending on the given device type, the file at the storage location must be in one of the media formats understood by VirtualBox:

With a "HardDisk" device type, the file must be a hard disk image in one of the formats supported by VirtualBox (see *ISystemProperties.medium_formats()*). After this method succeeds, if the medium is a base medium, it will be added to the *hard_disks()* array attribute. With a "DVD" device type, the file must be an ISO 9960 CD/DVD image. After this method succeeds, the medium will be added to the *dvd_images()* array attribute. With a "Floppy" device type, the file must be an RAW floppy image. After this method succeeds, the medium will be added to the *floppy_images()* array attribute.

After having been opened, the medium can be re-found by this method and can be attached to virtual machines. See *IMedium* for more details.

The UUID of the newly opened medium will either be retrieved from the storage location, if the format supports it (e.g. for hard disk images), or a new UUID will be randomly generated (e.g. for ISO and RAW files). If for some reason you need to change the medium's UUID, use *IMedium.set_ids()* .

If a differencing hard disk medium is to be opened by this method, the operation will succeed only if its parent medium and all ancestors, if any, are already known to this VirtualBox installation (for example, were opened by this method before).

This method attempts to guess the storage format of the specified medium by reading medium data at the specified location.

If @a accessMode is ReadWrite (which it should be for hard disks and floppies), the image is opened for read/write access and must have according permissions, as VirtualBox may actually write status information into the disk's metadata sections.

Note that write access is required for all typical hard disk usage in VirtualBox, since VirtualBox may need to write metadata such as a UUID into the image. The only exception is opening a source image temporarily for copying and cloning (see *IMedium.clone_to()* when the image will be closed again soon.

The format of the location string is storage format specific. See *IMedium.location()* and IMedium for more details.

in location of type str Location of the storage unit that contains medium data in one of the supported storage formats.

in device_type of type *DeviceType* Must be one of "HardDisk", "DVD" or "Floppy".

in access_mode of type *AccessMode* Whether to open the image in read/write or read-only mode. For a "DVD" device type, this is ignored and read-only mode is always assumed.

in force_new_uuid of type bool Allows the caller to request a completely new medium UUID for the image which is to be opened. Useful if one intends to open an exact copy of a previously opened image, as this would normally fail due to the duplicate UUID.

return medium of type *IMedium* Opened medium object.

raises *VBoxErrorFileError* Invalid medium storage file location or could not find the medium

at the specified location.

raises *VBoxErrorIprtError* Could not get medium storage format.

raises *OleErrorInvalidarg* Invalid medium storage format.

raises *VBoxErrorInvalidObjectState* Medium has already been added to a media registry.

package_type

Get str value for '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.

performance_collector
Get IPerformanceCollector value for 'performanceCollector' Associated performance collector object.

progress_operations
Get IProgress value for 'progressOperations'

register_machine(*machine*)
Registers the machine previously created using *create_machine()* or opened using *open_machine()* within this VirtualBox installation. After successful method invocation, the *IMachineRegisteredEvent* event is fired.

This method implicitly calls *IMachine.save_settings()* to save all current machine settings before registering it.

in machine of type *IMachine*
raises ***VBoxErrorObjectNotFound*** No matching virtual machine found.
raises ***VBoxErrorInvalidObjectState*** Virtual machine was not created within this VirtualBox instance.

register_on_extra_data_can_change(*callback*)
Set the callback function to consume on extra data changed events.

Callback receives a IExtraDataCanChangeEvent object.

Returns the callback_id

remove_dhcp_server(*server*)
Removes the DHCP server settings
in server of type ***IDHCPServer*** DHCP server settings to be removed
raises ***OleErrorInvalidarg*** Host network interface @a name already exists.

remove_nat_network(*network*)
in network of type *INATNetwork*

remove_shared_folder(*name*)
Removes the global shared folder with the given name previously created by *create_shared_folder()* from the collection of shared folders and stops sharing it.

In the current implementation, this operation is not implemented.
in name of type str Logical name of the shared folder to remove.

revision
Get int value for 'revision' The internal build revision number of the product.

set_extra_data(*key*, *value*)
Sets associated global extra data.

If you pass @c null or empty string as a key @a value, the given @a key will be deleted.

Before performing the actual data change, this method will ask all registered event listener using the *IExtraDataCanChangeEvent* notification for a permission. If one of the listeners refuses the new value, the change will not be performed.

On success, the *IExtraDataChangedEvent* notification is called to inform all registered listeners about a successful data change.
in key of type str Name of the data key to set.
in value of type str Value to assign to the key.

raises *VBoxErrorFileError* Settings file not accessible.
raises *VBoxErrorXmlError* Could not parse the settings file.
raises *OleErrorAccessdenied* Modification request refused.

set_settings_secret(*password*)
Unlocks the secret data by passing the unlock password to the server. The server will cache the password for that machine.
in password of type str The cipher key.
raises *VBoxErrorInvalidVmState* Virtual machine is not mutable.

settings_file_path
Get str value for 'settingsFilePath' Full name of the global settings file. The value of this property corresponds to the value of *home_folder()* plus /VirtualBox.xml.

shared_folders
Get ISharedFolder value for 'sharedFolders' Collection of global shared folders. Global shared folders are available to all virtual machines.

New shared folders are added to the collection using *create_shared_folder()* . Existing shared folders can be removed using *remove_shared_folder()* .

In the current version of the product, global shared folders are not implemented and therefore this collection is always empty.

system_properties
Get ISystemProperties value for 'systemProperties' Associated system information object.

version
Get str value for '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.

This may be followed by a _ALPHA[0-9]*, _BETA[0-9]* or _RC[0-9]* tag in prerelease builds. Non-Oracle builds may (/shall) also have a publisher tag, at the end. The publisher tag starts with an underscore just like the prerelease build type tag.

version_normalized
Get str value for 'versionNormalized' A string representing the version number of the product, without the publisher information (but still with other tags). See *version()* .

class virtualbox.library.**ISession**(*interface=None*, *manager=None*)
The ISession interface represents a client process and allows for locking virtual machines (represented by IMachine objects) to prevent conflicting changes to the machine.

Any caller wishing to manipulate a virtual machine needs to create a session object first, which lives in its own process space. Such session objects are then associated with *IMachine* objects living in the VirtualBox server process to coordinate such changes.

There are two typical scenarios in which sessions are used:

To alter machine settings or control a running virtual machine, one needs to lock a machine for a given session (client process) by calling *IMachine.lock_machine()* .

Whereas multiple sessions may control a running virtual machine, only one process can obtain a write lock on the machine to prevent conflicting changes. A write lock is also needed if a process wants to actually run a virtual machine in its own context, such as the VirtualBox GUI or VBoxHeadless front-ends. They must also lock a machine for their own sessions before they are allowed to power up the virtual machine.

As a result, no machine settings can be altered while another process is already using it, either because that process is modifying machine settings or because the machine is running.

To start a VM using one of the existing VirtualBox front-ends (e.g. the VirtualBox GUI or VBoxHeadless), one would use *IMachine.launch_vm_process()* , which also takes a session object as its first parameter. This session then identifies the caller and lets the caller control the started machine (for example, pause machine execution or power it down) as well as be notified about machine execution state changes.

How sessions objects are created in a client process depends on whether you use the Main API via COM or via the webservice:

When using the COM API directly, an object of the Session class from the VirtualBox type library needs to be created. In regular COM C++ client code, this can be done by calling createLocalObject(), a standard COM API. This object will then act as a local session object in further calls to open a session.

In the webservice, the session manager (IWebsessionManager) instead creates a session object automatically whenever `IWebsessionManager.logon()` is called. A managed object reference to that session object can be retrieved by calling `IWebsessionManager.get_session_object()`.

console
Get IConsole value for 'console' Console object associated with this session. Only sessions which locked the machine for a VM process have a non-null console.

machine
Get IMachine value for 'machine' Machine object associated with this session.

name
Get or set str value for 'name' Name of this session. Important only for VM sessions, otherwise it it will be remembered, but not used for anything significant (and can be left at the empty string which is the default). The value can only be changed when the session state is SessionState_Unlocked. Make sure that you use a descriptive name which does not conflict with the VM process session names: "GUI/Qt", "GUI/SDL" and "headless".

state
Get SessionState value for 'state' Current state of this session.

type_p
Get SessionType value for 'type' Type of this session. The value of this attribute is valid only if the session currently has a machine locked (i.e. its *state()* is Locked), otherwise an error will be returned.

unlock_machine()
Unlocks a machine that was previously locked for the current session.

Calling this method is required every time a machine has been locked for a particular session using the *IMachine.launch_vm_process()* or *IMachine.lock_machine()* calls. Otherwise the state of the machine will be set to *MachineState.aborted* on the server, and changes made to the machine settings will be lost.

Generally, it is recommended to unlock all machines explicitly before terminating the application (regardless of the reason for the termination).

Do not expect the session state (*ISession.state()* to return to "Unlocked" immediately after you invoke this method, particularly if you have started a new VM process. The session state will automatically return to "Unlocked" once the VM is no longer executing, which can of course take a very long time.
raises ***OleErrorUnexpected*** Session is not locked.

class virtualbox.library.**IKeyboard**(*interface=None*)

The IKeyboard interface represents the virtual machine's keyboard. Used in *IConsole.keyboard()* .

Use this interface to send keystrokes or the Ctrl-Alt-Del sequence to the virtual machine.

event_source
Get IEventSource value for 'eventSource' Event source for keyboard events.

keyboard_le_ds
Get KeyboardLED value for 'keyboardLEDs' Current status of the guest keyboard LEDs.

put_cad()
Sends the Ctrl-Alt-Del sequence to the keyboard. This function is nothing special, it is just a convenience function calling *IKeyboard.put_scancodes()* with the proper scancodes.
raises ***VBoxErrorIprtError*** Could not send all scan codes to virtual keyboard.

put_keys(*press_keys=None*, *hold_keys=None*, *press_delay=50*)
Put scancodes that represent keys defined in the sequences provided.
Arguments: press_keys: Press a sequence of keys
hold_keys: While pressing the sequence of keys, hold down the keys defined in hold_keys.
press_delay: Number of milliseconds to delay between each press
Note: Both press_keys and hold_keys are interable objects that yield
self.SCANCODE.keys() keys.

put_scancode(*scancode*)
Sends a scancode to the keyboard.

in scancode of type int
raises ***VBoxErrorIprtError*** Could not send scan code to virtual keyboard.

put_scancodes(*scancodes*)
Sends an array of scancodes to the keyboard.

in scancodes of type int

return codes_stored of type int
raises ***VBoxErrorIprtError*** Could not send all scan codes to virtual keyboard.

release_keys()
Causes the virtual keyboard to release any keys which are currently pressed. Useful when host and guest keyboard may be out of sync.
raises ***VBoxErrorIprtError*** Could not release some or all keys.

register_on_guest_keyboard(*callback*)
Set the callback function to consume on guest keyboard events

Callback receives a IGuestKeyboardEvent object.
Example:
def callback(event): print(event.scancodes)

register_key_callback(*callback*)
Set a callback handler to consume decoded key events

Callback receives state and key where state is ON (1) or OFF (0) and a string representation for that key.
Example:
def callback(state, key): print("state = %s, key = %s" % (state, repr(key)))

class virtualbox.library.**IGuestSession**(*interface=None*)
A guest session represents one impersonated user account in the guest, so every operation will use the same credentials specified when creating the session object via *IGuest.create_session()* .

There can be a maximum of 32 sessions at once per VM, whereas session 0 is reserved for the root session. <!– r=bird: Is the root session part of the maximum of 32?? Not really clear. –> This root session is controlling all other guest sessions and also is responsible for actions which require system level privileges.

Each guest session keeps track of the guest directories and files that it opened as well as guest processes it has created. To work on guest files or directories a guest session offers methods to open or create such objects (see *IGuestSession.file_open()* or *IGuestSession.directory_open()* for instance). Similarly, there a methods for creating guest processes.

There can be up to 2048 objects (guest processes, files and directories) a time per guest session. Exceeding the limit will result in an error. <!– @todo r=bird: Add specific VBOX_E_XXX error for this and document it here! –>

When done with either of these objects, including the guest session itself, use the appropriate close() method to let the object do its cleanup work.

Closing a session via *IGuestSession.close()* will try to close all the mentioned objects above unless these objects are still used by a client.

A set of environment variables changes is associated with each session (*IGuestSession.environment_changes()*). These are applied to the base environment of the impersonated guest user when creating a new guest process. For additional flexibility the *IGuestSession.process_create()* and *IGuestSession.process_create_ex()* methods allows you to specify individual environment changes for each process you create. With newer guest addition versions, the base environment is also made available via *IGuestSession.environment_base()* . (One reason for why we record changes to a base environment instead of working directly on an environment block is that we need to be compatible with older guest additions. Another reason is that this way it is always possible to undo all the changes you've scheduled.)

execute(*command*, *arguments=None*, *stdin=''*, *environment=None*, *flags=None*, *priority=ProcessPriority(1)*, *affinity=None*, *timeout_ms=0*)

Execute a command in the Guest

Arguments: command - Command to execute. arguments - List of arguments for the command stdin - A buffer to write to the stdin of the command. environment - See IGuestSession.create_process? flags - List of ProcessCreateFlag objects.

Default value set to [wait_for_std_err, wait_for_stdout, ignore_orphaned_processes]

timeout_ms - ms to wait for the process to complete. If 0, wait for ever. . .
priority - Set the ProcessPriority priority to be used for execution.

affinity - Process affinity to use for execution.

Return IProcess, stdout, stderr

makedirs(*path*, *mode=1911*)

Super-mkdir: create a leaf directory and all intermediate ones.

directory_remove_recursive(*path*, *flags=None*)

Removes a guest directory recursively.

<!– Add this back when the warning can be removed: Unless *DirectoryRemoveRecFlag.content_and_dir* or *DirectoryRemoveRecFlag.content_only* is given, only the directory structure is removed. Which means it will fail if there are directories which are not empty in the directory tree @a path points to. –>

WARNING!! THE FLAGS ARE NOT CURRENTLY IMPLEMENTED. THE IMPLEMENTATION WORKS AS IF FLAGS WAS SET TO *DirectoryRemoveRecFlag.content_and_dir* .

If the final path component is a symbolic link, this method will fail as it can only be applied to directories.

in path of type str Path of the directory that is to be removed recursively. Guest path style.

in flags of type *DirectoryRemoveRecFlag* Zero or more *DirectoryRemoveRecFlag* flags. WARNING! SPECIFYING *DirectoryRemoveRecFlag.content_and_dir* IS MANDATORY AT THE MOMENT!!

return progress of type *IProgress* Progress object to track the operation completion. This is not implemented yet and therefore this method call will block until deletion is completed.

file_exists (*path*, *follow_symlinks=True*)

Checks whether a regular file exists in the guest or not.

in path of type str Path to the alleged regular file. Guest path style.

in follow_symlinks of type bool If @c true, symbolic links in the final component will be followed and the existance of the symlink target made the question for this method. If @c false, a symbolic link in the final component will make the method return @c false (because a symlink isn't a regular file).

return exists of type bool Returns @c true if the file exists, @c false if not. @c false is also return if this @a path does not point to a file object.

raises *VBoxErrorIprtError* Error while checking existence of the file specified.

symlink_exists (*path*, *follow_symlinks=True*)

Checks whether a symbolic link exists in the guest.

in symlink of type str Path to the alleged symbolic link. Guest path style.

return exists of type bool Returns @c true if the symbolic link exists. Returns @c false if it does not exist, if the file system object identified by the path is not a symbolic link, or if the object type is inaccessible to the user, or if the @a symlink argument is empty.

raises *OleErrorNotimpl* The method is not implemented yet.

directory_exists (*path*, *follow_symlinks=True*)

Checks whether a directory exists in the guest or not.

in path of type str Path to the directory to check if exists. Guest path style.

in follow_symlinks of type bool If @c true, symbolic links in the final component will be followed and the existance of the symlink target made the question for this method. If @c false, a symbolic link in the final component will make the method return @c false (because a symlink isn't a directory).

return exists of type bool Returns @c true if the directory exists, @c false if not.

raises *VBoxErrorIprtError* Error while checking existence of the directory specified.

path_exists (*path*, *follow_symlinks=True*)

test if path exists

close ()

Closes this session. All opened guest directories, files and processes which are not referenced by clients anymore will be closed. Guest processes which fall into this category and still are running in the guest will be terminated automatically.

current_directory

Get or set str value for 'currentDirectory' The current directory of the session. Guest path style.

directories

Get IGuestDirectory value for 'directories' Returns all currently opened guest directories.

directory_copy (*source*, *destination*, *flags*)

Recursively copies a directory from one guest location to another.

in source of type str The path to the directory to copy (in the guest). Guest path style.

in destination of type str The path to the target directory (in the guest). Unless the *DirectoryCopyFlags.copy_into_existing* flag is given, the directory shall not

already exist. Guest path style.
in flags of type ***DirectoryCopyFlags*** Zero or more *DirectoryCopyFlags* values.
return progress of type ***IProgress*** Progress object to track the operation to completion.
raises ***OleErrorNotimpl*** Not yet implemented.

directory_copy_from_guest(*source*, *destination*, *flags*)
Recursively copies a directory from the guest to the host.
in source of type str Path to the directory on the guest side that should be copied to the host. Guest path style.
in destination of type str Where to put the directory on the host. Unless the *DirectoryCopyFlags.copy_into_existing* flag is given, the directory shall not already exist. Host path style.
in flags of type ***DirectoryCopyFlags*** Zero or more *DirectoryCopyFlags* values.
return progress of type ***IProgress*** Progress object to track the operation to completion.
raises ***OleErrorNotimpl*** Not yet implemented.

directory_copy_to_guest(*source*, *destination*, *flags*)
Recursively copies a directory from the host to the guest.
in source of type str Path to the directory on the host side that should be copied to the guest. Host path style.
in destination of type str Where to put the file in the guest. Unless the *DirectoryCopyFlags.copy_into_existing* flag is given, the directory shall not already exist. Guest style path.
in flags of type ***DirectoryCopyFlags*** Zero or more *DirectoryCopyFlags* values.
return progress of type ***IProgress*** Progress object to track the operation to completion.
raises ***OleErrorNotimpl*** Not yet implemented.

directory_create(*path*, *mode*, *flags*)
Creates a directory in the guest.
in path of type str Path to the directory directory to be created. Guest path style.
in mode of type int The UNIX-style access mode mask to create the directory with. Whether/how all three access groups and associated access rights are realized is guest OS dependent. The API does the best it can on each OS.
in flags of type ***DirectoryCreateFlag*** Zero or more *DirectoryCreateFlag* flags.
raises ***VBoxErrorIprtError*** Error while creating the directory.

directory_create_temp(*template_name*, *mode*, *path*, *secure*)
Creates a temporary directory in the guest.
in template_name of type str Template for the name of the directory to create. This must contain at least one 'X' character. The first group of consecutive 'X' characters in the template will be replaced by a random alphanumeric string to produce a unique name.
in mode of type int The UNIX-style access mode mask to create the directory with. Whether/how all three access groups and associated access rights are realized is guest OS dependent. The API does the best it can on each OS.

This parameter is ignore if the @a secure parameter is set to @c true. It is strongly recommended to use 0700.
in path of type str The path to the directory in which the temporary directory should be created. Guest path style.
in secure of type bool Whether to fail if the directory can not be securely created. Currently this means that another unprivileged user cannot manipulate the path specified or remove the temporary directory after it has been created. Also causes the mode specified to be ignored. May not be supported on all guest types.
return directory of type str On success this will contain the full path to the created directory. Guest path style.

raises *VBoxErrorNotSupported* The operation is not possible as requested on this particular
guest type.
raises *OleErrorInvalidarg* Invalid argument. This includes an incorrectly formatted template,
or a non-absolute path.
raises *VBoxErrorIprtError* The temporary directory could not be created. Possible reasons
include a non-existing path or an insecure path when the secure option was requested.

directory_open(*path*, *filter_p*, *flags*)
Opens a directory in the guest and creates a *IGuestDirectory* object that can be used for further operations.

This method follows symbolic links by default at the moment, this may change in the future.
in path of type str Path to the directory to open. Guest path style.
in filter_p of type str Optional directory listing filter to apply. This uses the DOS/NT style wildcard characters '?' and '*'.
in flags of type *DirectoryOpenFlag* Zero or more *DirectoryOpenFlag* flags.
return directory of type *IGuestDirectory* *IGuestDirectory* object containing the opened directory.
raises *VBoxErrorObjectNotFound* Directory to open was not found.
raises *VBoxErrorIprtError* Error while opening the directory.

directory_remove(*path*)
Removes a guest directory if empty.

Symbolic links in the final component will not be followed, instead an not-a-directory error is reported.
in path of type str Path to the directory that should be removed. Guest path style.

domain
Get str value for 'domain' Returns the domain name used by this session to impersonate users in the guest.

environment_base
Get str value for 'environmentBase' The base environment of the session. They are on the "VAR=VALUE" form, one array entry per variable. <!– @todo/TODO/FIXME: This doesn't end up in the PDF. –>

Access fails with VBOX_E_NOT_SUPPORTED if the guest additions does not support the session base environment feature. Support for this was introduced with protocol version XXXX.

