SPICE
Project

VD Interfaces V0.1

Draft 2

Table of Contents

1. Introduction

Virtual Device Interfaces (VDI) provide a standard way to publish interfaces of virtual devices by a software component. This enables other software components to interact with these devices. Going forward, the first component will be called the back-end and the second component will be called the front-end. An example for using Virtual Device Interfaces is as part of a virtual machine system, where the back-end will be the hardware emulation layer. The back-end will expose interfaces like display port, mouse input etc. The front-end will plug into the display output and will render its output according to it's specific implementation. The front-end will also plug into the mouse input and send mouse events to be processed by the back-end. In addition many other interface types can be exposed by the back-end. Another example of back-end is a remote display system in a physical machine environment. Here, the back-and is implemented using known techniques for interacting with the native OS for the purpose of receiving display updates and pushing inputs. The back-end exposes interfaces like display output, mouse input etc. The front-end can be exactly the same as in the previous example.

By using VDI one back-end can use many types of front-ends without any special code modification. It is also possible for the back-end to dynamically switch front-ends, and improve back-end usability and flexibility. The use of front-ends by many back-ends allows for a better sharing of development, maintenance, and overall product quality.

2. Basic operation

Back-end to front-end interaction is initiated by back-end. Back-end uses VDI_init symbol to pass its Core interface to the front-end. In addition to core interface, back-end also passes options argument to the front-end. Options argument is a string holding initialization argument specific to front-end implementation. Core interface, like every other interface in VDI, is represented as a data structure containing data members and member functions. Every interface starts with a common base structure “VDInterface”. The common structure contains information for identifying the type of the interface and the instance within the type group (i.e. The actual unique is {type, id}). Core interface provides basic functionality for attaching with other interfaces that the back-end implements. Core interface provides methods for receiving interface change events (i.e. interface added event and removing interface event) and for enumerating registers. Using these services, the front-end can find and interact with other interfaces that back-end publishes. The front-end uses VDI type for knowing what functionality the interface provides and the means to interact with the interface (i.e interface specific logic function and data members).

3. Front-end public symbols

VDI defines a minimal set of external symbols for enabling back-end to initiate with the front-end. The back-end can load front-end and retrieve front-end external symbols dynamically at run-time using native platform services. Alternatively, the front-end static or shared library can be linked to the back-end during back-end build process.

- int **VDI_init**(const char *args, CoreInterface *core)

Front-and initialization function. The first parameter is options argument, the content of which depends on the specific front-end. The second parameter is core interface that provides the basic communication channel between front-end and back-end, core interface will be explained later on. VDI_init return value is VDI_ERROR_?

- const char ****VDI_get_usage**()

  This function return array of strings in human readable form for displaying the available options argument that can be pass in args parameter of VDI_init call.

4. Front-end termination

No explicit symbol is defined for front-end termination. Removing core interface by back-end implicitly instructs front end to terminate itself.

5. Common Definition

VDI_ERROR_OK = 0

VDI_ERROR_ERROR = -1

VM_INTERFACE_VERSION = 2. Current VDI version, front-end an back-end must have the same VDI version for proper operation.

VDIObjectRef – opaque object reference that passes between back-end and front-end

VDI_INVALID_OBJECT_REF = 0

6. Base structure of VD interface.

The following is definition of VD Interface, all VDI interfaces are derived from VD Interface.

UINT32 **version**	- VDI version must be VM_INTERFACE_VERSION]
CONST CHAR ***type**	- type of the interface
UNSIGNED INT **id**	- unique id of the interface
CONST CHAR ***description**	- interface human readable interface description
UINT32 **major_version**	- interface major version. Front-end and back-end interfaces having different major_version are not compatible.

UINT32 **minor_version** - interface minor version. Minor version increments on changes that add functionality but do not break compatible and will be reset on major version increment.