Access fails with VBOX_E_INVALID_OBJECT_STATE if the guest additions has yet to report the session base environment.

environment_changes
Get or set str value for 'environmentChanges' The set of scheduled environment changes to the base environment of the session. They are in putenv format, i.e. "VAR=VALUE" for setting and "VAR" for unsetting. One entry per variable (change). The changes are applied when creating new guest processes.

This is writable, so to undo all the scheduled changes, assign it an empty array.

environment_does_base_variable_exist(*name*)
Checks if the given environment variable exists in the session's base environment (*IGuestSession.environment_base()*).
in name of type str Name of the environment variable to look for. This cannot be empty nor can it contain any equal signs.

return exists of type bool TRUE if the variable exists, FALSE if not.
raises ***VBoxErrorNotSupported*** If the guest additions does not support the session base environment feature. Support for this was introduced with protocol version XXXX.
raises ***VBoxErrorInvalidObjectState*** If the guest additions has yet to report the session base environment.

environment_get_base_variable(*name*)
Gets an environment variable from the session's base environment (*IGuestSession.environment_base()*).
in name of type str Name of the environment variable to get.This cannot be empty nor can it contain any equal signs.
return value of type str The value of the variable. Empty if not found. To deal with variables that may have empty values, use *IGuestSession.environment_does_base_variable_exist()* .
raises ***VBoxErrorNotSupported*** If the guest additions does not support the session base environment feature. Support for this was introduced with protocol version XXXX.
raises ***VBoxErrorInvalidObjectState*** If the guest additions has yet to report the session base environment.

environment_schedule_set(*name*, *value*)
Schedules setting an environment variable when creating the next guest process. This affects the *IGuestSession.environment_changes()* attribute.
in name of type str Name of the environment variable to set. This cannot be empty nor can it contain any equal signs.
in value of type str Value to set the session environment variable to.

environment_schedule_unset(*name*)
Schedules unsetting (removing) an environment variable when creating the next guest process. This affects the *IGuestSession.environment_changes()* attribute.
in name of type str Name of the environment variable to unset. This cannot be empty nor can it contain any equal signs.

event_source
Get IEventSource value for 'eventSource' Event source for guest session events.

file_copy(*source*, *destination*, *flags*)
Copies a file from one guest location to another.

Will overwrite the destination file unless *FileCopyFlag.no_replace* is specified.
in source of type str The path to the file to copy (in the guest). Guest path style.
in destination of type str The path to the target file (in the guest). This cannot be a directory. Guest path style.
in flags of type ***FileCopyFlag*** Zero or more *FileCopyFlag* values.
return progress of type ***IProgress*** Progress object to track the operation to completion.
raises ***OleErrorNotimpl*** Not yet implemented.

file_copy_from_guest(*source*, *destination*, *flags*)
Copies a file from the guest to the host.

Will overwrite the destination file unless *FileCopyFlag.no_replace* is specified.
in source of type str Path to the file on the guest side that should be copied to the host. Guest path style.
in destination of type str Where to put the file on the host (file, not directory). Host path style.
in flags of type ***FileCopyFlag*** Zero or more *FileCopyFlag* values.
return progress of type ***IProgress*** Progress object to track the operation to completion.
raises ***VBoxErrorIprtError*** Error starting the copy operation.

file_copy_to_guest(*source*, *destination*, *flags*)
Copies a file from the host to the guest.

Will overwrite the destination file unless *FileCopyFlag.no_replace* is specified.
in source of type str Path to the file on the host side that should be copied to the guest. Host path style.
in destination of type str Where to put the file in the guest (file, not directory). Guest style path.
in flags of type *FileCopyFlag* Zero or more *FileCopyFlag* values.
return progress of type *IProgress* Progress object to track the operation to completion.
raises *VBoxErrorIprtError* Error starting the copy operation.

file_create_temp(*template_name*, *mode*, *path*, *secure*)
Creates a temporary file in the guest.
in template_name of type str Template for the name of the file to create. This must contain at least one 'X' character. The first group of consecutive 'X' characters in the template will be replaced by a random alphanumeric string to produce a unique name.
in mode of type int The UNIX-style access mode mask to create the file with. Whether/how all three access groups and associated access rights are realized is guest OS dependent. The API does the best it can on each OS.

This parameter is ignore if the @a secure parameter is set to @c true. It is strongly recommended to use 0600.
in path of type str The path to the directory in which the temporary file should be created.
in secure of type bool Whether to fail if the file can not be securely created. Currently this means that another unprivileged user cannot manipulate the path specified or remove the temporary file after it has been created. Also causes the mode specified to be ignored. May not be supported on all guest types.
return file_p of type *IGuestFile* On success this will contain an open file object for the new temporary file.
raises *VBoxErrorNotSupported* The operation is not possible as requested on this particular
guest OS.
raises *OleErrorInvalidarg* Invalid argument. This includes an incorrectly formatted template,
or a non-absolute path.
raises *VBoxErrorIprtError* The temporary file could not be created. Possible reasons include
a non-existing path or an insecure path when the secure option was requested.

file_open(*path*, *access_mode*, *open_action*, *creation_mode*)
Opens a file and creates a *IGuestFile* object that can be used for further operations.
in path of type str Path to file to open. Guest path style.
in access_mode of type *FileAccessMode* The file access mode (read, write and/or append). See *FileAccessMode* for details.
in open_action of type *FileOpenAction* What action to take depending on whether the file exists or not. See *FileOpenAction* for details.
in creation_mode of type int The UNIX-style access mode mask to create the file with if @a openAction requested the file to be created (otherwise ignored). Whether/how all three access groups and associated access rights are realized is guest OS dependent. The API does the best it can on each OS.
return file_p of type *IGuestFile* *IGuestFile* object representing the opened file.
raises *VBoxErrorObjectNotFound* File to open was not found.
raises *VBoxErrorIprtError* Error while opening the file.

file_open_ex(*path*, *access_mode*, *open_action*, *sharing_mode*, *creation_mode*, *flags*)
Opens a file and creates a *IGuestFile* object that can be used for further operations, extended version.
in path of type str Path to file to open. Guest path style.
in access_mode of type *FileAccessMode* The file access mode (read, write and/or append). See *FileAccessMode* for details.
in open_action of type *FileOpenAction* What action to take depending on whether the file exists or not. See *FileOpenAction* for details.
in sharing_mode of type *FileSharingMode* The file sharing mode in the guest. This parameter is currently ignore for all guest OSes. It will in the future be implemented for Windows, OS/2 and maybe Solaris guests only, the others will ignore it. Use *FileSharingMode.all_p*.
in creation_mode of type int The UNIX-style access mode mask to create the file with if @a openAction requested the file to be created (otherwise ignored). Whether/how all three access groups and associated access rights are realized is guest OS dependent. The API does the best it can on each OS.
in flags of type *FileOpenExFlags* Zero or more *FileOpenExFlags* values.
return file_p of type *IGuestFile* *IGuestFile* object representing the opened file.
raises *VBoxErrorObjectNotFound* File to open was not found.
raises *VBoxErrorIprtError* Error while opening the file.

file_query_size(*path*, *follow_symlinks*)
Queries the size of a regular file in the guest.
in path of type str Path to the file which size is requested. Guest path style.
in follow_symlinks of type bool It @c true, symbolic links in the final path component will be followed to their target, and the size of the target is returned. If @c false, symbolic links in the final path component will make the method call fail (symblink is not a regular file).
return size of type int Queried file size.
raises *VBoxErrorObjectNotFound* File to was not found.
raises *VBoxErrorIprtError* Error querying file size.

files
Get IGuestFile value for 'files' Returns all currently opened guest files.

fs_obj_exists(*path*, *follow_symlinks*)
Checks whether a file system object (file, directory, etc) exists in the guest or not.
in path of type str Path to the file system object to check the existance of. Guest path style.
in follow_symlinks of type bool If @c true, symbolic links in the final component will be followed and the method will instead check if the target exists. If @c false, symbolic links in the final component will satisfy the method and it will return @c true in @a exists.
return exists of type bool Returns @c true if the file exists, @c false if not.
raises *VBoxErrorIprtError* Error while checking existence of the file specified.

fs_obj_move(*source*, *destination*, *flags*)
Moves a file system object (file, directory, symlink, etc) from one guest location to another.

This differs from *IGuestSession.fs_obj_rename()* in that it can move accross file system boundraries. In that case it will perform a copy and then delete the original. For directories, this can take a while and is subject to races.
in source of type str Path to the file to move. Guest path style.
in destination of type str Where to move the file to (file, not directory). Guest path style.
in flags of type *FsObjMoveFlags* Zero or more *FsObjMoveFlags* values.
return progress of type *IProgress* Progress object to track the operation to completion.
raises *OleErrorNotimpl* Not yet implemented.

fs_obj_query_info(*path*, *follow_symlinks*)
Queries information about a file system object (file, directory, etc) in the guest.

in path of type str Path to the file system object to gather information about. Guest path style.
in follow_symlinks of type bool Information about symbolic links is returned if @c false. Otherwise, symbolic links are followed and the returned information concerns itself with the symlink target if @c true.
return info of type ***IGuestFsObjInfo*** *IGuestFsObjInfo* object containing the information.
raises ***VBoxErrorObjectNotFound*** The file system object was not found.
raises ***VBoxErrorIprtError*** Error while querying information.

fs_obj_remove (*path*)
Removes a file system object (file, symlink, etc) in the guest. Will not work on directories, use *IGuestSession.directory_remove()* to remove directories.

This method will remove symbolic links in the final path component, not follow them.
in path of type str Path to the file system object to remove. Guest style path.
raises ***OleErrorNotimpl*** The method has not been implemented yet.
raises ***VBoxErrorObjectNotFound*** The file system object was not found.
raises ***VBoxErrorIprtError*** For most other errors. We know this is unhelpful, will fix shortly...

fs_obj_rename (*old_path*, *new_path*, *flags*)
Renames a file system object (file, directory, symlink, etc) in the guest.
in old_path of type str The current path to the object. Guest path style.
in new_path of type str The new path to the object. Guest path style.
in flags of type ***FsObjRenameFlag*** Zero or more *FsObjRenameFlag* values.
raises ***VBoxErrorObjectNotFound*** The file system object was not found.
raises ***VBoxErrorIprtError*** For most other errors. We know this is unhelpful, will fix shortly...

fs_obj_set_acl (*path*, *follow_symlinks*, *acl*, *mode*)
Sets the access control list (ACL) of a file system object (file, directory, etc) in the guest.
in path of type str Full path of the file system object which ACL to set
in follow_symlinks of type bool If @c true symbolic links in the final component will be followed, otherwise, if @c false, the method will work directly on a symbolic link in the final component.
in acl of type str The ACL specification string. To-be-defined.
in mode of type int UNIX-style mode mask to use if @a acl is empty. As mention in *IGuestSession.directory_create()* this is realized on a best effort basis and the exact behavior depends on the Guest OS.
raises ***OleErrorNotimpl*** The method is not implemented yet.

id_p
Get int value for 'id' Returns the internal session ID.

name
Get str value for 'name' Returns the session's friendly name.

path_style
Get PathStyle value for 'pathStyle' The style of paths used by the guest. Handy for giving the right kind of path specifications to *IGuestSession.file_open()* and similar methods.

process_create (*executable*, *arguments*, *environment_changes*, *flags*, *timeout_ms*)
Creates a new process running in the guest. The new process will be started asynchronously, meaning on return of this function it is not be guaranteed that the guest process is in a started state. To wait for successful startup, use the *IProcess.wait_for()* call.

Starting at VirtualBox 4.2 guest process execution by is default limited to serve up to 255 guest processes at a time. If all 255 guest processes are active and running, creating a new guest

process will result in an error.

If ProcessCreateFlag_WaitForStdOut and/or ProcessCreateFlag_WaitForStdErr are set, the guest process will not enter the terminated state until all data from the specified streams have been read read.

in executable of type str Full path to the file to execute in the guest. The file has to exists in the guest VM with executable right to the session user in order to succeed. If empty/null, the first entry in the @a arguments array will be used instead (i.e. argv[0]).

in arguments of type str Array of arguments passed to the new process.

Starting with VirtualBox 5.0 this array starts with argument 0 instead of argument 1 as in previous versions. Whether the zeroth argument can be passed to the guest depends on the VBoxService version running there. If you depend on this, check that the *IGuestSession.protocol_version()* is 3 or higher.

in environment_changes of type str Set of environment changes to complement *IGuestSession.environment_changes()* . Takes precedence over the session ones. The changes are in putenv format, i.e. "VAR=VALUE" for setting and "VAR" for unsetting.

The changes are applied to the base environment of the impersonated guest user (*IGuestSession.environment_base()*) when creating the process. (This is done on the guest side of things in order to be compatible with older guest additions. That is one of the motivations for not passing in the whole environment here.)

in flags of type *ProcessCreateFlag* Process creation flags; see *ProcessCreateFlag* for more information.

in timeout_ms of type int Timeout (in ms) for limiting the guest process' running time. Pass 0 for an infinite timeout. On timeout the guest process will be killed and its status will be put to an appropriate value. See *ProcessStatus* for more information.

return guest_process of type *IGuestProcess* Guest process object of the newly created process.

raises *VBoxErrorIprtError* Error creating guest process.

process_create_ex(*executable*, *arguments*, *environment_changes*, *flags*, *timeout_ms*, *priority*, *affinity*)

Creates a new process running in the guest with the extended options for setting the process priority and affinity.

See *IGuestSession.process_create()* for more information.

in executable of type str Full path to the file to execute in the guest. The file has to exists in the guest VM with executable right to the session user in order to succeed. If empty/null, the first entry in the @a arguments array will be used instead (i.e. argv[0]).

in arguments of type str Array of arguments passed to the new process.

Starting with VirtualBox 5.0 this array starts with argument 0 instead of argument 1 as in previous versions. Whether the zeroth argument can be passed to the guest depends on the VBoxService version running there. If you depend on this, check that the *IGuestSession.protocol_version()* is 3 or higher.

in environment_changes of type str Set of environment changes to complement *IGuestSession.environment_changes()* . Takes precedence over the session ones. The changes are in putenv format, i.e. "VAR=VALUE" for setting and "VAR" for unsetting.

The changes are applied to the base environment of the impersonated guest user (*IGuestSession.environment_base()*) when creating the process. (This is done on the guest side of things in order to be compatible with older guest additions. That is one of the motivations for not passing in the whole environment here.)

in flags of type *ProcessCreateFlag* Process creation flags, see *ProcessCreateFlag* for detailed description of available flags.

in timeout_ms of type int Timeout (in ms) for limiting the guest process' running time. Pass 0 for an infinite timeout. On timeout the guest process will be killed and its status will be put to an appropriate value. See *ProcessStatus* for more information.

in priority of type *ProcessPriority* Process priority to use for execution, see *ProcessPriority* for available priority levels. This is silently ignored if not supported by guest additions.

in affinity of type int Processor affinity to set for the new process. This is a list of guest CPU numbers the process is allowed to run on.

This is silently ignored if the guest does not support setting the affinity of processes, or if the guest additions does not implemet this feature.

return guest_process of type *IGuestProcess* Guest process object of the newly created process.

process_get (*pid*)

Gets a certain guest process by its process ID (PID).

in pid of type int Process ID (PID) to get guest process for.

return guest_process of type *IGuestProcess* Guest process of specified process ID (PID).

processes

Get IGuestProcess value for 'processes' Returns all current guest processes.

protocol_version

Get int value for 'protocolVersion' Returns the protocol version which is used by this session to communicate with the guest.

status

Get GuestSessionStatus value for 'status' Returns the current session status.

symlink_create (*symlink*, *target*, *type_p*)

Creates a symbolic link in the guest.

in symlink of type str Path to the symbolic link that should be created. Guest path style.

in target of type str The path to the symbolic link target. If not an absolute, this will be relative to the @a symlink location at access time. Guest path style.

in type_p of type *SymlinkType* The symbolic link type (mainly for Windows). See *SymlinkType* for more information.

raises *OleErrorNotimpl* The method is not implemented yet.

symlink_read (*symlink*, *flags*)

Reads the target value of a symbolic link in the guest.

in symlink of type str Path to the symbolic link to read.

in flags of type *SymlinkReadFlag* Zero or more *SymlinkReadFlag* values.

return target of type str Target value of the symbolic link. Guest path style.

raises *OleErrorNotimpl* The method is not implemented yet.

timeout

Get or set int value for 'timeout' <!– r=bird: Using 'Returns' for writable attributes is misleading. –> Returns the session timeout (in ms).

user

Get str value for 'user' Returns the user name used by this session to impersonate users in the guest.

wait_for (*wait_for*, *timeout_ms*)

Waits for one or more events to happen.

in wait_for of type int Specifies what to wait for; see *GuestSessionWaitForFlag* for more information.

in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.

return reason of type ***GuestSessionWaitResult*** The overall wait result; see *GuestSessionWaitResult* for more information.

wait_for_array (*wait_for*, *timeout_ms*)

Waits for one or more events to happen. Scriptable version of *wait_for()* .

in wait_for of type ***GuestSessionWaitForFlag*** Specifies what to wait for; see *GuestSessionWaitForFlag* for more information.

in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.

return reason of type ***GuestSessionWaitResult*** The overall wait result; see *GuestSessionWaitResult* for more information.

class virtualbox.library.**IGuest** (*interface=None*)

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.

create_session (*user*, *password*, *domain=''*, *session_name='pyvbox'*, *timeout_ms=0*)

Creates a new guest session for controlling the guest. The new session will be started asynchronously, meaning on return of this function it is not guaranteed that the guest session is in a started and/or usable state. To wait for successful startup, use the *IGuestSession.wait_for()* call.

A guest session represents one impersonated user account in the guest, so every operation will use the same credentials specified when creating the session object via *IGuest.create_session()* . Anonymous sessions, that is, sessions without specifying a valid user account in the guest are not allowed reasons of security.

There can be a maximum of 32 sessions at once per VM. An error will be returned if this has been reached. <!– This should actually read: VBOX_E_IPRT_ERROR will be return if this limit has been reached. However, keep in mind that VBOX_E_IPRT_ERROR can be returned for about 88 unrelated reasons, so you don't know what happend unless you parse the error text. (bird) –> <!– @todo r=bird: Seriously, add an dedicated VBOX_E_MAX_GUEST_SESSIONS status for this condition. Do the same for all other maximums and things that could be useful to the API client. –>

For more information please consult *IGuestSession*

in user of type str User name this session will be using to control the guest; has to exist and have the appropriate rights to execute programs in the VM. Must not be empty.

in password of type str Password of the user account to be used. Empty passwords are allowed.

in domain of type str Domain name of the user account to be used if the guest is part of a domain. Optional. This feature is not implemented yet.

in session_name of type str The session's friendly name. Optional, can be empty.

return guest_session of type ***IGuestSession*** The newly created session object.

update_guest_additions (*source=None*, *arguments=None*, *flags=None*)

Automatically updates already installed Guest Additions in a VM.

At the moment only Windows guests are supported.

Because the VirtualBox Guest Additions drivers are not WHQL-certified yet there might be warning dialogs during the actual Guest Additions update. These need to be confirmed manually in order to continue the installation process. This applies to Windows 2000 and Windows XP guests and therefore these guests can't be updated

in a fully automated fashion without user interaction. However, to start a Guest Additions update for the mentioned Windows versions anyway, the flag AdditionsUpdateFlag_WaitForUpdateStartOnly can be specified. See *AdditionsUpdateFlag* for more information.

in source of type str Path to the Guest Additions .ISO file to use for the update.

in arguments of type str Optional command line arguments to use for the Guest Additions installer. Useful for retrofitting features which weren't installed before in the guest.

in flags of type *AdditionsUpdateFlag* *AdditionsUpdateFlag* flags.

return progress of type *IProgress* Progress object to track the operation completion.

raises *VBoxErrorNotSupported* Guest OS is not supported for automated Guest Additions updates or the
already installed Guest Additions are not ready yet.

raises *VBoxErrorIprtError* Error while updating.

additions_revision

Get int value for 'additionsRevision' The internal build revision number of the installed Guest Additions.

See also *IVirtualBox.revision()* .

additions_run_level

Get AdditionsRunLevelType value for 'additionsRunLevel' Current run level of the installed Guest Additions.

additions_version

Get str value for 'additionsVersion' Version of the installed Guest Additions in the same format as *IVirtualBox.version()* .

dn_d_source

Get IGuestDnDSource value for 'dnDSource' Retrieves the drag'n drop source implementation for the guest side, that is, handling and retrieving drag'n drop data from the guest.

dn_d_target

Get IGuestDnDTarget value for 'dnDTarget' Retrieves the drag'n drop source implementation for the host side. This will allow the host to handle and initiate a drag'n drop operation to copy data from the host to the guest.

event_source

Get IEventSource value for 'eventSource' Event source for guest events.

facilities

Get IAdditionsFacility value for 'facilities' Returns a collection of current known facilities. Only returns facilities where a status is known, e.g. facilities with an unknown status will not be returned.

find_session(*session_name*)

Finds guest sessions by their friendly name and returns an interface array with all found guest sessions.

in session_name of type str The session's friendly name to find. Wildcards like ? and * are allowed.

return sessions of type *IGuestSession* Array with all guest sessions found matching the name specified.

get_additions_status(*level*)

Retrieve the current status of a certain Guest Additions run level.

in level of type *AdditionsRunLevelType* Status level to check

return active of type bool Flag whether the status level has been reached or not

raises *VBoxErrorNotSupported* Wrong status level specified.

get_facility_status (*facility*)
Get the current status of a Guest Additions facility.
in facility of type *AdditionsFacilityType* Facility to check status for.
out timestamp of type int Timestamp (in ms) of last status update seen by the host.
return status of type *AdditionsFacilityStatus* The current (latest) facility status.

internal_get_statistics ()
Internal method; do not use as it might change at any time.
out cpu_user of type int Percentage of processor time spent in user mode as seen by the guest.
out cpu_kernel of type int Percentage of processor time spent in kernel mode as seen by the guest.
out cpu_idle of type int Percentage of processor time spent idling as seen by the guest.
out mem_total of type int Total amount of physical guest RAM.
out mem_free of type int Free amount of physical guest RAM.
out mem_balloon of type int Amount of ballooned physical guest RAM.
out mem_shared of type int Amount of shared physical guest RAM.
out mem_cache of type int Total amount of guest (disk) cache memory.
out paged_total of type int Total amount of space in the page file.
out mem_alloc_total of type int Total amount of memory allocated by the hypervisor.
out mem_free_total of type int Total amount of free memory available in the hypervisor.
out mem_balloon_total of type int Total amount of memory ballooned by the hypervisor.
out mem_shared_total of type int Total amount of shared memory in the hypervisor.

memory_balloon_size
Get or set int value for 'memoryBalloonSize' Guest system memory balloon size in megabytes (transient property).

os_type_id
Get str value for 'OSTypeId' Identifier of the Guest OS type as reported by the Guest Additions. You may use *IVirtualBox.get_guest_os_type()* to obtain an IGuestOSType object representing details about the given Guest OS type.

If Guest Additions are not installed, this value will be the same as *IMachine.os_type_id()* .

sessions
Get IGuestSession value for 'sessions' Returns a collection of all opened guest sessions.

set_credentials (*user_name*, *password*, *domain*, *allow_interactive_logon*)
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.
in user_name of type str User name string, can be empty
in password of type str Password string, can be empty
in domain of type str Domain name (guest logon scheme specific), can be empty
in allow_interactive_logon of type bool Flag whether the guest should alternatively allow the user to interactively specify different credentials. This flag might not be supported by all versions of the Additions.
raises *VBoxErrorVmError* VMM device is not available.

statistics_update_interval
Get or set int value for 'statisticsUpdateInterval' Interval to update guest statistics in seconds.

class virtualbox.library.**IGuestProcess** (*interface=None*)
Implementation of the *IProcess* object for processes the host has started in the guest.

arguments
Get str value for 'arguments' The arguments this process is using for execution.

environment
Get str value for 'environment' The initial process environment. Not yet implemented.

event_source
Get IEventSource value for 'eventSource' Event source for process events.

executable_path
Get str value for 'executablePath' Full path of the actual executable image.

exit_code
Get int value for 'exitCode' The exit code. Only available when the process has been terminated normally.

name
Get str value for 'name' The friendly name of this process.

pid
Get int value for 'PID' The process ID (PID).

read(*handle*, *to_read*, *timeout_ms*)
Reads data from a running process.
in handle of type int Handle to read from. Usually 0 is stdin.
in to_read of type int Number of bytes to read.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return data of type str Array of data read.

status
Get ProcessStatus value for 'status' The current process status; see *ProcessStatus* for more information.

terminate()
Terminates (kills) a running process. It can take up to 30 seconds to get a guest process killed. In case a guest process could not be killed an appropriate error is returned.

wait_for(*wait_for*, *timeout_ms=0*)
Abstract parent interface for processes handled by VirtualBox.

wait_for_array(*wait_for*, *timeout_ms*)
Waits for one or more events to happen. Scriptable version of *wait_for()* .
in wait_for of type *ProcessWaitForFlag* Specifies what to wait for; see *ProcessWaitForFlag* for more information.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return reason of type *ProcessWaitResult* The overall wait result; see *ProcessWaitResult* for more information.

write(*handle*, *flags*, *data*, *timeout_ms*)
Writes data to a running process.
in handle of type int Handle to write to. Usually 0 is stdin, 1 is stdout and 2 is stderr.
in flags of type int A combination of *ProcessInputFlag* flags.
in data of type str Array of bytes to write. The size of the array also specifies how much to write.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return written of type int How much bytes were written.

write_array(*handle*, *flags*, *data*, *timeout_ms*)
Writes data to a running process. Scriptable version of *write()* .
in handle of type int Handle to write to. Usually 0 is stdin, 1 is stdout and 2 is stderr.
in flags of type *ProcessInputFlag* A combination of *ProcessInputFlag* flags.
in data of type str Array of bytes to write. The size of the array also specifies how much to write.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return written of type int How much bytes were written.

class virtualbox.library.**IMachine**(*interface=None*)
The IMachine interface represents a virtual machine, or guest, created in VirtualBox.

This interface is used in two contexts. First of all, a collection of objects implementing this interface is stored in the *IVirtualBox.machines()* attribute which lists all the virtual machines that are currently registered with this VirtualBox installation. Also, once a session has been opened for the given virtual machine (e.g. the virtual machine is running), the machine object associated with the open session can be queried from the session object; see *ISession* for details.

The main role of this interface is to expose the settings of the virtual machine and provide methods to change various aspects of the virtual machine's configuration. For machine objects stored in the *IVirtualBox.machines()* collection, all attributes are read-only unless explicitly stated otherwise in individual attribute and method descriptions.

In order to change a machine setting, a session for this machine must be opened using one of the *IMachine.lock_machine()* or *IMachine.launch_vm_process()* methods. After the machine has been successfully locked for a session, a mutable machine object needs to be queried from the session object and then the desired settings changes can be applied to the returned object using IMachine attributes and methods. See the *ISession* interface description for more information about sessions.

Note that IMachine does not provide methods to control virtual machine execution (such as start the machine, or power it down) – these methods are grouped in a separate interface called *IConsole* .

ISession, *IConsole*

accelerate2_d_video_enabled
Get or set bool value for 'accelerate2DVideoEnabled' This setting determines whether VirtualBox allows this machine to make use of the 2D video acceleration support available on the host.

accelerate3_d_enabled
Get or set bool value for 'accelerate3DEnabled' This setting determines whether VirtualBox allows this machine to make use of the 3D graphics support available on the host.

access_error
Get IVirtualBoxErrorInfo value for 'accessError' Error information describing the reason of machine inaccessibility.

Reading this property is only valid after the last call to *accessible()* returned @c false (i.e. the machine is currently inaccessible). Otherwise, a @c null IVirtualBoxErrorInfo object will be returned.

accessible
Get bool value for 'accessible' Whether this virtual machine is currently accessible or not.

A machine is always deemed accessible unless it is registered *and* its settings file cannot be read or parsed (either because the file itself is unavailable or has invalid XML contents).

Every time this property is read, the accessibility state of this machine is re-evaluated. If the

returned value is @c false, the *access_error()* property may be used to get the detailed error information describing the reason of inaccessibility, including XML error messages.

When the machine is inaccessible, only the following properties can be used on it:

parent() id_p() settings_file_path() accessible() access_error()

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 *IMachine.unregister()* call (or, to check for the accessibility state once more by querying this property).

In the current implementation, once this property returns @c 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.

add_storage_controller(*name*, *connection_type*)

Adds a new storage controller (SCSI, SAS or SATA controller) to the machine and returns it as an instance of *IStorageController* .

@a name identifies the controller for subsequent calls such as *get_storage_controller_by_name()* , *get_storage_controller_by_instance()* , *remove_storage_controller()* , *attach_device()* or *mount_medium()* .

After the controller has been added, you can set its exact type by setting the *IStorageController.controller_type()* .

in name of type str

in connection_type of type *StorageBus*

return controller of type *IStorageController*
raises ***VBoxErrorObjectInUse*** A storage controller with given name exists already.
raises ***OleErrorInvalidarg*** Invalid @a controllerType.

add_usb_controller(*name*, *type_p*)

Adds a new USB controller to the machine and returns it as an instance of *IUSBController* .

in name of type str

in type_p of type *USBControllerType*

return controller of type *IUSBController*
raises ***VBoxErrorObjectInUse*** A USB controller with given type exists already.
raises ***OleErrorInvalidarg*** Invalid @a controllerType.

adopt_saved_state(*saved_state_file*)

Associates the given saved state file to the virtual machine.

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

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

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

raises *VBoxErrorInvalidVmState* Virtual machine state neither PoweredOff nor Aborted.

allow_tracing_to_access_vm
Get or set bool value for 'allowTracingToAccessVM' Enables tracepoints in PDM devices and drivers to use the VMCPU or VM structures when firing off trace points. This is especially useful with DTrace tracepoints, as it allows you to use the VMCPU or VM pointer to obtain useful information such as guest register state.

This is disabled by default because devices and drivers normally has no business accessing the VMCPU or VM structures, and are therefore unable to get any pointers to these.

apply_defaults (*flags*)
Applies the defaults for the configured guest OS type. This is primarily for getting sane settings straight after creating a new VM, but it can also be applied later.

This is primarily a shortcut, centralizing the tedious job of getting the recommended settings and translating them into settings updates. The settings are made at the end of the call, but not saved.
in flags of type str Additional flags, to be defined later.
raises *OleErrorNotimpl* This method is not implemented yet.

attach_device (*name*, *controller_port*, *device*, *type_p*, *medium*)
Attaches a device and optionally mounts a medium to the given storage controller (*IStorageController* , identified by @a name), at the indicated port and device.

This method is intended for managing storage devices in general while a machine is powered off. It can be used to attach and detach fixed and removable media. The following kind of media can be attached to a machine:

For fixed and removable media, you can pass in a medium that was previously opened using *IVirtualBox.open_medium()* .

Only for storage devices supporting removable media (such as DVDs and floppies), you can also specify a null pointer to indicate an empty drive or one of the medium objects listed in the `IHost.dvd_drives()` and `IHost.floppy_drives()` arrays to indicate a host drive. For removable devices, you can also use *IMachine.mount_medium()* to change the media while the machine is running.

In a VM's default configuration of virtual machines, the secondary master of the IDE controller is used for a CD/DVD drive.

After calling this returns successfully, a new instance of *IMediumAttachment* will appear in the machine's list of medium attachments (see *IMachine.medium_attachments()*).

See *IMedium* and *IMediumAttachment* for more information about attaching media.

The specified device slot must not have a device attached to it, or this method will fail.

You cannot attach a device to a newly created machine until this machine's settings are saved to disk using *save_settings()* .

If the medium is being attached indirectly, a new differencing medium will implicitly be created for it and attached instead. If the changes made to the machine settings (including this indirect attachment) are later cancelled using *discard_settings()* , this implicitly created differencing medium will implicitly be deleted.
in name of type str Name of the storage controller to attach the device to.
in controller_port of type int Port to attach the device to. For an IDE controller, 0 specifies the primary controller and 1 specifies the secondary controller. For a SCSI controller,

this must range from 0 to 15; for a SATA controller, from 0 to 29; for an SAS controller, from 0 to 7.

in device of type int Device slot in the given port to attach the device to. This is only relevant for IDE controllers, for which 0 specifies the master device and 1 specifies the slave device. For all other controller types, this must be 0.

in type_p of type *DeviceType* Device type of the attached device. For media opened by *IVirtualBox.open_medium()* , this must match the device type specified there.

in medium of type *IMedium* Medium to mount or @c null for an empty drive.

raises *OleErrorInvalidarg* SATA device, SATA port, IDE port or IDE slot out of range, or

file or UUID not found.

raises *VBoxErrorInvalidObjectState* Machine must be registered before media can be attached.

raises *VBoxErrorInvalidVmState* Invalid machine state.

raises *VBoxErrorObjectInUse* A medium is already attached to this or another virtual machine.

attach_device_without_medium(*name*, *controller_port*, *device*, *type_p*)

Attaches a device and optionally mounts a medium to the given storage controller (*IStorageController* , identified by @a name), at the indicated port and device.

This method is intended for managing storage devices in general while a machine is powered off. It can be used to attach and detach fixed and removable media. The following kind of media can be attached to a machine:

For fixed and removable media, you can pass in a medium that was previously opened using *IVirtualBox.open_medium()* .

Only for storage devices supporting removable media (such as DVDs and floppies) with an empty drive or one of the medium objects listed in the `IHost.dvd_drives()` and `IHost.floppy_drives()` arrays to indicate a host drive. For removable devices, you can also use *IMachine.mount_medium()* to change the media while the machine is running.

In a VM's default configuration of virtual machines, the secondary master of the IDE controller is used for a CD/DVD drive. *IMediumAttachment* will appear in the machine's list of medium attachments (see *IMachine.medium_attachments()*).

See *IMedium* and *IMediumAttachment* for more information about attaching media.

The specified device slot must not have a device attached to it, or this method will fail.

You cannot attach a device to a newly created machine until this machine's settings are saved to disk using *save_settings()* .

If the medium is being attached indirectly, a new differencing medium will implicitly be created for it and attached instead. If the changes made to the machine settings (including this indirect attachment) are later cancelled using *discard_settings()* , this implicitly created differencing medium will implicitly be deleted.

in name of type str Name of the storage controller to attach the device to.

in controller_port of type int Port to attach the device to. For an IDE controller, 0 specifies the primary controller and 1 specifies the secondary controller. For a SCSI controller, this must range from 0 to 15; for a SATA controller, from 0 to 29; for an SAS controller, from 0 to 7.

in device of type int Device slot in the given port to attach the device to. This is only relevant for IDE controllers, for which 0 specifies the master device and 1 specifies the slave device. For all other controller types, this must be 0.

in type_p of type ***DeviceType*** Device type of the attached device. For media opened by *IVirtualBox.open_medium()* , this must match the device type specified there.
raises ***OleErrorInvalidarg*** SATA device, SATA port, IDE port or IDE slot out of range, or
file or UUID not found.
raises ***VBoxErrorInvalidObjectState*** Machine must be registered before media can be attached.
raises ***VBoxErrorInvalidVmState*** Invalid machine state.
raises ***VBoxErrorObjectInUse*** A medium is already attached to this or another virtual machine.

attach_host_pci_device(*host_address*, *desired_guest_address*, *try_to_unbind*)
Attaches host PCI device with the given (host) PCI address to the PCI bus of the virtual machine. Please note, that this operation is two phase, as real attachment will happen when VM will start, and most information will be delivered as IHostPCIDevicePlugEvent on IVirtualBox event source.

IHostPCIDevicePlugEvent
in host_address of type int Address of the host PCI device.
in desired_guest_address of type int Desired position of this device on guest PCI bus.
in try_to_unbind of type bool If VMM shall try to unbind existing drivers from the device before attaching it to the guest.
raises ***VBoxErrorInvalidVmState*** Virtual machine state is not stopped (PCI hotplug not yet implemented).
raises ***VBoxErrorPdmError*** Virtual machine does not have a PCI controller allowing attachment of physical devices.
raises ***VBoxErrorNotSupported*** Hardware or host OS doesn't allow PCI device passthrough.

audio_adapter
Get IAudioAdapter value for 'audioAdapter' Associated audio adapter, always present.

autostart_delay
Get or set int value for 'autostartDelay' Number of seconds to wait until the VM should be started during system boot.

autostart_enabled
Get or set bool value for 'autostartEnabled' Enables autostart of the VM during system boot.

autostop_type
Get or set AutostopType value for 'autostopType' Action type to do when the system is shutting down.

bandwidth_control
Get IBandwidthControl value for 'bandwidthControl' Bandwidth control manager.

bios_settings
Get IBIOSSettings value for 'BIOSSettings' Object containing all BIOS settings.

can_show_console_window()
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.

This method will fail if a session for this machine is not currently open.
return can_show of type bool @c true if the console window can be shown and @c false otherwise.
raises ***VBoxErrorInvalidVmState*** Machine session is not open.

chipset_type
Get or set ChipsetType value for 'chipsetType' Chipset type used in this VM.

clipboard_mode
Get or set ClipboardMode value for 'clipboardMode' Synchronization mode between the host OS clipboard and the guest OS clipboard.

clone(*snapshot_name_or_id=None*, *mode=CloneMode(1)*, *options=None*, *name=None*, *uuid=None*, *groups=None*, *basefolder=''*, *register=True*)
Clone this Machine
Options: snapshot_name_or_id - value can be either ISnapshot, name, or id mode - set the CloneMode value options - define the CloneOptions options name - define a name of the new VM uuid - set the uuid of the new VM groups - specify which groups the new VM will exist under basefolder - specify which folder to set the VM up under register - register this VM with the server
Note: Default values create a linked clone from the current machine state
Return a IMachine object for the newly cloned vm

clone_to(*target*, *mode*, *options*)
Creates a clone of this machine, either as a full clone (which means creating independent copies of the hard disk media, save states and so on), or as a linked clone (which uses its own differencing media, sharing the parent media with the source machine).

The target machine object must have been created previously with *IVirtualBox.create_machine()* , and all the settings will be transferred except the VM name and the hardware UUID. You can set the VM name and the new hardware UUID when creating the target machine. The network MAC addresses are newly created for all enabled network adapters. You can change that behaviour with the options parameter. The operation is performed asynchronously, so the machine object will be not be usable until the @a progress object signals completion.
in target of type ***IMachine*** Target machine object.
in mode of type ***CloneMode*** Which states should be cloned.
in options of type ***CloneOptions*** Options for the cloning operation.
return progress of type ***IProgress*** Progress object to track the operation completion.
raises ***OleErrorInvalidarg*** @a target is @c null.

cpu_count
Get or set int value for 'CPUCount' Number of virtual CPUs in the VM.

cpu_execution_cap
Get or set int value for 'CPUExecutionCap' Means to limit the number of CPU cycles a guest can use. The unit is percentage of host CPU cycles per second. The valid range is 1 - 100. 100 (the default) implies no limit.

cpu_hot_plug_enabled
Get or set bool value for 'CPUHotPlugEnabled' This setting determines whether VirtualBox allows CPU hotplugging for this machine.

cpu_profile
Get or set str value for 'CPUProfile' Experimental feature to select the guest CPU profile. The default is "host", which indicates the host CPU. All other names are subject to change.

The profiles are found in src/VBox/VMM/VMMR3/cpus/.

cpuid_portability_level
Get or set int value for 'CPUIDPortabilityLevel' Virtual CPUID portability level, the higher number the fewer newer or vendor specific CPU feature is reported to the guest (via the CPUID instruction). The default level of zero (0) means that all virtualized feautres supported by the

host is pass thru to the guest. While the three (3) is currently the level supressing the most features.

Exactly which of the CPUID features are left out by the VMM at which level is subject to change with each major version.

create_session(*lock_type=LockType(1)*, *session=None*)
Lock this machine
Arguments: lock_type - see IMachine.lock_machine for details session - optionally define a session object to lock this machine
against. If not defined, a new ISession object is created to lock against
return an ISession object

create_shared_folder(*name*, *host_path*, *writable*, *automount*)
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.
in name of type str Unique logical name of the shared folder.
in host_path of type str Full path to the shared folder in the host file system.
in writable of type bool Whether the share is writable or read-only.
in automount of type bool Whether the share gets automatically mounted by the guest or not.
raises *VBoxErrorObjectInUse* Shared folder already exists.
raises *VBoxErrorFileError* Shared folder @a hostPath not accessible.

current_snapshot
Get ISnapshot value for 'currentSnapshot' Current snapshot of this machine. This is @c null if the machine currently has no snapshots. If it is not @c null, then it was set by one of *take_snapshot()* , *delete_snapshot()* or *restore_snapshot()* , depending on which was called last. See *ISnapshot* for details.

current_state_modified
Get bool value for 'currentStateModified' Returns @c true if the current state of the machine is not identical to the state stored in the current snapshot.

The current state is identical to the current snapshot only directly after one of the following calls are made:

restore_snapshot()

take_snapshot() (issued on a "powered off" or "saved" machine, for which *settings_modified()* returns @c false)

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

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

For machines that don't have snapshots, this property is always @c false.

default_frontend
Get or set str value for 'defaultFrontend' Selects which VM frontend should be used by default when launching this VM through the *IMachine.launch_vm_process()* method. Empty or @c null strings do not define a particular default, it is up to *IMachine.launch_vm_process()* to select one. See the description of *IMachine.launch_vm_process()* for the valid frontend types.

This per-VM setting overrides the default defined by *ISystemProperties.default_frontend()* attribute, and is overridden by a frontend type passed to *IMachine.launch_vm_process()* .

delete_config(*media*)
Deletes the files associated with this machine from disk. If medium objects are passed in with the @a aMedia argument, they are closed and, if closing was successful, their storage files are deleted as well. For convenience, this array of media files can be the same as the one returned from a previous *unregister()* call.

This method must only be called on machines which are either write-locked (i.e. on instances returned by *ISession.machine()*) or on unregistered machines (i.e. not yet registered machines created by *IVirtualBox.create_machine()* or opened by *IVirtualBox.open_machine()* , or after having called *unregister()*).

The following files will be deleted by this method:

If *unregister()* had been previously called with a @a cleanupMode argument other than "UnregisterOnly", this will delete all saved state files that the machine had in use; possibly one if the machine was in "Saved" state and one for each online snapshot that the machine had. On each medium object passed in the @a aMedia array, this will call *IMedium.close()* . If that succeeds, this will attempt to delete the medium's storage on disk. Since the *IMedium.close()* call will fail if the medium is still in use, e.g. because it is still attached to a second machine; in that case the storage will not be deleted. Finally, the machine's own XML file will be deleted.