7. Core interface

Core interface is the main interface for front-end to back-end communication. It is not allowed to have more the one interface with this type. Removal of Core interface by the back-end is explicit front-end termination.

7.1. VDInterface members value

type = “core”
major_version = 1
minor_version = 0
id = 0

7.2. Types

VDI_CHANGE_NEW = 0
VDI_CHANGE_REMOVING = 1

void (***vdi_interface_change_notifier_t**)(void *opaque, VDInterface *interface,
VDInterfaceChangeType change)

Callback for receiving notification event on adding and removing VDI interfaces.

opaque – back-end passes front-end's opaque value that was passed during notifier registration.

Interface - the relevant interface

change – type of change VDI_CHANGE_NEW or VDI_CHANGE_REMOVING

void (***timer_callback_t**)(void *opaque)

Callback for receiving timer event.

opaque – back-end passes front-end's opaque value that was passed during timer registration.

VDI_LOG_LEVEL_ERROR = 1
VDI_LOG_LEVEL_WARN = 2
VDI_LOG_LEVEL_INFO = 3

7.3. VDInterface specific extension

- VDInterface *(* **next**)(CoreInterface *core, VDInterface *prev)

  Get next interface – return interface following prev. If prev is NULL return first interface. If next return NULL then prev is the last interface. The function is used for interfaces discovery.

- VDIObjectRef (***register_change_notifier**)(CoreInterface *core, void *opaque,
  vdi_interface_change_notifier_t in_notifier)

  Register in_notifier to be called on interface change events. opaque will be pushed back by back-end in notify call. On success the function return notifier object ref otherwise VDI_INVALID_OBJECT_REF is returned.

- void (***unregister_change_notifiers**)(CoreInterface *core, VDObjectRef notifier)

  Unregister interface change notifier hat was registered using register_change_notifiers.

- VDIObjectRef (***create_timer**)(CoreInterface *core, timer_callback_t callback,
  void* opaque)

  Create timer object. The timer handler will call callback on time expiration. opaque will be pushed back by back-end in callback call. On success the function return timer object ref otherwise VDI_INVALID_OBJECT_REF is returned.

- void (***arm_timer**)(CoreInterface *core, VDIObjectRef timer, uint32_t ms)

  Schedule timer object to fire it's callback after ms. Timer is VDI ref object returned by create_timer and ms is time in millisecond.

- void (***disarm_timer**)(CoreInterface *core, VDIObjectRef timer)

  Cancel previous **arm_timer scheduling.** Timer is VDI ref object returned by create_timer

- void (***destroy_timer**)(CoreInterface *core, VDIObjectRef timer)

  Destroy timer object. Timer is vdi ref object returned by create_timer.

- int (***set_file_handlers**)(CoreInterface *core, int fd, void (*on_read)(void *),
  void (*on_write)(void *), void *opaque)

  Request notification on file descriptor for non blocking operation. Fd is the target file descriptor. Back-end will call on_read, if not NULL, on read events passing opaque as first argument. Back-end will call on_write, if not NULL, on write events passing opaque as first argument. On success VDI_ERROR_OK is return.

- void (***term_printf**)(CoreInterface *core, const char* format, ...)

  Send message to back-end interactive terminal if exist. format is printf format, following format are variable arguments list that are format arguments.

- void (***log**)(CoreInterface *core, LogLevel level, const char* component,
  const char* format, ...)

  Send message to be logged using back-end logging service. Level is one of VDI_LOG_LEVEL_?. Component is component name that will be attached as prefix to the message. Format is printf format, following format are variable arguments list that are format arguments.

8. Other interfaces

Core interface is the only interface that is an integral part of VDI specifications. Any other interface can be defined independently of this specification. It's required that both the back-and the front-end know specific interface definitions for proper operation. The only restriction for defining new interface is to ensure the uniqueness of VD Interface type. The following interfaces are known VD interfaces that discussed here as a reference for better understanding of the specification.