Since deleting large disk image files can be a time-consuming I/O operation, this method operates asynchronously and returns an IProgress object to allow the caller to monitor the progress. There will be one sub-operation for each file that is being deleted (saved state or medium storage file).

settings_modified() will return @c true after this method successfully returns.
in media of type ***IMedium*** List of media to be closed and whose storage files will be deleted.
return progress of type ***IProgress*** Progress object to track the operation completion.
raises ***VBoxErrorInvalidVmState*** Machine is registered but not write-locked.
raises ***VBoxErrorIprtError*** Could not delete the settings file.

delete_guest_property(*name*)
Deletes an entry from the machine's guest property store.
in name of type str The name of the property to delete.
raises ***VBoxErrorInvalidVmState*** Machine session is not open.

delete_snapshot(*id_p*)
Starts deleting the specified snapshot asynchronously. See *ISnapshot* for an introduction to snapshots.

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

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

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

Deleting a snapshot has the following preconditions:

Child media of all normal media of the deleted snapshot must be accessible (see *IMedium.state()*) for this operation to succeed. If only one running VM refers to all images which participates in merging the operation can be performed while the VM is running. Otherwise all virtual machines whose media are directly or indirectly based on the media of deleted snapshot must be powered off. In any case, online snapshot deleting usually is slower than the same operation without any running VM.

You cannot delete the snapshot if a medium attached to it has more than one child medium (differencing images) because otherwise merging would be impossible. This might be the case if there is more than one child snapshot or differencing images were created for other reason (e.g. implicitly because of multiple machine attachments).

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

Merging medium contents can be very time and disk space consuming, if these media are big in size and have many children. However, if the snapshot being deleted is the last (head) snapshot on the branch, the operation will be rather quick.
in id_p of type str UUID of the snapshot to delete.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* The running virtual machine prevents deleting this snapshot. This
happens only in very specific situations, usually snapshots can be deleted without trouble while a VM is running. The error message text explains the reason for the failure.

delete_snapshot_and_all_children(*id_p*)
Starts deleting the specified snapshot and all its children asynchronously. See *ISnapshot* for an introduction to snapshots. The conditions and many details are the same as with *delete_snapshot()* .

This operation is very fast if the snapshot subtree does not include the current state. It is still significantly faster than deleting the snapshots one by one if the current state is in the subtree and there are more than one snapshots from current state to the snapshot which marks the subtree, since it eliminates the incremental image merging.

This API method is right now not implemented!
in id_p of type str UUID of the snapshot to delete, including all its children.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* The running virtual machine prevents deleting this snapshot. This
happens only in very specific situations, usually snapshots can be deleted without trouble while a VM is running. The error message text explains the reason for the failure.
raises *OleErrorNotimpl* The method is not implemented yet.

delete_snapshot_range(*start_id*, *end_id*)
Starts deleting the specified snapshot range. This is limited to linear snapshot lists, which means there may not be any other child snapshots other than the direct sequence between the start and end snapshot. If the start and end snapshot point to the same snapshot this method is completely equivalent to *delete_snapshot()* . See *ISnapshot* for an introduction to snapshots. The conditions and many details are the same as with *delete_snapshot()* .

This operation is generally faster than deleting snapshots one by one and often also needs less extra disk space before freeing up disk space by deleting the removed disk images corresponding to the snapshot.

This API method is right now not implemented!
in start_id of type str UUID of the first snapshot to delete.

in end_id of type str UUID of the last snapshot to delete.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* The running virtual machine prevents deleting this snapshot. This
happens only in very specific situations, usually snapshots can be deleted without trouble while a VM is running. The error message text explains the reason for the failure.
raises *OleErrorNotimpl* The method is not implemented yet.

description
Get or set str value for '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).

detach_device(*name*, *controller_port*, *device*)
Detaches the device attached to a device slot of the specified bus.
Detaching the device from the virtual machine is deferred. This means that the medium remains associated with the machine when this method returns and gets actually deassociated only after a successful *save_settings()* call. See *IMedium* for more detailed information about attaching media.

You cannot detach a device from a running machine.

Detaching differencing media implicitly created by *attach_device()* for the indirect attachment using this method will **not** implicitly delete them. The *IMedium.delete_storage()* operation should be explicitly performed by the caller after the medium is successfully detached and the settings are saved with *save_settings()* , if it is the desired action.
in name of type str Name of the storage controller to detach the medium from.
in controller_port of type int Port number to detach the medium from.
in device of type int Device slot number to detach the medium from.
raises *VBoxErrorInvalidVmState* Attempt to detach medium from a running virtual machine.
raises *VBoxErrorObjectNotFound* No medium attached to given slot/bus.
raises *VBoxErrorNotSupported* Medium format does not support storage deletion (only for implicitly
created differencing media, should not happen).

detach_host_pci_device(*host_address*)
Detach host PCI device from the virtual machine. Also HostPCIDevicePlugEvent on IVirtualBox event source will be delivered. As currently we don't support hot device unplug, IHostPCIDevicePlugEvent event is delivered immediately.

IHostPCIDevicePlugEvent
in host_address of type int Address of the host PCI device.
raises *VBoxErrorInvalidVmState* Virtual machine state is not stopped (PCI hotplug not yet implemented).
raises *VBoxErrorObjectNotFound* This host device is not attached to this machine.
raises *VBoxErrorPdmError* Virtual machine does not have a PCI controller allowing attachment of physical devices.
raises *VBoxErrorNotSupported* Hardware or host OS doesn't allow PCI device passthrough.

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

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

If @a fRemoveFile is @c true, the file in the machine directory into which the machine state was saved is also deleted. If this is @c false, then the state can be recovered and later re-inserted into a machine using *adopt_saved_state()* . The location of the file can be found in the *state_file_path()* attribute.

in f_remove_file of type bool Whether to also remove the saved state file.

raises *VBoxErrorInvalidVmState* Virtual machine not in state Saved.

discard_settings()

Discards any changes to the machine settings made since the session has been opened or since the last call to *save_settings()* or *discard_settings()* .

Calling this method is only valid on instances returned by *ISession.machine()* and on new machines created by *IVirtualBox.create_machine()* or opened by *IVirtualBox.open_machine()* but not yet registered, or on unregistered machines after calling *IMachine.unregister()* .

raises *VBoxErrorInvalidVmState* Virtual machine is not mutable.

dn_d_mode

Get or set DnDMode value for 'dnDMode' Sets or retrieves the current drag'n drop mode.

emulated_usb_card_reader_enabled

Get or set bool value for 'emulatedUSBCardReaderEnabled'

enumerate_guest_properties(*patterns*)

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

in patterns of type str The patterns to match the properties against, separated by '|' characters. If this is empty or @c null, all properties will match.

out names of type str The names of the properties returned.

out values of type str The values of the properties returned. The array entries match the corresponding entries in the @a name array.

out timestamps of type int The time stamps of the properties returned. The array entries match the corresponding entries in the @a name array.

out flags of type str The flags of the properties returned. The array entries match the corresponding entries in the @a name array.

export_to(*appliance*, *location*)

Exports the machine to an OVF appliance. See *IAppliance* for the steps required to export VirtualBox machines to OVF.

in appliance of type *IAppliance* Appliance to export this machine to.

in location of type str The target location.

return description of type *IVirtualSystemDescription* VirtualSystemDescription object which is created for this machine.

fault_tolerance_address

Get or set str value for 'faultToleranceAddress' The address the fault tolerance source or target.

fault_tolerance_password

Get or set str value for 'faultTolerancePassword' The password to check for on the standby VM. This is just a very basic measure to prevent simple hacks and operators accidentally choosing the wrong standby VM.

fault_tolerance_port

Get or set int value for 'faultTolerancePort' The TCP port the fault tolerance source or target will use for communication.

fault_tolerance_state
Get or set FaultToleranceState value for 'faultToleranceState' Fault tolerance state; disabled, source or target. This property can be changed at any time. If you change it for a running VM, then the fault tolerance address and port must be set beforehand.

fault_tolerance_sync_interval
Get or set int value for 'faultToleranceSyncInterval' The interval in ms used for syncing the state between source and target.

find_snapshot (*name_or_id*)
Returns a snapshot of this machine with the given name or UUID.

Returns a snapshot of this machine with the given UUID. A @c null argument can be used to obtain the first snapshot taken on this machine. To traverse the whole tree of snapshots starting from the root, inspect the root snapshot's *ISnapshot.children()* attribute and recurse over those children.
in name_or_id of type str What to search for. Name or UUID of the snapshot to find
return snapshot of type *ISnapshot* Snapshot object with the given name.
raises *VBoxErrorObjectNotFound* Virtual machine has no snapshots or snapshot not found.

firmware_type
Get or set FirmwareType value for 'firmwareType' Type of firmware (such as legacy BIOS or EFI), used for initial bootstrap in this VM.

get_boot_order (*position*)
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()
in position of type int Position in the boot order (@c 1 to the total number of devices the machine can boot from, as returned by *ISystemProperties.max_boot_position()*).
return device of type *DeviceType* Device at the given position.
raises *OleErrorInvalidarg* Boot @a position out of range.

get_cpu_property (*property_p*)
Returns the virtual CPU boolean value of the specified property.
in property_p of type *CPUPropertyType* Property type to query.
return value of type bool Property value.
raises *OleErrorInvalidarg* Invalid property.

get_cpu_status (*cpu*)
Returns the current status of the given CPU.
in cpu of type int The CPU id to check for.
return attached of type bool Status of the CPU.

get_cpuid_leaf (*id_p*)
Returns the virtual CPU cpuid information for the specified leaf.

Currently supported index values for cpuid: Standard CPUID leafs: 0 - 0xA Extended CPUID leafs: 0x80000000 - 0x8000000A

See the Intel and AMD programmer's manuals for detailed information about the cpuid instruction and its leafs.

in id_p of type int CPUID leaf index.
out val_eax of type int CPUID leaf value for register eax.
out val_ebx of type int CPUID leaf value for register ebx.
out val_ecx of type int CPUID leaf value for register ecx.
out val_edx of type int CPUID leaf value for register edx.
raises *OleErrorInvalidarg* Invalid id.

get_effective_paravirt_provider()
Returns the effective paravirtualization provider for this VM.
return paravirt_provider of type *ParavirtProvider* The effective paravirtualization provider for this VM.

get_extra_data(*key*)
Returns associated machine-specific extra data.

If the requested data @a key does not exist, this function will succeed and return an empty string in the @a value argument.
in key of type str Name of the data key to get.
return value of type str Value of the requested data key.
raises *VBoxErrorFileError* Settings file not accessible.
raises *VBoxErrorXmlError* Could not parse the settings file.

get_extra_data_keys()
Returns an array representing the machine-specific extra data keys which currently have values defined.
return keys of type str Array of extra data keys.

get_guest_property(*name*)
Reads an entry from the machine's guest property store.
in name of type str The name of the property to read.
out value of type str The value of the property. If the property does not exist then this will be empty.
out timestamp of type int The time at which the property was last modified, as seen by the server process.
out flags of type str Additional property parameters, passed as a comma-separated list of "name=value" type entries.
raises *VBoxErrorInvalidVmState* Machine session is not open.

get_guest_property_timestamp(*property_p*)
Reads a property timestamp from the machine's guest property store.
in property_p of type str The name of the property to read.
return value of type int The timestamp. If the property does not exist then this will be empty.
raises *VBoxErrorInvalidVmState* Machine session is not open.

get_guest_property_value(*property_p*)
Reads a value from the machine's guest property store.
in property_p of type str The name of the property to read.
return value of type str The value of the property. If the property does not exist then this will be empty.
raises *VBoxErrorInvalidVmState* Machine session is not open.

get_hw_virt_ex_property(*property_p*)
Returns the value of the specified hardware virtualization boolean property.
in property_p of type *HWVirtExPropertyType* Property type to query.
return value of type bool Property value.
raises *OleErrorInvalidarg* Invalid property.

get_medium(*name*, *controller_port*, *device*)

Returns the virtual medium attached to a device slot of the specified bus.

Note that if the medium was indirectly attached by *mount_medium()* to the given device slot then this method will return not the same object as passed to the *mount_medium()* call. See *IMedium* for more detailed information about mounting a medium.
in name of type str Name of the storage controller the medium is attached to.
in controller_port of type int Port to query.
in device of type int Device slot in the given port to query.
return medium of type *IMedium* Attached medium object.
raises *VBoxErrorObjectNotFound* No medium attached to given slot/bus.

get_medium_attachment(*name*, *controller_port*, *device*)
Returns a medium attachment which corresponds to the controller with the given name, on the given port and device slot.

in name of type str

in controller_port of type int

in device of type int

return attachment of type *IMediumAttachment*
raises *VBoxErrorObjectNotFound* No attachment exists for the given controller/port/device combination.

get_medium_attachments_of_controller(*name*)
Returns an array of medium attachments which are attached to the the controller with the given name.

in name of type str

return medium_attachments of type *IMediumAttachment*
raises *VBoxErrorObjectNotFound* A storage controller with given name doesn't exist.

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

in slot of type int

return adapter of type *INetworkAdapter*
raises *OleErrorInvalidarg* Invalid @a slot number.

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

in slot of type int

return port of type *IParallelPort*
raises *OleErrorInvalidarg* Invalid @a slot number.

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

in slot of type int

return port of type *ISerialPort*
raises ***OleErrorInvalidarg*** Invalid @a slot number.

get_storage_controller_by_instance(*connection_type*, *instance*)
Returns a storage controller of a specific storage bus with the given instance number.

in connection_type of type *StorageBus*

in instance of type int

return storage_controller of type *IStorageController*
raises ***VBoxErrorObjectNotFound*** A storage controller with given instance number doesn't exist.

get_storage_controller_by_name(*name*)
Returns a storage controller with the given name.

in name of type str

return storage_controller of type *IStorageController*
raises ***VBoxErrorObjectNotFound*** A storage controller with given name doesn't exist.

get_usb_controller_by_name(*name*)
Returns a USB controller with the given type.

in name of type str

return controller of type *IUSBController*
raises ***VBoxErrorObjectNotFound*** A USB controller with given name doesn't exist.

get_usb_controller_count_by_type(*type_p*)
Returns the number of USB controllers of the given type attached to the VM.

in type_p of type *USBControllerType*

return controllers of type int

graphics_controller_type
Get or set GraphicsControllerType value for 'graphicsControllerType' Graphics controller type.

groups
Get or set str value for 'groups' Array of machine group names of which this machine is a member. "" and "/" are synonyms for the toplevel group. Each group is only listed once, however they are listed in no particular order and there is no guarantee that there are no gaps in the group hierarchy (i.e. "/group", "/group/subgroup/subsubgroup" is a valid result).

hardware_uuid
Get or set str value for 'hardwareUUID' The UUID presented to the guest via memory tables, hardware and guest properties. For most VMs this is the same as the @a id, but for VMs which have been cloned or teleported it may be the same as the source VM. The latter is because the guest shouldn't notice that it was cloned or teleported.

hardware_version
Get or set str value for 'hardwareVersion' Hardware version identifier. Internal use only for now.

hot_plug_cpu(*cpu*)
Plugs a CPU into the machine.
in cpu of type int The CPU id to insert.

hot_unplug_cpu(*cpu*)
Removes a CPU from the machine.

in cpu of type int The CPU id to remove.

hpet_enabled
Get or set bool value for 'HPETEnabled' This attribute controls if High Precision Event Timer (HPET) is enabled in this VM. Use this property if you want to provide guests with additional time source, or if guest requires HPET to function correctly. Default is false.

icon
Get or set str value for 'icon' Overridden VM Icon details.

id_p
Get str value for 'id' UUID of the virtual machine.

io_cache_enabled
Get or set bool value for 'IOCacheEnabled' When set to @a true, the builtin I/O cache of the virtual machine will be enabled.

io_cache_size
Get or set int value for 'IOCacheSize' Maximum size of the I/O cache in MB.

keyboard_hid_type
Get or set KeyboardHIDType value for 'keyboardHIDType' Type of keyboard HID used in this VM. The default is typically "PS2Keyboard" but can vary depending on the requirements of the guest operating system.

last_state_change
Get int value for 'lastStateChange' Time stamp of the last execution state change, in milliseconds since 1970-01-01 UTC.

launch_vm_process(*session=None*, *type_p='gui'*, *environment=''*)
Spawns a new process that will execute the virtual machine and obtains a shared lock on the machine for the calling session.

If launching the VM succeeds, the new VM process will create its own session and write-lock the machine for it, preventing conflicting changes from other processes. If the machine is already locked (because it is already running or because another session has a write lock), launching the VM process will therefore fail. Reversely, future attempts to obtain a write lock will also fail while the machine is running.

The caller's session object remains separate from the session opened by the new VM process. It receives its own *IConsole* object which can be used to control machine execution, but it cannot be used to change all VM settings which would be available after a *lock_machine()* call.

The caller must eventually release the session's shared lock by calling *ISession.unlock_machine()* on the local session object once this call has returned. However, the session's state (see *ISession.state()*) will not return to "Unlocked" until the remote session has also unlocked the machine (i.e. the machine has stopped running).

Launching a VM process can take some time (a new VM is started in a new process, for which memory and other resources need to be set up). Because of this, an *IProgress* object is returned to allow the caller to wait for this asynchronous operation to be completed. Until then, the caller's session object remains in the "Unlocked" state, and its *ISession.machine()* and *ISession.console()* attributes cannot be accessed. It is recommended to use *IProgress.wait_for_completion()* or similar calls to wait for completion. Completion is signalled when the VM is powered on. If launching the VM fails, error messages can be queried via the progress object, if available.

The progress object will have at least 2 sub-operations. The first operation covers the period up to the new VM process calls powerUp. The subsequent operations mirror the *IConsole.*

power_up() progress object. Because *IConsole.power_up()* may require some extra sub-operations, the *IProgress.operation_count()* may change at the completion of operation.

For details on the teleportation progress operation, see *IConsole.power_up()* .

<!– TODO/r=bird: What about making @a environment into a smart array? Guess this predates our safe array support by a year or so... Dmitry wrote the text here, right? Just rename it to @a environmentChanges and shorten the documentation to say the string are applied onto the server environment putenv style, i.e. "VAR=VALUE" for setting/replacing and "VAR" for unsetting. –> The @a environment argument is a string containing definitions of environment variables in the following format:

```
NAME[=VALUE]

NAME[=VALUE]

...
```

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 or empty, the server environment is inherited by the started process as is.

in session of type ***ISession*** Client session object to which the VM process will be connected (this must be in "Unlocked" state).

in name of type str Front-end to use for the new VM process. The following are currently supported:

"gui": VirtualBox Qt GUI front-end "headless": VBoxHeadless (VRDE Server) front-end "sdl": VirtualBox SDL front-end "emergencystop": reserved value, used for aborting the currently running VM or session owner. In this case the @a session parameter may be @c null (if it is non-null it isn't used in any way), and the @a progress return value will be always @c null. The operation completes immediately. "": use the per-VM default frontend if set, otherwise the global default defined in the system properties. If neither are set, the API will launch a "gui" session, which may fail if there is no windowing environment available. See *IMachine.default_frontend()* and *ISystemProperties.default_frontend()* .

in environment of type str Environment to pass to the VM process.

return progress of type ***IProgress*** Progress object to track the operation completion.

raises ***OleErrorUnexpected*** Virtual machine not registered.

raises ***OleErrorInvalidarg*** Invalid session type @a type.

raises ***VBoxErrorObjectNotFound*** No machine matching @a machineId found.

raises ***VBoxErrorInvalidObjectState*** Session already open or being opened.

raises ***VBoxErrorIprtError*** Launching process for machine failed.

raises ***VBoxErrorVmError*** Failed to assign machine to session.

lock_machine(*session*, *lock_type*)

Locks the machine for the given session to enable the caller to make changes to the machine or start the VM or control VM execution.

There are two ways to lock a machine for such uses:

If you want to make changes to the machine settings, you must obtain an exclusive write lock on the machine by setting @a lockType to @c Write.

This will only succeed if no other process has locked the machine to prevent conflicting changes.

Only after an exclusive write lock has been obtained using this method, one can change all VM settings or execute the VM in the process space of the session object. (Note that the latter is only of interest if you actually want to write a new front-end for virtual machines; but this API gets called internally by the existing front-ends such as VBoxHeadless and the VirtualBox GUI to acquire a write lock on the machine that they are running.)

On success, write-locking the machine for a session creates a second copy of the IMachine object. It is this second object upon which changes can be made; in VirtualBox terminology, the second copy is "mutable". It is only this second, mutable machine object upon which you can call methods that change the machine state. After having called this method, you can obtain this second, mutable machine object using the *ISession.machine()* attribute.

If you only want to check the machine state or control machine execution without actually changing machine settings (e.g. to get access to VM statistics or take a snapshot or save the machine state), then set the @a lockType argument to @c Shared.

If no other session has obtained a lock, you will obtain an exclusive write lock as described above. However, if another session has already obtained such a lock, then a link to that existing session will be established which allows you to control that existing session.

To find out which type of lock was obtained, you can inspect *ISession.type_p()* , which will have been set to either @c WriteLock or @c Shared.

In either case, you can get access to the *IConsole* object which controls VM execution.

Also in all of the above cases, one must always call *ISession.unlock_machine()* to release the lock on the machine, or the machine's state will eventually be set to "Aborted".

To change settings on a machine, the following sequence is typically performed:

Call this method to obtain an exclusive write lock for the current session.

Obtain a mutable IMachine object from *ISession.machine()* .

Change the settings of the machine by invoking IMachine methods.

Call *IMachine.save_settings()* .

Release the write lock by calling *ISession.unlock_machine()* .

in session of type *ISession* Session object for which the machine will be locked.

in lock_type of type *LockType* If set to @c Write, then attempt to acquire an exclusive write lock or fail. If set to @c Shared, then either acquire an exclusive write lock or establish a link to an existing session.

raises *OleErrorUnexpected* Virtual machine not registered.

raises *OleErrorAccessdenied* Process not started by

raises *VBoxErrorInvalidObjectState* Session already open or being opened.

raises *VBoxErrorVmError* Failed to assign machine to session.

log_folder
Get str value for 'logFolder' Full path to the folder that stores a set of rotated log files recorded during machine execution. The most recent log file is named VBox.log, the previous log file is named VBox.log.1 and so on (up to VBox.log.3 in the current version).

medium_attachments
Get IMediumAttachment value for 'mediumAttachments' Array of media attached to this machine.

memory_balloon_size
Get or set int value for 'memoryBalloonSize' Memory balloon size in megabytes.

memory_size
Get or set int value for 'memorySize' System memory size in megabytes.

monitor_count
Get or set int value for 'monitorCount' Number of virtual monitors.

Only effective on Windows XP and later guests with Guest Additions installed.

mount_medium(*name*, *controller_port*, *device*, *medium*, *force*)
Mounts a medium (*IMedium* , identified by the given UUID @a id) to the given storage controller (*IStorageController* , identified by @a name), at the indicated port and device. The device must already exist; see *IMachine.attach_device()* for how to attach a new device.

This method is intended only for managing removable media, where the device is fixed but media is changeable at runtime (such as DVDs and floppies). It cannot be used for fixed media such as hard disks.

The @a controllerPort and @a device parameters specify the device slot and have have the same meaning as with *IMachine.attach_device()* .

The specified device slot can have a medium mounted, which will be unmounted first. Specifying a zero UUID (or an empty string) for @a medium does just an unmount.

See *IMedium* for more detailed information about attaching media.
in name of type str Name of the storage controller to attach the medium to.
in controller_port of type int Port to attach the medium to.
in device of type int Device slot in the given port to attach the medium to.
in medium of type *IMedium* Medium to mount or @c null for an empty drive.
in force of type bool Allows to force unmount/mount of a medium which is locked by the device slot in the given port to attach the medium to.
raises *OleErrorInvalidarg* SATA device, SATA port, IDE port or IDE slot out of range.
raises *VBoxErrorInvalidObjectState* Attempt to attach medium to an unregistered virtual machine.
raises *VBoxErrorInvalidVmState* Invalid machine state.
raises *VBoxErrorObjectInUse* Medium already attached to this or another virtual machine.

name
Get or set str value for '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 *save_settings()* 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 *save_settings()* is called.

If any of the above limitations are hit, *save_settings()* will return an appropriate error message explaining the exact reason and the changes you made to this machine will not be saved.

Starting with VirtualBox 4.0, a ".vbox" extension of the settings file is recommended, but not enforced. (Previous versions always used a generic ".xml" extension.)

non_rotational_device (*name*, *controller_port*, *device*, *non_rotational*)
Sets a flag in the device information which indicates that the medium is not based on rotational technology, i.e. that the access times are more or less independent of the position on the medium. This may or may not be supported by a particular drive, and is silently ignored in the latter case. At the moment only hard disks (which is a misnomer in this context) accept this setting. Changing the setting while the VM is running is forbidden. The device must already exist; see *IMachine.attach_device()* for how to attach a new device.

The @a controllerPort and @a device parameters specify the device slot and have have the same meaning as with *IMachine.attach_device()* .
in name of type str Name of the storage controller.
in controller_port of type int Storage controller port.
in device of type int Device slot in the given port.
in non_rotational of type bool New value for the non-rotational device flag.
raises *OleErrorInvalidarg* SATA device, SATA port, IDE port or IDE slot out of range.
raises *VBoxErrorInvalidObjectState* Attempt to modify an unregistered virtual machine.
raises *VBoxErrorInvalidVmState* Invalid machine state.

os_type_id
Get or set str value for 'OSTypeId' User-defined identifier of the Guest OS type. You may use *IVirtualBox.get_guest_os_type()* to obtain an IGuestOSType object representing details about the given Guest OS type.

This value may differ from the value returned by *IGuest.os_type_id()* if Guest Additions are installed to the guest OS.

page_fusion_enabled
Get or set bool value for 'pageFusionEnabled' This setting determines whether VirtualBox allows page fusion for this machine (64-bit hosts only).

paravirt_debug
Get or set str value for 'paravirtDebug' Debug parameters for the paravirtualized guest interface provider.

paravirt_provider
Get or set ParavirtProvider value for 'paravirtProvider' The paravirtualized guest interface provider.

parent
Get IVirtualBox value for 'parent' Associated parent object.

passthrough_device (*name*, *controller_port*, *device*, *passthrough*)
Sets the passthrough mode of an existing DVD device. Changing the setting while the VM is running is forbidden. The setting is only used if at VM start the device is configured as a host DVD drive, in all other cases it is ignored. The device must already exist; see *IMachine.attach_device()* for how to attach a new device.

The @a controllerPort and @a device parameters specify the device slot and have have the same meaning as with *IMachine.attach_device()* .
in name of type str Name of the storage controller.
in controller_port of type int Storage controller port.
in device of type int Device slot in the given port.
in passthrough of type bool New value for the passthrough setting.
raises *OleErrorInvalidarg* SATA device, SATA port, IDE port or IDE slot out of range.

raises *VBoxErrorInvalidObjectState* Attempt to modify an unregistered virtual machine.

raises *VBoxErrorInvalidVmState* Invalid machine state.

pci_device_assignments

Get IPCIDeviceAttachment value for 'PCIDeviceAssignments' Array of PCI devices assigned to this machine, to get list of all PCI devices attached to the machine use *IConsole.attached_pci_devices()* attribute, as this attribute is intended to list only devices additional to what described in virtual hardware config. Usually, this list keeps host's physical devices assigned to the particular machine.

pointing_hid_type

Get or set PointingHIDType value for 'pointingHIDType' Type of pointing HID (such as mouse or tablet) used in this VM. The default is typically "PS2Mouse" but can vary depending on the requirements of the guest operating system.

query_log_filename(*idx*)

Queries for the VM log file name of an given index. Returns an empty string if a log file with that index doesn't exists.

in idx of type int Which log file name to query. 0=current log file.

return filename of type str On return the full path to the log file or an empty string on error.

query_saved_guest_screen_info(*screen_id*)

Returns the guest dimensions from the saved state.

in screen_id of type int Saved guest screen to query info from.

out origin_x of type int The X position of the guest monitor top left corner.

out origin_y of type int The Y position of the guest monitor top left corner.

out width of type int Guest width at the time of the saved state was taken.

out height of type int Guest height at the time of the saved state was taken.

out enabled of type bool Whether the monitor is enabled in the guest.

query_saved_screenshot_info(*screen_id*)

Returns available formats and size of the screenshot from saved state.

in screen_id of type int Saved guest screen to query info from.

out width of type int Image width.

out height of type int Image height.

return bitmap_formats of type *BitmapFormat* Formats supported by readSavedScreenshotToArray.

read_log(*idx*, *offset*, *size*)

Reads the VM log file. The chunk size is limited, so even if you ask for a big piece there might be less data returned.

in idx of type int Which log file to read. 0=current log file.

in offset of type int Offset in the log file.

in size of type int Chunk size to read in the log file.

return data of type str Data read from the log file. A data size of 0 means end of file if the requested chunk size was not 0. This is the unprocessed file data, i.e. the line ending style depends on the platform of the system the server is running on.

read_saved_screenshot_to_array(*screen_id*, *bitmap_format*)

Screenshot in requested format is retrieved to an array of bytes.

in screen_id of type int Saved guest screen to read from.

in bitmap_format of type *BitmapFormat* The requested format.

out width of type int Image width.

out height of type int Image height.

return data of type str Array with resulting image data.

read_saved_thumbnail_to_array(*screen_id*, *bitmap_format*)
Thumbnail is retrieved to an array of bytes in the requested format.
in screen_id of type int Saved guest screen to read from.
in bitmap_format of type ***BitmapFormat*** The requested format.
out width of type int Bitmap width.
out height of type int Bitmap height.
return data of type str Array with resulting bitmap data.

remove(*delete=True*)
Unregister and optionally delete associated config
Options: delete - remove all elements of this VM from the system
Return the IMedia from unregistered VM

remove_all_cpuid_leaves()
Removes all the virtual CPU cpuid leaves

remove_cpuid_leaf(*id_p*)
Removes the virtual CPU cpuid leaf for the specified index
in id_p of type int CPUID leaf index.
raises ***OleErrorInvalidarg*** Invalid id.

remove_shared_folder(*name*)
Removes the permanent shared folder with the given name previously created by *create_shared_folder()* from the collection of shared folders and stops sharing it.
in name of type str Logical name of the shared folder to remove.
raises ***VBoxErrorInvalidVmState*** Virtual machine is not mutable.
raises ***VBoxErrorObjectNotFound*** Shared folder @a name does not exist.

remove_storage_controller(*name*)
Removes a storage controller from the machine with all devices attached to it.
in name of type str
raises ***VBoxErrorObjectNotFound*** A storage controller with given name doesn't exist.
raises ***VBoxErrorNotSupported*** Medium format does not support storage deletion (only for implicitly
created differencing media, should not happen).

remove_usb_controller(*name*)
Removes a USB controller from the machine.

in name of type str
raises ***VBoxErrorObjectNotFound*** A USB controller with given type doesn't exist.

restore_snapshot(*snapshot=None*)
Starts resetting the machine's current state to the state contained in the given snapshot, asynchronously. All current settings of the machine will be reset and changes stored in differencing media will be lost. See *ISnapshot* for an introduction to snapshots.

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

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

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

If the machine state is *MachineState.saved* Saved prior to this operation, the saved state file will be implicitly deleted (as if *IMachine.discard_saved_state()* were called).
in snapshot of type ***ISnapshot*** The snapshot to restore the VM state from.

return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* Virtual machine is running.

rtc_use_utc
Get or set bool value for 'RTCUseUTC' When set to @a true, the RTC device of the virtual machine will run in UTC time, otherwise in local time. Especially Unix guests prefer the time in UTC.

save_settings()
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 *save_settings()* or *discard_settings()* . For registered machines, new settings become visible to all other VirtualBox clients after successful invocation of this method.

The method sends *IMachineDataChangedEvent* notification event after the configuration has been successfully saved (only for registered machines).

Calling this method is only valid on instances returned by *ISession.machine()* and on new machines created by *IVirtualBox.create_machine()* but not yet registered, or on unregistered machines after calling *IMachine.unregister()* .
raises *VBoxErrorFileError* Settings file not accessible.
raises *VBoxErrorXmlError* Could not parse the settings file.
raises *OleErrorAccessdenied* Modification request refused.

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

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

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

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

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

take_snapshot()
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* Virtual machine state neither Running nor Paused.
raises *VBoxErrorFileError* Failed to create directory for saved state file.

session_name
Get str value for 'sessionName' Name of the session. If *session_state()* is Spawning or Locked, this attribute contains the same value as passed to the *IMachine.launch_vm_process()* method in the @a name parameter. If the session was established with *IMachine.lock_machine()* , it is the name of the session (if set, otherwise empty string). If *session_state()* is SessionClosed, the value of this attribute is an empty string.

session_pid
Get int value for 'sessionPID' Identifier of the session process. This attribute contains the platform-dependent identifier of the process whose session was used with *IMachine.lock_machine()* call. The returned value is only valid if *session_state()* is Locked or Unlocking by the time this property is read.

session_state
Get SessionState value for 'sessionState' Current session state for this machine.

set_auto_discard_for_device(*name*, *controller_port*, *device*, *discard*)
Sets a flag in the device information which indicates that the medium supports discarding unused blocks (called trimming for SATA or unmap for SCSI devices) .This may or may not be supported by a particular drive, and is silently ignored in the latter case. At the moment only hard disks (which is a misnomer in this context) accept this setting. Changing the setting while the VM is running is forbidden. The device must already exist; see *IMachine.attach_device()* for how to attach a new device.

The @a controllerPort and @a device parameters specify the device slot and have have the same meaning as with *IMachine.attach_device()* .
in name of type str Name of the storage controller.
in controller_port of type int Storage controller port.
in device of type int Device slot in the given port.
in discard of type bool New value for the discard device flag.
raises *OleErrorInvalidarg* SATA device, SATA port, SCSI port out of range.
raises *VBoxErrorInvalidObjectState* Attempt to modify an unregistered virtual machine.
raises *VBoxErrorInvalidVmState* Invalid machine state.

set_bandwidth_group_for_device(*name*, *controller_port*, *device*, *bandwidth_group*)
Sets the bandwidth group of an existing storage device. The device must already exist; see *IMachine.attach_device()* for how to attach a new device.

The @a controllerPort and @a device parameters specify the device slot and have have the same meaning as with *IMachine.attach_device()* .
in name of type str Name of the storage controller.
in controller_port of type int Storage controller port.
in device of type int Device slot in the given port.
in bandwidth_group of type *IBandwidthGroup* New value for the bandwidth group or @c null for no group.
raises *OleErrorInvalidarg* SATA device, SATA port, IDE port or IDE slot out of range.
raises *VBoxErrorInvalidObjectState* Attempt to modify an unregistered virtual machine.
raises *VBoxErrorInvalidVmState* Invalid machine state.

set_boot_order(*position*, *device*)
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()
in position of type int Position in the boot order (@c 1 to the total number of devices the machine can boot from, as returned by *ISystemProperties.max_boot_position()*).
in device of type *DeviceType* The type of the device used to boot at the given position.
raises *OleErrorInvalidarg* Boot @a position out of range.
raises *OleErrorNotimpl* Booting from USB @a device currently not supported.

set_cpu_property(*property_p*, *value*)
Sets the virtual CPU boolean value of the specified property.
in property_p of type *CPUPropertyType* Property type to query.
in value of type bool Property value.
raises *OleErrorInvalidarg* Invalid property.

set_cpuid_leaf (*id_p*, *val_eax*, *val_ebx*, *val_ecx*, *val_edx*)
Sets the virtual CPU cpuid information for the specified leaf. Note that these values are not passed unmodified. VirtualBox clears features that it doesn't support.

Currently supported index values for cpuid: Standard CPUID leafs: 0 - 0xA Extended CPUID leafs: 0x80000000 - 0x8000000A

See the Intel and AMD programmer's manuals for detailed information about the cpuid instruction and its leafs.

Do not use this method unless you know exactly what you're doing. Misuse can lead to random crashes inside VMs.
in id_p of type int CPUID leaf index.
in val_eax of type int CPUID leaf value for register eax.
in val_ebx of type int CPUID leaf value for register ebx.
in val_ecx of type int CPUID leaf value for register ecx.
in val_edx of type int CPUID leaf value for register edx.
raises ***OleErrorInvalidarg*** Invalid id.

set_extra_data (*key*, *value*)
Sets associated machine-specific extra data.

If you pass @c null or an empty string as a key @a value, the given @a key will be deleted.

Before performing the actual data change, this method will ask all registered listeners using the *IExtraDataCanChangeEvent* notification for a permission. If one of the listeners refuses the new value, the change will not be performed.

On success, the *IExtraDataChangedEvent* notification is called to inform all registered listeners about a successful data change.

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.
in key of type str Name of the data key to set.
in value of type str Value to assign to the key.
raises ***VBoxErrorFileError*** Settings file not accessible.
raises ***VBoxErrorXmlError*** Could not parse the settings file.

set_guest_property (*property_p*, *value*, *flags*)
Sets, changes or deletes an entry in the machine's guest property store.
in property_p of type str The name of the property to set, change or delete.
in value of type str The new value of the property to set, change or delete. If the property does not yet exist and value is non-empty, it will be created. If the value is @c null or empty, the property will be deleted if it exists.
in flags of type str Additional property parameters, passed as a comma-separated list of "name=value" type entries.
raises ***OleErrorAccessdenied*** Property cannot be changed.
raises ***OleErrorInvalidarg*** Invalid @a flags.
raises ***VBoxErrorInvalidVmState*** Virtual machine is not mutable or session not open.
raises ***VBoxErrorInvalidObjectState*** Cannot set transient property when machine not running.

set_guest_property_value (*property_p*, *value*)
Sets or changes a value in the machine's guest property store. The flags field will be left unchanged or created empty for a new property.
in property_p of type str The name of the property to set or change.
in value of type str The new value of the property to set or change. If the property does not yet exist and value is non-empty, it will be created.
raises ***OleErrorAccessdenied*** Property cannot be changed.

raises *VBoxErrorInvalidVmState* Virtual machine is not mutable or session not open.
raises *VBoxErrorInvalidObjectState* Cannot set transient property when machine not running.

set_hot_pluggable_for_device(*name*, *controller_port*, *device*, *hot_pluggable*)
Sets a flag in the device information which indicates that the attached device is hot pluggable or not. This may or may not be supported by a particular controller and/or drive, and is silently ignored in the latter case. Changing the setting while the VM is running is forbidden. The device must already exist; see *IMachine.attach_device()* for how to attach a new device.

The @a controllerPort and @a device parameters specify the device slot and have have the same meaning as with *IMachine.attach_device()* .
in name of type str Name of the storage controller.
in controller_port of type int Storage controller port.
in device of type int Device slot in the given port.
in hot_pluggable of type bool New value for the hot-pluggable device flag.
raises *OleErrorInvalidarg* SATA device, SATA port, IDE port or IDE slot out of range.
raises *VBoxErrorInvalidObjectState* Attempt to modify an unregistered virtual machine.
raises *VBoxErrorInvalidVmState* Invalid machine state.
raises *VBoxErrorNotSupported* Controller doesn't support hot plugging.

set_hw_virt_ex_property(*property_p*, *value*)
Sets a new value for the specified hardware virtualization boolean property.
in property_p of type *HWVirtExPropertyType* Property type to set.
in value of type bool New property value.
raises *OleErrorInvalidarg* Invalid property.

set_no_bandwidth_group_for_device(*name*, *controller_port*, *device*)
Sets no bandwidth group for an existing storage device. The device must already exist; see *IMachine.attach_device()* for how to attach a new device. The @a controllerPort and @a device parameters specify the device slot and have have the same meaning as with *IMachine.attach_device()* .
in name of type str Name of the storage controller.
in controller_port of type int Storage controller port.
in device of type int Device slot in the given port.
raises *OleErrorInvalidarg* SATA device, SATA port, IDE port or IDE slot out of range.
raises *VBoxErrorInvalidObjectState* Attempt to modify an unregistered virtual machine.
raises *VBoxErrorInvalidVmState* Invalid machine state.

set_settings_file_path(*settings_file_path*)
Currently, it is an error to change this property on any machine. Later this will allow setting a new path for the settings file, with automatic relocation of all files (including snapshots and disk images) which are inside the base directory. This operation is only allowed when there are no pending unsaved settings.

Setting this property to @c null or to an empty string is forbidden. When setting this property, the specified path must be absolute. The specified path may not exist, it will be created when necessary.
in settings_file_path of type str New settings file path, will be used to determine the new location for the attached media if it is in the same directory or below as the original settings file.
return progress of type *IProgress* Progress object to track the operation completion.
raises *OleErrorNotimpl* The operation is not implemented yet.

set_storage_controller_bootable(*name*, *bootable*)

Sets the bootable flag of the storage controller with the given name.

in name of type str

in bootable of type bool
raises *VBoxErrorObjectNotFound* A storage controller with given name doesn't exist.
raises *VBoxErrorObjectInUse* Another storage controller is marked as bootable already.

settings_aux_file_path
Get str value for 'settingsAuxFilePath' Full name of the file containing auxiliary machine settings data.

settings_file_path
Get str value for 'settingsFilePath' Full name of the file containing machine settings data.

settings_modified
Get bool value for 'settingsModified' Whether the settings of this machine have been modified (but neither yet saved nor discarded).

Reading this property is only valid on instances returned by *ISession.machine()* and on new machines created by *IVirtualBox.create_machine()* or opened by *IVirtualBox.open_machine()* but not yet registered, or on unregistered machines after calling *IMachine.unregister()* . For all other cases, the settings can never be modified.

For newly created unregistered machines, the value of this property is always @c true until *save_settings()* is called (no matter if any machine settings have been changed after the creation or not). For opened machines the value is set to @c false (and then follows to normal rules).

shared_folders
Get ISharedFolder value for '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 *create_shared_folder()* . Existing shared folders can be removed using *remove_shared_folder()* .

show_console_window()
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.