9. Keyboard interface

Keyboard interface exposes standard keyboard functionally. It enables pushing scan-code set 1 codes and getting keyboard LED information by query or change notifications mechanism.

9.1. VDInterface members value

type	= “keyboard”
major_version	= 1
minor_version	= 0
id	= keyboard instance id

9.2. Types

VDI_KEYBOARD_LED_SCROLL_LOCK	0
VDI_KEYBOARD_LED_NUM_LOCK	1
VDI_KEYBOARD_LED_CAPS_LOCK	2

void (*keyboard**_leads_notifier_t**)(void *opaque, uint8_t leds)

Callback for receiving notification event on keyboard LED changes.

opaque – back-end passed front-end's opaque value that was passed during notifier registration.

leds – current LED state mask. LED X is on if bit VDI_KEYBOARD_LED_X is set.

9.3. VDInterface specific extension

- void (***push_scan_freg**)(KeyboardInterface *keyboard, uint8_t frag)

  Push scan-code code fragment to the keyboard. Frag is part of scan code. For example calling push_scan_freg(0xe0) push_scan_freg(0x1d) will result with Right CTRL.

- uint8_t (***get_leds**)(KeyboardInterface *keyboard)

  Get current keyboard lads mask. Return lads mask (see lesd in 8.2)

- VDIObjectRef (***register_leds_notifier**)(KeyboardInterface *keyboard,
  keyboard_leads_notifier_t notifier,
  void *opaque)

  Register for LED notification. Back-end will call notifier on every keyboard LED change. Opaque will be pushed back as notifier argument. On success, the function return notifier object ref otherwise VDI_INVALID_OBJECT_REF is returned.

- void (***unregister_leds_notifier**)(KeyboardInterface *keyboard, VDObjectRef notifier)

  Unregister LED change notifier that was registered using register_leds_notifier.

10. Mouse interface

Mouse interface for pushing mouse motion and mouse buttons state events.

10.1. VDInterface members value

type	= “mouse”
major_version	= 1
minor_version	= 0
id	= mouse instance id

10.2. Types

VDI_MOUSE_BUTTON_LEFT 0

VDI_MOUSE_BUTTON_RIGHT	1
VDI_MOUSE_BUTTON_MIDEL	2

10.3. VDInterface specific extension

- void (***motion**)(MouseInterface* mouse, int dx, int dy, int dz, uint32_t buttons_state)

  Push mouse motion and buttons state. dx and dy are mouse motion in pixels in x and y axis, positive dx is rightward motion and positive dy is downward motion. dz is scroll wheel ticks, positive value is scroll down. buttons_state is buttons state mask. Button X is press if bit VDI_MOUSE_BUTTON_X is set.

- void (***buttons**)(MouseInterface* mouse, uint32_t buttons_state)

  Push mouse buttons state. buttons_state is the same as in the motion call

11. Tablet interface

Tablet interface for supporting absolute mouse pointing.

11.1. Interface members value

type	= “tablet”
major_version	= 1
minor_version	= 0
id	= tablet instance id

11.2. Types

VDI_TABLET_BUTTON_LEFT	0
VDI_TABLET_BUTTON_RIGHT	1
VDI_TABLET_BUTTON_MIDEL	2

11.3. VDInterface specific extension

void (***set_logical_size**)(TabletInterface* tablet, int width, int height)

Set logical dimension of the tablet. The logical dimension will be used by the tablet device for mapping position to local coordinate system.

void (***position**)(TabletInterface* tablet, int x, int y, uint32_t buttons_state)

Push mouse position and mouse button state. {x, y} position is relative to the upper left corner. Positive value on x axis advances rightward and positive value on y axis advances downward. buttons_state is buttons state mask. Button X is press if bit VDI_**TABLET**_BUTTON_X is set.

void (***wheel**)(TabletInterface* tablet, int wheel_motion, uint32_t buttons_state)

Push scroll wheel motion and buttons state. wheel_motion is scroll sicks. positive wheel_motion is for scroll down. buttons_state is as in “position”

void (***buttons**)(TabletInterface* tablet, uint32_t buttons_state)

Push buttons state. buttons_state is as in “position”

12. Playback interface

Playback interface for receiving audio stream for playback. Playback use fix audio configuration, channels = 2, frequency = 44100 and channel sample format signed 16bit.