This method will fail if a session for this machine is not currently open.
return win_id of type int Platform-dependent identifier of the top-level VM console window, or zero if this method has performed all actions necessary to implement the *show window* semantics for the given platform and/or VirtualBox front-end.
raises *VBoxErrorInvalidVmState* Machine session is not open.

snapshot_count
Get int value for 'snapshotCount' Number of snapshots taken on this machine. Zero means the machine doesn't have any snapshots.

snapshot_folder
Get or set str value for 'snapshotFolder' Full path to the directory used to store snapshot data (differencing media and saved state files) of this machine.

The initial value of this property is <*settings_file_path()* path_to_settings_file>/< *id_p()* 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.

Setting this property to @c null or to an empty string will restore the initial value.

When setting this property, the specified path can be absolute (full path) or relative to the directory where the *settings_file_path()* machine settings file is located. When reading this property, a full path is always returned.

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

state
Get MachineState value for 'state' Current execution state of this machine.

state_file_path
Get str value for 'stateFilePath' Full path to the file that stores the execution state of the machine when it is in the *MachineState.saved* state.

When the machine is not in the Saved state, this attribute is an empty string.

storage_controllers
Get IStorageController value for 'storageControllers' Array of storage controllers attached to this machine.

take_snapshot (*name*, *description*, *pause*)
Saves the current execution state and all settings of the machine and creates differencing images for all normal (non-independent) media. See *ISnapshot* for an introduction to snapshots.

This method can be called for a PoweredOff, Saved (see *save_state()*), Running or Paused virtual machine. When the machine is PoweredOff, an offline snapshot is created. When the machine is Running a live snapshot is created, and an online snapshot is created when Paused.

The taken snapshot is always based on the *current_snapshot()* current snapshot of the associated virtual machine and becomes a new current snapshot.

This method implicitly calls *save_settings()* to save all current machine settings before taking an offline snapshot.
in name of type str Short name for the snapshot.
in description of type str Optional description of the snapshot.
in pause of type bool Whether the VM should be paused while taking the snapshot. Only relevant when the VM is running, and distinguishes between online (@c true) and live (@c false) snapshots. When the VM is not running the result is always an offline snapshot.
out id_p of type str UUID of the snapshot which will be created. Useful for follow-up operations after the snapshot has been created.
return progress of type ***IProgress*** Progress object to track the operation completion.
raises ***VBoxErrorInvalidVmState*** Virtual machine currently changing state.

teleporter_address
Get or set str value for 'teleporterAddress' The address the target teleporter will listen on. If set to an empty string, it will listen on all addresses.

teleporter_enabled
Get or set bool value for 'teleporterEnabled' When set to @a true, the virtual machine becomes a target teleporter the next time it is powered on. This can only set to @a true when the VM is in the @a PoweredOff or @a Aborted state.

<!– This property is automatically set to @a false when the VM is powered on. (bird: This doesn't work yet) –>

teleporter_password
Get or set str value for 'teleporterPassword' The password to check for on the target teleporter. This is just a very basic measure to prevent simple hacks and operators accidentally beaming a virtual machine to the wrong place.

Note that you SET a plain text password while reading back a HASHED password. Setting a hashed password is currently not supported.

teleporter_port
Get or set int value for 'teleporterPort' The TCP port the target teleporter will listen for incoming teleportations on.

0 means the port is automatically selected upon power on. The actual value can be read from this property while the machine is waiting for incoming teleportations.

temporary_eject_device (*name*, *controller_port*, *device*, *temporary_eject*)
Sets the behavior for guest-triggered medium eject. In some situations it is desirable that such ejects update the VM configuration, and in others the eject should keep the VM configuration. The device must already exist; see *IMachine.attach_device()* for how to attach a new device.

The @a controllerPort and @a device parameters specify the device slot and have have the same meaning as with *IMachine.attach_device()* .
in name of type str Name of the storage controller.
in controller_port of type int Storage controller port.
in device of type int Device slot in the given port.
in temporary_eject of type bool New value for the eject behavior.
raises *OleErrorInvalidarg* SATA device, SATA port, IDE port or IDE slot out of range.
raises *VBoxErrorInvalidObjectState* Attempt to modify an unregistered virtual machine.
raises *VBoxErrorInvalidVmState* Invalid machine state.

tracing_config
Get or set str value for 'tracingConfig' Tracepoint configuration to apply at startup when *IMachine.tracing_enabled()* is true. The string specifies a space separated of tracepoint group names to enable. The special group 'all' enables all tracepoints. Check DBGFR3TracingConfig for more details on available tracepoint groups and such.

Note that on hosts supporting DTrace (or similar), a lot of the tracepoints may be implemented exclusively as DTrace probes. So, the effect of the same config may differ between Solaris and Windows for example.

tracing_enabled
Get or set bool value for 'tracingEnabled' Enables the tracing facility in the VMM (including PDM devices + drivers). The VMM will consume about 0.5MB of more memory when enabled and there may be some extra overhead from tracepoints that are always enabled.

unmount_medium (*name*, *controller_port*, *device*, *force*)
Unmounts any currently mounted medium (*IMedium* , identified by the given UUID @a id) to the given storage controller (*IStorageController* , identified by @a name), at the indicated port and device. The device must already exist;

This method is intended only for managing removable media, where the device is fixed but media is changeable at runtime (such as DVDs and floppies). It cannot be used for fixed media such as hard disks.

The @a controllerPort and @a device parameters specify the device slot and have have the same meaning as with *IMachine.attach_device()* .

The specified device slot must have a medium mounted, which will be unmounted. If there is no mounted medium it will do nothing. See *IMedium* for more detailed information about attaching/unmounting media.

in name of type str Name of the storage controller to unmount the medium from.

in controller_port of type int Port to unmount the medium from.

in device of type int Device slot in the given port to unmount the medium from.

in force of type bool Allows to force unmount of a medium which is locked by the device slot in the given port medium is attached to.

raises *OleErrorInvalidarg* SATA device, SATA port, IDE port or IDE slot out of range.

raises *VBoxErrorInvalidObjectState* Attempt to unmount medium that is not removable - not DVD or floppy.

raises *VBoxErrorInvalidVmState* Invalid machine state.

raises *VBoxErrorObjectInUse* Medium already attached to this or another virtual machine.

raises *VBoxErrorObjectNotFound* Medium not attached to specified port, device, controller.

unregister(*cleanup_mode*)

Unregisters a machine previously registered with *IVirtualBox.register_machine()* and optionally do additional cleanup before the machine is unregistered.

This method does not delete any files. It only changes the machine configuration and the list of registered machines in the VirtualBox object. To delete the files which belonged to the machine, including the XML file of the machine itself, call *delete_config()* , optionally with the array of IMedium objects which was returned from this method.

How thoroughly this method cleans up the machine configuration before unregistering the machine depends on the @a cleanupMode argument.

With "UnregisterOnly", the machine will only be unregistered, but no additional cleanup will be performed. The call will fail if the machine is in "Saved" state or has any snapshots or any media attached (see *IMediumAttachment*). It is the responsibility of the caller to delete all such configuration in this mode. In this mode, the API behaves like the former @c IVirtualBox::unregisterMachine() API which it replaces. With "DetachAllReturnNone", the call will succeed even if the machine is in "Saved" state or if it has snapshots or media attached. All media attached to the current machine state or in snapshots will be detached. No medium objects will be returned; all of the machine's media will remain open. With "DetachAllReturnHardDisksOnly", the call will behave like with "DetachAllReturnNone", except that all the hard disk medium objects which were detached from the machine will be returned as an array. This allows for quickly passing them to the *delete_config()* API for closing and deletion. With "Full", the call will behave like with "DetachAllReturnHardDisksOnly", except that all media will be returned in the array, including removable media like DVDs and floppies. This might be useful if the user wants to inspect in detail which media were attached to the machine. Be careful when passing the media array to *delete_config()* in that case because users will typically want to preserve ISO and RAW image files.

A typical implementation will use "DetachAllReturnHardDisksOnly" and then pass the resulting IMedium array to *delete_config()* . This way, the machine is completely deleted with all its saved states and hard disk images, but images for removable drives (such as ISO and RAW files) will remain on disk.

This API does not verify whether the media files returned in the array are still attached to other machines (i.e. shared between several machines). If such a shared image is passed to *delete_config()* however, closing the image will fail there and the image will be silently

skipped.

This API may, however, move media from this machine's media registry to other media registries (see *IMedium* for details on media registries). For machines created with VirtualBox 4.0 or later, if media from this machine's media registry are also attached to another machine (shared attachments), each such medium will be moved to another machine's registry. This is because without this machine's media registry, the other machine cannot find its media any more and would become inaccessible.

This API implicitly calls *save_settings()* to save all current machine settings before unregistering it. It may also silently call *save_settings()* on other machines if media are moved to other machines' media registries.

After successful method invocation, the *IMachineRegisteredEvent* event is fired.

The call will fail if the machine is currently locked (see *ISession*).

If the given machine is inaccessible (see *accessible()*), it will be unregistered and fully uninitialized right afterwards. As a result, the returned machine object will be unusable and an attempt to call **any** method will return the "Object not ready" error.

in cleanup_mode of type ***CleanupMode*** How to clean up after the machine has been unregistered.

return media of type ***IMedium*** List of media detached from the machine, depending on the @a cleanupMode parameter.

raises ***VBoxErrorInvalidObjectState*** Machine is currently locked for a session.

usb_controllers

Get IUSBController value for 'USBControllers' Array of USB controllers attached to this machine.

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

usb_device_filters

Get IUSBDeviceFilters value for 'USBDeviceFilters' Associated USB device filters object.

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

usb_proxy_available

Get bool value for 'USBProxyAvailable' Returns whether there is an USB proxy available.

video_capture_enabled

Get or set bool value for 'videoCaptureEnabled' This setting determines whether VirtualBox uses video recording to record VM session.

video_capture_file

Get or set str value for 'videoCaptureFile' This setting determines the filename VirtualBox uses to save the recorded content. This setting cannot be changed while video capturing is enabled.

When setting this attribute, the specified path has to be absolute (full path). When reading this attribute, a full path is always returned.

video_capture_fps

Get or set int value for 'videoCaptureFPS' This setting determines the maximum number of frames per second. Frames with a higher frequency will be skipped. Reducing this value increases the number of skipped frames and reduces the file size. This setting cannot be changed while video capturing is enabled.

video_capture_height

Get or set int value for 'videoCaptureHeight' This setting determines the vertical resolution of

the recorded video. This setting cannot be changed while video capturing is enabled.

video_capture_max_file_size
Get or set int value for 'videoCaptureMaxFileSize' This setting determines the maximal number of captured video file size in MB. The capture stops as the captured video file size has reached the defined. If this value is zero the capturing will not be limited by file size. This setting cannot be changed while video capturing is enabled.

video_capture_max_time
Get or set int value for 'videoCaptureMaxTime' This setting determines the maximum amount of time in milliseconds the video capture will work for. The capture stops as the defined time interval has elapsed. If this value is zero the capturing will not be limited by time. This setting cannot be changed while video capturing is enabled.

video_capture_options
Get or set str value for 'videoCaptureOptions' This setting contains any additional video capture options required in comma-separated key=value format. This setting cannot be changed while video capturing is enabled.

video_capture_rate
Get or set int value for 'videoCaptureRate' This setting determines the bitrate in kilobits per second. Increasing this value makes the video look better for the cost of an increased file size. This setting cannot be changed while video capturing is enabled.

video_capture_screens
Get or set bool value for 'videoCaptureScreens' This setting determines for which screens video recording is enabled.

video_capture_width
Get or set int value for 'videoCaptureWidth' This setting determines the horizontal resolution of the recorded video. This setting cannot be changed while video capturing is enabled.

vm_process_priority
Get or set str value for 'VMProcessPriority' Sets the priority of the VM process. It is a VM setting which can be changed both before starting the VM and at runtime. The valid values are system specific, and if a value is specified which does not get recognized, then it will be remembered (useful for preparing VM configs for other host OSes), with a successful result.

The default value is the empty string, which selects the default process priority.

vram_size
Get or set int value for 'VRAMSize' Video memory size in megabytes.

vrde_server
Get IVRDEServer value for 'VRDEServer' VirtualBox Remote Desktop Extension (VRDE) server object.

class virtualbox.library.**IProgress**(*interface=None*)
The IProgress interface is used to track and control asynchronous tasks within VirtualBox.

An instance of this is returned every time VirtualBox starts an asynchronous task (in other words, a separate thread) which continues to run after a method call returns. For example, *IMachine.save_state()* , which saves the state of a running virtual machine, can take a long time to complete. To be able to display a progress bar, a user interface such as the VirtualBox graphical user interface can use the IProgress object returned by that method.

Note that IProgress is a "read-only" interface in the sense that only the VirtualBox internals behind the Main API can create and manipulate progress objects, whereas client code can only use the IProgress object to monitor a task's progress and, if *cancelable()* is @c true, cancel the task by calling *cancel()* .

A task represented by IProgress consists of either one or several sub-operations that run sequentially, one by one (see *operation()* and *operation_count()*). Every operation is identified by a number (starting from 0) and has a separate description.

You can find the individual percentage of completion of the current operation in *operation_percent()* and the percentage of completion of the task as a whole in *percent()* .

Similarly, you can wait for the completion of a particular operation via *wait_for_operation_completion()* or for the completion of the whole task via *wait_for_completion()* .

wait_for_completion(*timeout=-1*)
Waits until the task is done (including all sub-operations) with a given timeout in milliseconds; specify -1 for an indefinite wait.

Note that the VirtualBox/XPCOM/COM/native event queues of the calling thread are not processed while waiting. Neglecting event queues may have dire consequences (degrade performance, resource hogs, deadlocks, etc.), this is specially so for the main thread on platforms using XPCOM. Callers are advised wait for short periods and service their event queues between calls, or to create a worker thread to do the waiting.
in timeout of type int Maximum time in milliseconds to wait or -1 to wait indefinitely.
raises *VBoxErrorIprtError* Failed to wait for task completion.

cancel()
Cancels the task.

If *cancelable()* is @c false, then this method will fail.
raises *VBoxErrorInvalidObjectState* Operation cannot be canceled.

cancelable
Get bool value for 'cancelable' Whether the task can be interrupted.

canceled
Get bool value for 'canceled' Whether the task has been canceled.

completed
Get bool value for 'completed' Whether the task has been completed.

description
Get str value for 'description' Description of the task.

error_info
Get IVirtualBoxErrorInfo value for 'errorInfo' Extended information about the unsuccessful result of the progress operation. May be @c null if no extended information is available. Valid only if *completed()* is @c true and *result_code()* indicates a failure.

id_p
Get str value for 'id' ID of the task.

initiator
Get Interface value for 'initiator' Initiator of the task.

operation
Get int value for 'operation' Number of the sub-operation being currently executed.

operation_count
Get int value for 'operationCount' Number of sub-operations this task is divided into. Every task consists of at least one suboperation.

operation_description
Get str value for 'operationDescription' Description of the sub-operation being currently executed.

operation_percent
Get int value for 'operationPercent' Progress value of the current sub-operation only, in percent.

operation_weight
Get int value for 'operationWeight' Weight value of the current sub-operation only.

percent
Get int value for 'percent' Current progress value of the task as a whole, in percent. This value depends on how many operations are already complete. Returns 100 if *completed()* is @c true.

result_code
Get int value for 'resultCode' Result code of the progress task. Valid only if *completed()* is @c true.

set_current_operation_progress(*percent*)
Internal method, not to be called externally.

in percent of type int

set_next_operation(*next_operation_description*, *next_operations_weight*)
Internal method, not to be called externally.

in next_operation_description of type str

in next_operations_weight of type int

time_remaining
Get int value for 'timeRemaining' Estimated remaining time until the task completes, in seconds. Returns 0 once the task has completed; returns -1 if the remaining time cannot be computed, in particular if the current progress is 0.

Even if a value is returned, the estimate will be unreliable for low progress values. It will become more reliable as the task progresses; it is not recommended to display an ETA before at least 20% of a task have completed.

timeout
Get or set int value for 'timeout' When non-zero, this specifies the number of milliseconds after which the operation will automatically be canceled. This can only be set on cancelable objects.

wait_for_async_progress_completion(*p_progress_async*)
Waits until the other task is completed (including all sub-operations) and forward all changes from the other progress to this progress. This means sub-operation number, description, percent and so on.

You have to take care on setting up at least the same count on sub-operations in this progress object like there are in the other progress object.

If the other progress object supports cancel and this object gets any cancel request (when here enabled as well), it will be forwarded to the other progress object.

If there is an error in the other progress, this error isn't automatically transfered to this progress object. So you have to check any operation error within the other progress object, after this method returns.
in p_progress_async of type *IProgress* The progress object of the asynchrony process.

wait_for_operation_completion(*operation*, *timeout*)
Waits until the given operation is done with a given timeout in milliseconds; specify -1 for an indefinite wait.

See *wait_for_completion()* for event queue considerations.
in operation of type int Number of the operation to wait for. Must be less than *operation_count()*.
in timeout of type int Maximum time in milliseconds to wait or -1 to wait indefinitely.
raises *VBoxErrorIprtError* Failed to wait for operation completion.

class virtualbox.library.**IConsole**(*interface=None*)
The IConsole interface represents an interface to control virtual machine execution.

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

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

ISession

register_on_network_adapter_changed(*callback*)
Set the callback function to consume on network adapter changed events.

Callback receives a INetworkAdapterChangedEvent object.

Returns the callback_id

register_on_serial_port_changed(*callback*)
Set the callback function to consume on serial port changed events.

Callback receives a ISerialPortChangedEvent object.

Returns the callback_id

register_on_parallel_port_changed(*callback*)
Set the callback function to consume on serial port changed events.

Callback receives a IParallelPortChangedEvent object.

Returns the callback_id

register_on_medium_changed(*callback*)
Set the callback function to consume on medium changed events.

Callback receives a IMediumChangedEvent object.

Returns the callback_id

register_on_clipboard_mode_changed(*callback*)
Set the callback function to consume on clipboard mode changed events.

Callback receives a IClipboardModeChangedEvent object.

Returns the callback_id

register_on_drag_and_drop_mode_changed(*callback*)
Set the callback function to consume on drag and drop mode changed events.

Callback receives a IDragAndDropModeChangedEvent object.

Returns the callback_id

register_on_vrde_server_changed(*callback*)
Set the callback function to consume on VirtualBox Remote Desktop Extension (VRDE) changed events.

Callback receives a IVRDEServerChangedEvent object.

Returns the callback_id

register_on_shared_folder_changed(*callback*)
Set the callback function to consume on shared folder changed events.

Callback receives a ISharedFolderChangedEvent object.

Returns the callback_id

register_on_additions_state_changed(*callback*)
Set the callback function to consume on additions state changed events.

Callback receives a IAdditionsStateChangedEvent object.
Note: Interested callees should query IGuest attributes to find out what has changed.
Returns the callback_id

register_on_state_changed(*callback*)
Set the callback function to consume on state changed events which are generated when the state of the machine changes.

Callback receives a IStateChangeEvent object.

Returns the callback_id

register_on_event_source_changed(*callback*)
Set the callback function to consume on event source changed events. This occurs when a listener is added or removed.

Callback receives a IEventStateChangedEvent object.

Returns the callback_id

register_on_can_show_window(*callback*)
Set the callback function to consume on can show window events. This occurs when the console window is to be activated and brought to the foreground of the desktop of the host PC. If this behaviour is not desired a call to event.add_veto will stop this from happening.

Callback receives a ICanShowWindowEvent object.

Returns the callback_id

register_on_show_window(*callback*)
Set the callback function to consume on show window events. This occurs when the console window is to be activated and brought to the foreground of the desktop of the host PC.

Callback receives a IShowWindowEvent object.

Returns the callback_id

add_disk_encryption_password(*id_p*, *password*, *clear_on_suspend*)
Adds a password used for hard disk encryption/decryption.
in id_p of type str The identifier used for the password. Must match the identifier used when the encrypted medium was created.
in password of type str The password.
in clear_on_suspend of type bool Flag whether to clear the password on VM suspend (due to a suspending host for example). The password must be supplied again before the VM can resume.

raises *VBoxErrorPasswordIncorrect* The password provided wasn't correct for at least one disk using the provided ID.

add_disk_encryption_passwords(*ids*, *passwords*, *clear_on_suspend*)
Adds a password used for hard disk encryption/decryption.
in ids of type str List of identifiers for the passwords. Must match the identifier used when the encrypted medium was created.
in passwords of type str List of passwords.
in clear_on_suspend of type bool Flag whether to clear the given passwords on VM suspend (due to a suspending host for example). The passwords must be supplied again before the VM can resume.
raises *VBoxErrorPasswordIncorrect* The password provided wasn't correct for at least one disk using the provided ID.

attach_usb_device(*id_p*, *capture_filename*)
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: *USBDeviceState.busy* , *USBDeviceState.available* or *USBDeviceState.held* , otherwise an error is immediately returned.

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

IUSBDeviceFilters.device_filters() , *USBDeviceState*
in id_p of type str UUID of the host USB device to attach.
in capture_filename of type str Filename to capture the USB traffic to.
raises *VBoxErrorInvalidVmState* Virtual machine state neither Running nor Paused.
raises *VBoxErrorPdmError* Virtual machine does not have a USB controller.

attached_pci_devices
Get IPCIDeviceAttachment value for 'attachedPCIDevices' Array of PCI devices attached to this machine.

clear_all_disk_encryption_passwords()
Clears all provided supplied disk encryption passwords.

create_shared_folder(*name*, *host_path*, *writable*, *automount*)
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.
in name of type str Unique logical name of the shared folder.
in host_path of type str Full path to the shared folder in the host file system.
in writable of type bool Whether the share is writable or readonly
in automount of type bool Whether the share gets automatically mounted by the guest or not.
raises *VBoxErrorInvalidVmState* Virtual machine in Saved state or currently changing state.
raises *VBoxErrorFileError* Shared folder already exists or not accessible.

debugger
Get IMachineDebugger value for 'debugger' Debugging interface.

detach_usb_device(*id_p*)
Detaches an USB device with the given UUID from the USB controller of the virtual machine.

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

possible automatic re-attachment.

IUSBDeviceFilters.device_filters(), *USBDeviceState*
in id_p of type str UUID of the USB device to detach.
return device of type *IUSBDevice* Detached USB device.
raises *VBoxErrorPdmError* Virtual machine does not have a USB controller.
raises *OleErrorInvalidarg* USB device not attached to this virtual machine.

display
Get IDisplay value for 'display' Virtual display object.

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

emulated_usb
Get IEmulatedUSB value for 'emulatedUSB' Interface that manages emulated USB devices.

event_source
Get IEventSource value for 'eventSource' Event source for console events.

find_usb_device_by_address(*name*)
Searches for a USB device with the given host address.

IUSBDevice.address()
in name of type str Address of the USB device (as assigned by the host) to search for.
return device of type *IUSBDevice* Found USB device object.
raises *VBoxErrorObjectNotFound* Given @c name does not correspond to any USB device.

find_usb_device_by_id(*id_p*)
Searches for a USB device with the given UUID.

IUSBDevice.id_p()
in id_p of type str UUID of the USB device to search for.
return device of type *IUSBDevice* Found USB device object.
raises *VBoxErrorObjectNotFound* Given @c id does not correspond to any USB device.

get_device_activity(*type_p*)
Gets the current activity type of given devices or device groups.

in type_p of type *DeviceType*

return activity of type *DeviceActivity*
raises *OleErrorInvalidarg* Invalid device type.

get_guest_entered_acpi_mode()
Checks if the guest entered the ACPI mode G0 (working) or G1 (sleeping). If this method returns @c false, the guest will most likely not respond to external ACPI events.

return entered of type bool
raises *VBoxErrorInvalidVmState* Virtual machine not in Running state.

get_power_button_handled()
Checks if the last power button event was handled by guest.

return handled of type bool
raises *VBoxErrorPdmError* Checking if the event was handled by the guest OS failed.

guest
Get IGuest value for 'guest' Guest object.

keyboard
Get IKeyboard value for 'keyboard' Virtual keyboard object.

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

machine
Get IMachine value for 'machine' Machine object for this console session.

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

mouse
Get IMouse value for 'mouse' Virtual mouse object.

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

pause()
Pauses the virtual machine execution.
raises ***VBoxErrorInvalidVmState*** Virtual machine not in Running state.
raises ***VBoxErrorVmError*** Virtual machine error in suspend operation.

power_button()
Sends the ACPI power button event to the guest.
raises ***VBoxErrorInvalidVmState*** Virtual machine not in Running state.
raises ***VBoxErrorPdmError*** Controlled power off failed.

power_down()
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.
return progress of type ***IProgress*** Progress object to track the operation completion.
raises ***VBoxErrorInvalidVmState*** Virtual machine must be Running, Paused or Stuck to be powered down.

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

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

If the machine is powered off or aborted, the execution will start from the beginning (as if the real hardware were just powered on).

If the machine is in the *MachineState.saved* state, it will continue its execution the point where the state has been saved.

If the machine *IMachine.teleporter_enabled()* property is enabled on the machine being powered up, the machine will wait for an incoming teleportation in the *MachineState.teleporting_in* state. The returned progress object will have at least three operations where the last three are defined as: (1) powering up and starting TCP server, (2) waiting for incoming teleportations, and (3) perform teleportation. These operations will be reflected as the last three operations of the progress objected returned by *IMachine.launch_vm_process()* as well.

IMachine.save_state()
return progress of type ***IProgress*** Progress object to track the operation completion.
raises ***VBoxErrorInvalidVmState*** Virtual machine already running.
raises ***VBoxErrorHostError*** Host interface does not exist or name not set.

raises *VBoxErrorFileError* Invalid saved state file.

power_up_paused()
Identical to powerUp except that the VM will enter the *MachineState.paused* state, instead of *MachineState.running* .

power_up()
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* Virtual machine already running.
raises *VBoxErrorHostError* Host interface does not exist or name not set.
raises *VBoxErrorFileError* Invalid saved state file.

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

remove_disk_encryption_password(*id_p*)
Removes a password used for hard disk encryption/decryption from the running VM. As soon as the medium requiring this password is accessed the VM is paused with an error and the password must be provided again.
in id_p of type str The identifier used for the password. Must match the identifier used when the encrypted medium was created.

remove_shared_folder(*name*)
Removes a transient shared folder with the given name previously created by *create_shared_folder()* from the collection of shared folders and stops sharing it.
in name of type str Logical name of the shared folder to remove.
raises *VBoxErrorInvalidVmState* Virtual machine in Saved state or currently changing state.
raises *VBoxErrorFileError* Shared folder does not exists.

reset()
Resets the virtual machine.
raises *VBoxErrorInvalidVmState* Virtual machine not in Running state.
raises *VBoxErrorVmError* Virtual machine error in reset operation.

resume()
Resumes the virtual machine execution.
raises *VBoxErrorInvalidVmState* Virtual machine not in Paused state.
raises *VBoxErrorVmError* Virtual machine error in resume operation.

shared_folders
Get ISharedFolder value for '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.shared_folders()* 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 *create_shared_folder()* . Existing shared folders can be removed using *remove_shared_folder()* .

sleep_button()
Sends the ACPI sleep button event to the guest.
raises *VBoxErrorInvalidVmState* Virtual machine not in Running state.
raises *VBoxErrorPdmError* Sending sleep button event failed.

state
Get MachineState value for 'state' Current execution state of the machine.

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

teleport(*hostname*, *tcpport*, *password*, *max_downtime*)
Teleport the VM to a different host machine or process.

@todo Explain the details.
in hostname of type str The name or IP of the host to teleport to.
in tcpport of type int The TCP port to connect to (1..65535).
in password of type str The password.
in max_downtime of type int The maximum allowed downtime given as milliseconds. 0 is not a valid value. Recommended value: 250 ms.

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

The current implementation treats this a guideline, not as an absolute rule.
return progress of type ***IProgress*** Progress object to track the operation completion.
raises ***VBoxErrorInvalidVmState*** Virtual machine not running or paused.

usb_devices
Get IUSBDevice value for 'USBDevices' Collection of USB devices currently attached to the virtual USB controller.

The collection is empty if the machine is not running.

use_host_clipboard
Get or set bool value for 'useHostClipboard' Whether the guest clipboard should be connected to the host one or whether it should only be allowed access to the VRDE clipboard. This setting may not affect existing guest clipboard connections which are already connected to the host clipboard.

vrde_server_info
Get IVRDEServerInfo value for 'VRDEServerInfo' Interface that provides information on Remote Desktop Extension (VRDE) connection.

class virtualbox.library.**IEventSource**(*interface=None*)
Event source. Generally, any object which could generate events can be an event source, or aggregate one. To simplify using one-way protocols such as webservices running on top of HTTP(S), an event source can work with listeners in either active or passive mode. In active mode it is up to the IEventSource implementation to call *IEventListener.handle_event()* , in passive mode the event source keeps track of pending events for each listener and returns available events on demand.

See *IEvent* for an introduction to VirtualBox event handling.

register_callback(*callback*, *event_type*)
register a callback function for the provided given event_type

create_aggregator(*subordinates*)
Creates an aggregator event source, collecting events from multiple sources. This way a single listener can listen for events coming from multiple sources, using a single blocking *get_event()* on the returned aggregator.
in subordinates of type ***IEventSource*** Subordinate event source this one aggregates.
return result of type ***IEventSource*** Event source aggregating passed sources.

create_listener()
Creates a new listener object, useful for passive mode.

return listener of type *IEventListener*

event_processed(*listener*, *event*)
Must be called for waitable events after a particular listener finished its event processing. When all listeners of a particular event have called this method, the system will then call *IEvent.set_processed()* .
in listener of type *IEventListener* Which listener processed event.
in event of type *IEvent* Which event.

fire_event(*event*, *timeout*)
Fire an event for this source.
in event of type *IEvent* Event to deliver.
in timeout of type int Maximum time to wait for event processing (if event is waitable), in ms; 0 = no wait, -1 = indefinite wait.
return result of type bool true if an event was delivered to all targets, or is non-waitable.

get_event(*listener*, *timeout*)
Get events from this peer's event queue (for passive mode). Calling this method regularly is required for passive event listeners to avoid system overload; see *IEventSource.register_listener()* for details.
in listener of type *IEventListener* Which listener to get data for.
in timeout of type int Maximum time to wait for events, in ms; 0 = no wait, -1 = indefinite wait.
return event of type *IEvent* Event retrieved, or null if none available.
raises *VBoxErrorObjectNotFound* Listener is not registered, or autounregistered.

register_listener(*listener*, *interesting*, *active*)
Register an event listener.

To avoid system overload, the VirtualBox server process checks if passive event listeners call *IEventSource.get_event()* frequently enough. In the current implementation, if more than 500 pending events are detected for a passive event listener, it is forcefully unregistered by the system, and further *get_event()* calls will return @c VBOX_E_OBJECT_NOT_FOUND.
in listener of type *IEventListener* Listener to register.
in interesting of type *VBoxEventType* Event types listener is interested in. One can use wildcards like - *VBoxEventType.any_p* to specify wildcards, matching more than one event.
in active of type bool Which mode this listener is operating in. In active mode, *IEventListener.handle_event()* is called directly. In passive mode, an internal event queue is created for this this IEventListener. For each event coming in, it is added to queues for all interested registered passive listeners. It is then up to the external code to call the listener's *IEventListener.handle_event()* method. When done with an event, the external code must call *event_processed()* .

unregister_listener(*listener*)
Unregister an event listener. If listener is passive, and some waitable events are still in queue they are marked as processed automatically.
in listener of type *IEventListener* Listener to unregister.

class virtualbox.library.**IMouse**(*interface=None*)
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.

register_on_guest_mouse(*callback*)
Set the callback function to consume on guest mouse events.

Callback receives a IGuestMouseEvent object.
Example:

def callback(event): print((“%s %s %s” % (event.x, event.y, event.z))

absolute_supported
Get bool value for ‘absoluteSupported’ Whether the guest OS supports absolute mouse pointer positioning or not.

You can use the *IMouseCapabilityChangedEvent* event to be instantly informed about changes of this attribute during virtual machine execution.

put_mouse_event_absolute()

event_source
Get IEventSource value for ‘eventSource’ Event source for mouse events.

multi_touch_supported
Get bool value for ‘multiTouchSupported’ Whether the guest OS has enabled the multi-touch reporting device.

You can use the *IMouseCapabilityChangedEvent* event to be instantly informed about changes of this attribute during virtual machine execution.

put_mouse_event()

needs_host_cursor
Get bool value for ‘needsHostCursor’ Whether the guest OS can currently switch to drawing it’s own mouse cursor on demand.

You can use the *IMouseCapabilityChangedEvent* event to be instantly informed about changes of this attribute during virtual machine execution.

put_mouse_event()

pointer_shape
Get IMousePointerShape value for ‘pointerShape’ The current mouse pointer used by the guest.

put_event_multi_touch(*count*, *contacts*, *scan_time*)
Sends a multi-touch pointer event. The coordinates are expressed in pixels and start from [1,1] which corresponds to the top left corner of the virtual display.

The guest may not understand or may choose to ignore this event.

multi_touch_supported()
in count of type int Number of contacts in the event.
in contacts of type int Each array element contains packed information about one contact. Bits 0..15: X coordinate in pixels. Bits 16..31: Y coordinate in pixels. Bits 32..39: contact identifier. Bit 40: “in contact” flag, which indicates that there is a contact with the touch surface. Bit 41: “in range” flag, the contact is close enough to the touch surface. All other bits are reserved for future use and must be set to 0.
in scan_time of type int Timestamp of the event in milliseconds. Only relative time between events is important.
raises *OleErrorAccessdenied* Console not powered up.
raises *VBoxErrorIprtError* Could not send event to virtual device.

put_event_multi_touch_string(*count*, *contacts*, *scan_time*)
put_event_multi_touch()
in count of type int *put_event_multi_touch()*
in contacts of type str Contains information about all contacts: “id1,x1,y1,inContact1,inRange1;...;idN,xN,yN,inContactN,inRangeN”. For example for two contacts: “0,10,20,1,1;1,30,40,1,1”
in scan_time of type int *put_event_multi_touch()*

put_mouse_event(*dx*, *dy*, *dz*, *dw*, *button_state*)
Initiates a mouse event using relative pointer movements along x and y axis.
in dx of type int Amount of pixels the mouse should move to the right. Negative values move the mouse to the left.
in dy of type int Amount of pixels the mouse should move downwards. Negative values move the mouse upwards.
in dz of type int Amount of mouse wheel moves. Positive values describe clockwise wheel rotations, negative values describe counterclockwise rotations.
in dw of type int Amount of horizontal mouse wheel moves. Positive values describe a movement to the left, negative values describe a movement to the right.
in button_state of type int The current state of mouse buttons. Every bit represents a mouse button as follows:

Bit 0 (0x01)left mouse button Bit 1 (0x02)right mouse button Bit 2 (0x04)middle mouse button

A value of 1 means the corresponding button is pressed. otherwise it is released.
raises ***OleErrorAccessdenied*** Console not powered up.
raises ***VBoxErrorIprtError*** Could not send mouse event to virtual mouse.

put_mouse_event_absolute(*x*, *y*, *dz*, *dw*, *button_state*)
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.

This method will have effect only if absolute mouse positioning is supported by the guest OS.

absolute_supported()
in x of type int X coordinate of the pointer in pixels, starting from @c 1.
in y of type int Y coordinate of the pointer in pixels, starting from @c 1.
in dz of type int Amount of mouse wheel moves. Positive values describe clockwise wheel rotations, negative values describe counterclockwise rotations.
in dw of type int Amount of horizontal mouse wheel moves. Positive values describe a movement to the left, negative values describe a movement to the right.
in button_state of type int The current state of mouse buttons. Every bit represents a mouse button as follows:

Bit 0 (0x01)left mouse button Bit 1 (0x02)right mouse button Bit 2 (0x04)middle mouse button

A value of @c 1 means the corresponding button is pressed. otherwise it is released.
raises ***OleErrorAccessdenied*** Console not powered up.
raises ***VBoxErrorIprtError*** Could not send mouse event to virtual mouse.

relative_supported
Get bool value for 'relativeSupported' Whether the guest OS supports relative mouse pointer positioning or not.

You can use the *IMouseCapabilityChangedEvent* event to be instantly informed about changes of this attribute during virtual machine execution.

put_mouse_event()

class virtualbox.library.**IProcess**(*interface=None*)
Abstract parent interface for processes handled by VirtualBox.

wait_for(*wait_for*, *timeout_ms=0*)
Abstract parent interface for processes handled by VirtualBox.

arguments
Get str value for 'arguments' The arguments this process is using for execution.

environment
Get str value for 'environment' The initial process environment. Not yet implemented.

event_source
Get IEventSource value for 'eventSource' Event source for process events.

executable_path
Get str value for 'executablePath' Full path of the actual executable image.

exit_code
Get int value for 'exitCode' The exit code. Only available when the process has been terminated normally.

name
Get str value for 'name' The friendly name of this process.

pid
Get int value for 'PID' The process ID (PID).

read(*handle*, *to_read*, *timeout_ms*)
Reads data from a running process.
in handle of type int Handle to read from. Usually 0 is stdin.
in to_read of type int Number of bytes to read.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return data of type str Array of data read.

status
Get ProcessStatus value for 'status' The current process status; see *ProcessStatus* for more information.

terminate()
Terminates (kills) a running process. It can take up to 30 seconds to get a guest process killed. In case a guest process could not be killed an appropriate error is returned.

wait_for_array(*wait_for*, *timeout_ms*)
Waits for one or more events to happen. Scriptable version of *wait_for()* .
in wait_for of type *ProcessWaitForFlag* Specifies what to wait for; see *ProcessWaitForFlag* for more information.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return reason of type *ProcessWaitResult* The overall wait result; see *ProcessWaitResult* for more information.

write(*handle*, *flags*, *data*, *timeout_ms*)
Writes data to a running process.
in handle of type int Handle to write to. Usually 0 is stdin, 1 is stdout and 2 is stderr.
in flags of type int A combination of *ProcessInputFlag* flags.
in data of type str Array of bytes to write. The size of the array also specifies how much to write.
in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.
return written of type int How much bytes were written.

write_array(*handle*, *flags*, *data*, *timeout_ms*)
Writes data to a running process. Scriptable version of *write()* .
in handle of type int Handle to write to. Usually 0 is stdin, 1 is stdout and 2 is stderr.
in flags of type *ProcessInputFlag* A combination of *ProcessInputFlag* flags.

in data of type str Array of bytes to write. The size of the array also specifies how much to write.

in timeout_ms of type int Timeout (in ms) to wait for the operation to complete. Pass 0 for an infinite timeout.

return written of type int How much bytes were written.

class virtualbox.library.**IConsole**(*interface=None*)

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

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

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

ISession

register_on_network_adapter_changed(*callback*)

Set the callback function to consume on network adapter changed events.

Callback receives a INetworkAdapterChangedEvent object.

Returns the callback_id

register_on_serial_port_changed(*callback*)

Set the callback function to consume on serial port changed events.

Callback receives a ISerialPortChangedEvent object.

Returns the callback_id

register_on_parallel_port_changed(*callback*)

Set the callback function to consume on serial port changed events.

Callback receives a IParallelPortChangedEvent object.

Returns the callback_id

register_on_medium_changed(*callback*)

Set the callback function to consume on medium changed events.

Callback receives a IMediumChangedEvent object.

Returns the callback_id

register_on_clipboard_mode_changed(*callback*)

Set the callback function to consume on clipboard mode changed events.

Callback receives a IClipboardModeChangedEvent object.

Returns the callback_id

register_on_drag_and_drop_mode_changed(*callback*)

Set the callback function to consume on drag and drop mode changed events.

Callback receives a IDragAndDropModeChangedEvent object.

Returns the callback_id

register_on_vrde_server_changed(*callback*)

Set the callback function to consume on VirtualBox Remote Desktop Extension (VRDE) changed events.

Callback receives a IVRDEServerChangedEvent object.

Returns the callback_id

register_on_shared_folder_changed(*callback*)
Set the callback function to consume on shared folder changed events.

Callback receives a ISharedFolderChangedEvent object.

Returns the callback_id

register_on_additions_state_changed(*callback*)
Set the callback function to consume on additions state changed events.

Callback receives a IAdditionsStateChangedEvent object.
Note: Interested callees should query IGuest attributes to find out what has changed.
Returns the callback_id

register_on_state_changed(*callback*)
Set the callback function to consume on state changed events which are generated when the state of the machine changes.

Callback receives a IStateChangeEvent object.

Returns the callback_id

register_on_event_source_changed(*callback*)
Set the callback function to consume on event source changed events. This occurs when a listener is added or removed.

Callback receives a IEventStateChangedEvent object.

Returns the callback_id

register_on_can_show_window(*callback*)
Set the callback function to consume on can show window events. This occurs when the console window is to be activated and brought to the foreground of the desktop of the host PC. If this behaviour is not desired a call to event.add_veto will stop this from happening.

Callback receives a ICanShowWindowEvent object.

Returns the callback_id

register_on_show_window(*callback*)
Set the callback function to consume on show window events. This occurs when the console window is to be activated and brought to the foreground of the desktop of the host PC.

Callback receives a IShowWindowEvent object.

Returns the callback_id

add_disk_encryption_password(*id_p*, *password*, *clear_on_suspend*)
Adds a password used for hard disk encryption/decryption.
in id_p of type str The identifier used for the password. Must match the identifier used when the encrypted medium was created.
in password of type str The password.
in clear_on_suspend of type bool Flag whether to clear the password on VM suspend (due to a suspending host for example). The password must be supplied again before the VM can resume.
raises *VBoxErrorPasswordIncorrect* The password provided wasn't correct for at least one disk using the provided
ID.

add_disk_encryption_passwords(*ids*, *passwords*, *clear_on_suspend*)
Adds a password used for hard disk encryption/decryption.

in ids of type str List of identifiers for the passwords. Must match the identifier used when the encrypted medium was created.
in passwords of type str List of passwords.
in clear_on_suspend of type bool Flag whether to clear the given passwords on VM suspend (due to a suspending host for example). The passwords must be supplied again before the VM can resume.
raises ***VBoxErrorPasswordIncorrect*** The password provided wasn't correct for at least one disk using the provided
ID.

attach_usb_device(*id_p*, *capture_filename*)
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: *USBDeviceState.busy* , *USBDeviceState.available* or *USBDeviceState.held* , otherwise an error is immediately returned.