12.1. VDInterface members value

type	= “playback”
major_version	= 1
minor_version	= 0
id	= playback instance id

12.2. PlaybackPlug

uint32_t **major_version**

plug major version. Changing major version breaks compatibility.

uint32_t **minor_version**

interface minor version. Minor version increments on changes that add functionality but do not break compatible and will be reset on major version increment.

void (***start**)(PlaybackPlug *plug)

back-end will call this callback while staring audio playback void (***stop**)(PlaybackPlug *plug)

back-end will call this callback while stopping audio playback

void (***get_frame**)(PlaybackPlug *plug, uint32_t **frame, uint32_t *samples)

back-end will call this callback for getting audio frame. On return *frame will point to audio frame capable of holding *samples samples (i.e frame size is samples * 4). The frame will be pushed back filled with audio samples by the back-end. If no frame is available *frame will be NULL.

void (***put_frame**)(PlaybackPlug *plug, uint32_t *frame)

back-end will call this callback to deliver audio frame for playback. Frame is one that was returns by get_frame. This call also moves frame ownership back to the front-end.

12.3. VDInterface specific extension

VDIObjectRef (***plug**)(PlaybackInterface *playback, PlaybackPlug* plug, int *active)

Attach playback plug to playback interface instance. Plug is the plug to attach. On return, active will be 0 if playback is stopped and 1 if playback is running. On success, the function return object ref otherwise VDI_INVALID_OBJECT_REF is returned.

void (***unplug**)(PlaybackInterface *playback, VDIObjectRef)

Unplug playback that was plugged using plug.

13. Record interface

Record interface for pushing audio capture to back-end. Record use fix audio configuration, channels = 2, frequency = 44100 and channel sample format signed 16bit.

13.1. VDInterface members value

type	= “record”
major_version	= 1
minor_version	= 0
id	= record instance id

13.2. RecordPlug

uint32_t **minor_version**

plug major version. Changing major version breaks compatibility.

uint32_t **major_version**

interface minor version. Minor version increments on changes that add functionality but do not break compatible and will be reset on major version increment.

void (***start**)(RecordPlug *plug)

back-end will call this callback for staring audio capture

void (***stop**)(RecordPlug *plug)

back-end will call this callback to stop audio capture

uint32_t (***read**)(RecordPlug *plug, uint32_t num_samples, uint32_t *samples)

back-end will call this callback to receive audio capture samples. num_samples is the requested number samples. Samples is buffer for receiving samples (i.e samples size is num_samples * 4). Read return number of samples returned.

13.3. VDInterface specific extension

VDIObjectRef (***plug**)(RecordInterface *recorder, RecordPlug* plug, int *active)

Attach record plug to interface instance. Plug is the plug to attach. On return, active will be 0 if record is stopped and 1 if record is running. On success, the function return object ref otherwise VDI_INVALID_OBJECT_REF is returned.

void (***unplug**)(RecordInterface *recorder, VDIObjectRef)

Unplug record plug that was plugged using plug.

14. Additional registered interfaces

- QXL interface "qxl" – QXL interface enable front-end to plug into QXL display device in order to receive QXL and to render it's output locally or remotely.
- VDI port interface "vdi_port" – interface for plugging, reading and writing to vdi_port. vdi_port is used for communication with an agent residing on gust machine.
- Migration interface "migration" - interface for integrating with back-end migration process. This interface enable registering for pre and post migration hooks.
- Qemu terminal interface "qemu_terminal" – interface for integrating with QEMU style terminal.