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

IUSBDeviceFilters.device_filters() , *USBDeviceState*
in id_p of type str UUID of the host USB device to attach.
in capture_filename of type str Filename to capture the USB traffic to.
raises ***VBoxErrorInvalidVmState*** Virtual machine state neither Running nor Paused.
raises ***VBoxErrorPdmError*** Virtual machine does not have a USB controller.

attached_pci_devices
Get IPCIDeviceAttachment value for 'attachedPCIDevices' Array of PCI devices attached to this machine.

clear_all_disk_encryption_passwords()
Clears all provided supplied disk encryption passwords.

create_shared_folder(*name*, *host_path*, *writable*, *automount*)
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.
in name of type str Unique logical name of the shared folder.
in host_path of type str Full path to the shared folder in the host file system.
in writable of type bool Whether the share is writable or readonly
in automount of type bool Whether the share gets automatically mounted by the guest or not.
raises ***VBoxErrorInvalidVmState*** Virtual machine in Saved state or currently changing state.
raises ***VBoxErrorFileError*** Shared folder already exists or not accessible.

debugger
Get IMachineDebugger value for 'debugger' Debugging interface.

detach_usb_device(*id_p*)
Detaches an USB device with the given UUID from the USB controller of the virtual machine.

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

IUSBDeviceFilters.device_filters() , *USBDeviceState*
in id_p of type str UUID of the USB device to detach.
return device of type ***IUSBDevice*** Detached USB device.
raises ***VBoxErrorPdmError*** Virtual machine does not have a USB controller.

raises *OleErrorInvalidarg* USB device not attached to this virtual machine.

display
Get IDisplay value for 'display' Virtual display object.

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

emulated_usb
Get IEmulatedUSB value for 'emulatedUSB' Interface that manages emulated USB devices.

event_source
Get IEventSource value for 'eventSource' Event source for console events.

find_usb_device_by_address(*name*)
Searches for a USB device with the given host address.

IUSBDevice.address()
in name of type str Address of the USB device (as assigned by the host) to search for.
return device of type *IUSBDevice* Found USB device object.
raises *VBoxErrorObjectNotFound* Given @c name does not correspond to any USB device.

find_usb_device_by_id(*id_p*)
Searches for a USB device with the given UUID.

IUSBDevice.id_p()
in id_p of type str UUID of the USB device to search for.
return device of type *IUSBDevice* Found USB device object.
raises *VBoxErrorObjectNotFound* Given @c id does not correspond to any USB device.

get_device_activity(*type_p*)
Gets the current activity type of given devices or device groups.

in type_p of type *DeviceType*

return activity of type *DeviceActivity*
raises *OleErrorInvalidarg* Invalid device type.

get_guest_entered_acpi_mode()
Checks if the guest entered the ACPI mode G0 (working) or G1 (sleeping). If this method returns @c false, the guest will most likely not respond to external ACPI events.

return entered of type bool
raises *VBoxErrorInvalidVmState* Virtual machine not in Running state.

get_power_button_handled()
Checks if the last power button event was handled by guest.

return handled of type bool
raises *VBoxErrorPdmError* Checking if the event was handled by the guest OS failed.

guest
Get IGuest value for 'guest' Guest object.

keyboard
Get IKeyboard value for 'keyboard' Virtual keyboard object.

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

machine
Get IMachine value for 'machine' Machine object for this console session.

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

mouse
Get IMouse value for 'mouse' Virtual mouse object.

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

pause()
Pauses the virtual machine execution.
raises *VBoxErrorInvalidVmState* Virtual machine not in Running state.
raises *VBoxErrorVmError* Virtual machine error in suspend operation.

power_button()
Sends the ACPI power button event to the guest.
raises *VBoxErrorInvalidVmState* Virtual machine not in Running state.
raises *VBoxErrorPdmError* Controlled power off failed.

power_down()
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.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* Virtual machine must be Running, Paused or Stuck to be powered down.

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

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

If the machine is powered off or aborted, the execution will start from the beginning (as if the real hardware were just powered on).

If the machine is in the *MachineState.saved* state, it will continue its execution the point where the state has been saved.

If the machine *IMachine.teleporter_enabled()* property is enabled on the machine being powered up, the machine will wait for an incoming teleportation in the *MachineState.teleporting_in* state. The returned progress object will have at least three operations where the last three are defined as: (1) powering up and starting TCP server, (2) waiting for incoming teleportations, and (3) perform teleportation. These operations will be reflected as the last three operations of the progress objected returned by *IMachine.launch_vm_process()* as well.

IMachine.save_state()
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* Virtual machine already running.
raises *VBoxErrorHostError* Host interface does not exist or name not set.
raises *VBoxErrorFileError* Invalid saved state file.

power_up_paused()
Identical to powerUp except that the VM will enter the *MachineState.paused* state, instead of *MachineState.running* .

power_up()

return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* Virtual machine already running.
raises *VBoxErrorHostError* Host interface does not exist or name not set.
raises *VBoxErrorFileError* Invalid saved state file.

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

remove_disk_encryption_password(*id_p*)
Removes a password used for hard disk encryption/decryption from the running VM. As soon as the medium requiring this password is accessed the VM is paused with an error and the password must be provided again.
in id_p of type str The identifier used for the password. Must match the identifier used when the encrypted medium was created.

remove_shared_folder(*name*)
Removes a transient shared folder with the given name previously created by *create_shared_folder()* from the collection of shared folders and stops sharing it.
in name of type str Logical name of the shared folder to remove.
raises *VBoxErrorInvalidVmState* Virtual machine in Saved state or currently changing state.
raises *VBoxErrorFileError* Shared folder does not exists.

reset()
Resets the virtual machine.
raises *VBoxErrorInvalidVmState* Virtual machine not in Running state.
raises *VBoxErrorVmError* Virtual machine error in reset operation.

resume()
Resumes the virtual machine execution.
raises *VBoxErrorInvalidVmState* Virtual machine not in Paused state.
raises *VBoxErrorVmError* Virtual machine error in resume operation.

shared_folders
Get ISharedFolder value for '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.shared_folders()* 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 *create_shared_folder()* . Existing shared folders can be removed using *remove_shared_folder()* .

sleep_button()
Sends the ACPI sleep button event to the guest.
raises *VBoxErrorInvalidVmState* Virtual machine not in Running state.
raises *VBoxErrorPdmError* Sending sleep button event failed.

state
Get MachineState value for 'state' Current execution state of the machine.

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

teleport(*hostname*, *tcpport*, *password*, *max_downtime*)
Teleport the VM to a different host machine or process.

@todo Explain the details.
in hostname of type str The name or IP of the host to teleport to.
in tcpport of type int The TCP port to connect to (1..65535).
in password of type str The password.
in max_downtime of type int The maximum allowed downtime given as milliseconds. 0 is not a valid value. Recommended value: 250 ms.

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

The current implementation treats this a guideline, not as an absolute rule.
return progress of type *IProgress* Progress object to track the operation completion.
raises *VBoxErrorInvalidVmState* Virtual machine not running or paused.

usb_devices
Get IUSBDevice value for 'USBDevices' Collection of USB devices currently attached to the virtual USB controller.

The collection is empty if the machine is not running.

use_host_clipboard
Get or set bool value for 'useHostClipboard' Whether the guest clipboard should be connected to the host one or whether it should only be allowed access to the VRDE clipboard. This setting may not affect existing guest clipboard connections which are already connected to the host clipboard.

vrde_server_info
Get IVRDEServerInfo value for 'VRDEServerInfo' Interface that provides information on Remote Desktop Extension (VRDE) connection.

class virtualbox.library.**IAppliance**(*interface=None*)
Represents a platform-independent appliance in OVF format. An instance of this is returned by *IVirtualBox.create_appliance()* , which can then be used to import and export virtual machines within an appliance with VirtualBox.

The OVF standard suggests two different physical file formats:

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

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

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

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

Call *IVirtualBox.create_appliance()* . This will create an empty IAppliance object.

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

Next, call *interpret()* , which analyzes the OVF data and sets up the contents of the IAppliance attributes accordingly. These can be inspected by a VirtualBox front-end such as the GUI, and the suggestions can be displayed to the user. In particular, the *virtual_system_descriptions()* array contains instances of

IVirtualSystemDescription which represent the virtual systems (machines) in the OVF, which in turn describe the virtual hardware prescribed by the OVF (network and hardware adapters, virtual disk images, memory size and so on). The GUI can then give the user the option to confirm and/or change these suggestions.

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

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

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

As with importing, first call *IVirtualBox.create_appliance()* to create an empty IAppliance object.

For each machine you would like to export, call *IMachine.export_to()* with the IAppliance object you just created. Each such call creates one instance of *IVirtualSystemDescription* inside the appliance.

If desired, call *IVirtualSystemDescription.set_final_values()* for each virtual system (machine) to override the suggestions made by the *IMachine.export_to()* routine.

Finally, call *write()* with a path specification to have the OVF file written.

read(*ova_path*)
Reads an OVF file into the appliance object.

This method succeeds if the OVF is syntactically valid and, by itself, without errors. The mere fact that this method returns successfully does not mean that VirtualBox supports all features requested by the appliance; this can only be examined after a call to *interpret()* .
in file_p of type str Name of appliance file to open (either with an .ovf or .ova extension, depending on whether the appliance is distributed as a set of files or as a single file, respectively).
return progress of type *IProgress* Progress object to track the operation completion.

find_description(*name*)
Find a description for the given appliance name.

import_machines(*options=None*)
Imports the appliance into VirtualBox by creating instances of *IMachine* and other interfaces that match the information contained in the appliance as closely as possible, as represented by the import instructions in the *virtual_system_descriptions()* array.

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

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

After the import succeeded, the UUIDs of the IMachine instances created can be retrieved from the *machines()* array attribute.
in options of type *ImportOptions* Options for the importing operation.
return progress of type *IProgress* Progress object to track the operation completion.

add_passwords(*identifiers*, *passwords*)
Adds a list of passwords required to import or export encrypted virtual machines.
in identifiers of type str List of identifiers.
in passwords of type str List of matching passwords.

certificate
Get ICertificate value for 'certificate' The X.509 signing certificate, if the imported OVF was signed, @c null if not signed. This is available after calling *read()* .

create_vfs_explorer(*uri*)
Returns a *IVFSExplorer* object for the given URI.
in uri of type str The URI describing the file system to use.
return explorer of type *IVFSExplorer*

disks
Get str value for 'disks' Array of virtual disk definitions. One such description exists for each disk definition in the OVF; each string array item represents one such piece of disk information, with the information fields separated by tab (t) characters.

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

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

Disk ID (unique string identifier given to disk)

Capacity (unsigned integer indicating the maximum capacity of the disk)

Populated size (optional unsigned integer indicating the current size of the disk; can be approximate; -1 if unspecified)

Format (string identifying the disk format, typically "http://www.vmware.com/specifications/vmdk.html#sparse")

Reference (where to find the disk image, typically a file name; if empty, then the disk should be created on import)

Image size (optional unsigned integer indicating the size of the image, which need not necessarily be the same as the values specified above, since the image may be compressed or sparse; -1 if not specified)

Chunk size (optional unsigned integer if the image is split into chunks; presently unsupported and always -1)

Compression (optional string equaling "gzip" if the image is gzip-compressed)

get_medium_ids_for_password_id(*password_id*)
Returns a list of medium identifiers which use the given password identifier.
in password_id of type str The password identifier to get the medium identifiers for.
return identifiers of type str The list of medium identifiers returned on success.

get_password_ids()
Returns a list of password identifiers which must be supplied to import or export encrypted virtual machines.
return identifiers of type str The list of password identifiers required for export on success.

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

return warnings of type str

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

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

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

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

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

virtual_system_descriptions
Get IVirtualSystemDescription value for 'virtualSystemDescriptions' Array of virtual system descriptions. One such description is created for each virtual system (machine) found in the OVF. This array is empty until either *interpret()* (for import) or *IMachine.export_to()* (for export) has been called.

write(*format_p*, *options*, *path*)
Writes the contents of the appliance exports into a new OVF file.

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

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

in format_p of type str Output format, as a string. Currently supported formats are "ovf-0.9", "ovf-1.0" and "ovf-2.0"; future versions of VirtualBox may support additional formats.

in options of type *ExportOptions* Options for the exporting operation.

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

return progress of type *IProgress* Progress object to track the operation completion.

class virtualbox.library.**IVirtualSystemDescription**(*interface=None*)
Represents one virtual system (machine) in an appliance. This interface is used in the *IAppliance.virtual_system_descriptions()* array. After *IAppliance.interpret()* has been called, that array contains information about how the virtual systems described in the OVF should best be imported into VirtualBox virtual machines. See *IAppliance* for the steps required to import an OVF into VirtualBox.

set_final_value(*description_type*, *value*)
Set the value for the given description type.

in description_type type *VirtualSystemDescriptionType*

in value type str

set_name(*value*)
Set the name of the appliance (name of machine when imported).

set_cpu(*value*)
Set cpu value.

set_memory(*value*)
Set memory value.

set_soundcard(*value*)
Set soundcard value.

set_usb_controller(*value*)
Set usb controller value.

set_network_adapter(*value*)
Set network_adapter value.

set_cdrom(*value*)
Set cdrom value.

set_hard_disk_controller_ide(*value*)
Set hard_disk_controller_ide value.

set_hard_disk_controller_sas(*value*)
Set hard_disk_controller_sas value.

set_hard_disk_controller_sata(*value*)
Set hard_disk_controller_sata value.

set_hard_disk_controller_scsi(*value*)
Set hard_disk_controller_scsi value.

set_hard_disk_image(*value*)
Set hard_disk_image value.

add_description(*type_p*, *v_box_value*, *extra_config_value*)
This method adds an additional description entry to the stack of already available descriptions for this virtual system. This is handy for writing values which aren't directly supported by VirtualBox. One example would be the License type of *VirtualSystemDescriptionType* .

in type_p of type *VirtualSystemDescriptionType*

in v_box_value of type str

in extra_config_value of type str

count
Get int value for 'count' Return the number of virtual system description entries.

get_description()
Returns information about the virtual system as arrays of instruction items. In each array, the items with the same indices correspond and jointly represent an import instruction for VirtualBox.

The list below identifies the value sets that are possible depending on the *VirtualSystemDescriptionType* enum value in the array item in @a aTypes[]. In each case, the array item with the same index in @a OVFValues[] will contain the original value as contained in the OVF file (just for informational purposes), and the corresponding item in @a aVBoxValues[] will contain a suggested value to be used for VirtualBox. Depending on the description type, the @a aExtraConfigValues[] array item may also be used.

"OS": the guest operating system type. There must be exactly one such array item on import. The corresponding item in @a aVBoxValues[] contains the suggested guest operating system for VirtualBox. This will be one of the values listed in *IVirtualBox.guest_os_types()* . The corresponding item in @a OVFValues[] will contain a numerical value that described the operating system in the OVF.

"Name": the name to give to the new virtual machine. There can be at most one such array item; if none is present on import, then an automatic name will be created from the operating

system type. The corresponding item im @a OVFValues[] will contain the suggested virtual machine name from the OVF file, and @a aVBoxValues[] will contain a suggestion for a unique VirtualBox *IMachine* name that does not exist yet.

"Description": an arbitrary description.

"License": the EULA section from the OVF, if present. It is the responsibility of the calling code to display such a license for agreement; the Main API does not enforce any such policy.

Miscellaneous: reserved for future use.

"CPU": the number of CPUs. There can be at most one such item, which will presently be ignored.

"Memory": the amount of guest RAM, in bytes. There can be at most one such array item; if none is present on import, then VirtualBox will set a meaningful default based on the operating system type.

"HardDiskControllerIDE": an IDE hard disk controller. There can be at most two such items. An optional value in @a OVFValues[] and @a aVBoxValues[] can be "PIIX3" or "PIIX4" to specify the type of IDE controller; this corresponds to the ResourceSubType element which VirtualBox writes into the OVF. The matching item in the @a aRefs[] array will contain an integer that items of the "Harddisk" type can use to specify which hard disk controller a virtual disk should be connected to. Note that in OVF, an IDE controller has two channels, corresponding to "master" and "slave" in traditional terminology, whereas the IDE storage controller that VirtualBox supports in its virtual machines supports four channels (primary master, primary slave, secondary master, secondary slave) and thus maps to two IDE controllers in the OVF sense.

"HardDiskControllerSATA": an SATA hard disk controller. There can be at most one such item. This has no value in @a OVFValues[] or @a aVBoxValues[]. The matching item in the @a aRefs[] array will be used as with IDE controllers (see above).

"HardDiskControllerSCSI": a SCSI hard disk controller. There can be at most one such item. The items in @a OVFValues[] and @a aVBoxValues[] will either be "LsiLogic", "BusLogic" or "LsiLogicSas". (Note that in OVF, the LsiLogicSas controller is treated as a SCSI controller whereas VirtualBox considers it a class of storage controllers of its own; see *StorageControllerType*). The matching item in the @a aRefs[] array will be used as with IDE controllers (see above).

"HardDiskImage": a virtual hard disk, most probably as a reference to an image file. There can be an arbitrary number of these items, one for each virtual disk image that accompanies the OVF.

The array item in @a OVFValues[] will contain the file specification from the OVF file (without a path since the image file should be in the same location as the OVF file itself), whereas the item in @a aVBoxValues[] will contain a qualified path specification to where VirtualBox uses the hard disk image. This means that on import the image will be copied and converted from the "ovf" location to the "vbox" location; on export, this will be handled the other way round.

The matching item in the @a aExtraConfigValues[] array must contain a string of the following format: "controller=<index>;channel=<c>" In this string, <index> must be an integer specifying the hard disk controller to connect the image to. That number must be the index of an array item with one of the hard disk controller types (HardDiskControllerSCSI, HardDiskControllerSATA, HardDiskControllerIDE). In addition, <c> must specify the channel to use on that controller. For IDE controllers, this can be 0 or 1 for master or slave, respectively. For compatibility with VirtualBox versions before 3.2, the values 2 and 3 (for secondary master and secondary slave) are also supported, but no longer exported. For SATA and SCSI controllers, the channel can range from 0-29.

"CDROM": a virtual CD-ROM drive. The matching item in @a aExtraConfigValue[] contains the same attachment information as with "HardDiskImage" items.

"CDROM": a virtual floppy drive. The matching item in @a aExtraConfigValue[] contains the same attachment information as with "HardDiskImage" items.

"NetworkAdapter": a network adapter. The array item in @a aVBoxValues[] will specify the hardware for the network adapter, whereas the array item in @a aExtraConfigValues[] will have a string of the "type=<X>" format, where <X> must be either "NAT" or "Bridged".

"USBController": a USB controller. There can be at most one such item. If, and only if, such an item is present, USB support will be enabled for the new virtual machine.

"SoundCard": a sound card. There can be at most one such item. If and only if such an item is present, sound support will be enabled for the new virtual machine. Note that the virtual machine in VirtualBox will always be presented with the standard VirtualBox soundcard, which may be different from the virtual soundcard expected by the appliance.

out types of type *VirtualSystemDescriptionType*

out refs of type str

out ovf_values of type str

out v_box_values of type str

out extra_config_values of type str

get_description_by_type(*type_p*)
This is the same as *get_description()* except that you can specify which types should be returned.

in type_p of type *VirtualSystemDescriptionType*

out types of type *VirtualSystemDescriptionType*

out refs of type str

out ovf_values of type str

out v_box_values of type str

out extra_config_values of type str

get_values_by_type(*type_p*, *which*)
This is the same as *get_description_by_type()* except that you can specify which value types should be returned. See *VirtualSystemDescriptionValueType* for possible values.

in type_p of type *VirtualSystemDescriptionType*

in which of type *VirtualSystemDescriptionValueType*

return values of type str

set_final_values(*enabled*, *v_box_values*, *extra_config_values*)
This method allows the appliance's user to change the configuration for the virtual system descriptions. For each array item returned from *get_description()* , you must pass in one boolean value and one configuration value.

Each item in the boolean array determines whether the particular configuration item should be enabled. You can only disable items of the types HardDiskControllerIDE, HardDiskControllerSATA, HardDiskControllerSCSI, HardDiskImage, CDROM, Floppy, NetworkAdapter, USBController and SoundCard.

For the "vbox" and "extra configuration" values, if you pass in the same arrays as returned in the aVBoxValues and aExtraConfigValues arrays from *get_description()* , the configuration remains unchanged. Please see the documentation for *get_description()* for valid configuration values for the individual array item types. If the corresponding item in the aEnabled array is @c false, the configuration value is ignored.

in enabled of type bool

in v_box_values of type str

in extra_config_values of type str

3.6 virtualbox.library_base – base types used by library.py

The *virtualbox.library_base* provides the base types used by *virtualbox.library*. This module provides the base types used by *virtualbox.library*.

class virtualbox.library_base.**EnumType**(*name*, *bases*, *dct*)
EnumType is a metaclass for Enum. It is responsible for configuring the Enum class object's values defined in Enum.lookup_label

virtualbox.library_base.**add_metaclass**(*metaclass*)
Class decorator for creating a class with a metaclass.

class virtualbox.library_base.**Enum**(*value*)
Enum objects provide a container for VirtualBox enumerations

exception virtualbox.library_base.**VBoxError**
Generic VBoxError

class virtualbox.library_base.**Interface**(*interface=None*)
Interface objects provide a wrapper for the VirtualBox COM objects

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Project hosting provided by github.com and package distribution through PyPi.

[mjdorma+pyvbox@gmail.com]

Python Module Index

v

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z