Open Virtual Platforms
VMI OS Support Function Reference

Imperas Software Limited
Imperas Buildings, North Weston,
Thame, Oxfordshire, OX9 2HA, UK
docs@imperas.com

Author:	Imperas Software Limited
Version:	6.9.0
Filename:	OVP_VMI_OS_Support_Function_Reference.doc
Project:	OVP VMI OS Support Reference
Last Saved:	Thursday, 19 March 2015
Keywords:	

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

TABLE OF CONTENTS

1 Introduction

This is reference documentation for the VMI *OS Support* function interface, defined in ImpPublic/include/host/vmi/vmiOSLib.h. The functions in this interface are intended to be used at *run time* in order to delegate low level operating system tasks, such as file IO, to the host system. This delegation mechanism is referred to as *semi-hosting*.

Functions in the OS support interface have the prefix `vmios`.

As far as possible, the functions provided have equivalent behavior under all supported host platforms. To achieve this, the vmios API provides its own version of certain structure types and flag values that are passed as parameters to vmios functions. Any unavoidable host-specific behavior is described in each individual function section.

2 QuantumLeap Semantics

As of VMI version 6.0.0, Imperas Professional Simulation products implement a parallel simulation algorithm called *QuantumLeap*, which enables multicore platform simulation to be distributed over separate threads on multiple cores of the host machine for improved performance. Refer to the *OVP and CpuManager User Guide* for more information about QuantumLeap usage.

When QuantumLeap is active, some functions require the current thread to be suspended until all other concurrent threads have been stopped so that they may safely execute. There are three categories of function with different semantics:

Thread Safe
Thread safe functions never cause the current thread to be suspended.

Synchronizing
Synchronizing functions always cause the current thread to be suspended until all other threads have been safely stopped.

Non-self-synchronizing
Non-self-synchronizing functions are passed a processor as an argument. They cause the current thread to be suspended only if the processor is not the current processor.

All functions described in this manual are *synchronizing*.

3 Interface to Processor Model

3.1 vmiosGetRegDesc

This function has been removed. `vmiosGetRegDesc` is now a `#define` targeting `vmirtGetRegByName`. See the *VMI Run Time Function Reference* for more details.

3.2 vmiosRegRead

This function has been removed. `vmiosRegRead` is now a `#define` targeting `vmirtRegRead`. See the *VMI Run Time Function Reference* for more details.

3.3 vmiosRegWrite

This function has been removed. `vmiosRegWrite` is now a `#define` targeting `vmirtRegWrite`. See the *VMI Run Time Function Reference* for more details.

4 File Operations

4.1 vmiosOpen

Prototype

```
Int32 vmiosOpen(vmiProcessorP processor,
                const char *path,
                Int32 flags,
                Int32 mode);
```

Description

This function returns a file descriptor for the file with the specified path. If the open fails, the invalid file descriptor value -1 is returned.

The `flags` value should be one of the following values:

VMIOS_O_RDONLY
VMIOS_O_WRONLY
VMIOS_O_RDWR

bitwise-or'd with any number of the following:

VMIOS_O_CREAT
VMIOS_O_TRUNC
VMIOS_O_APPEND

The parameter `mode` is only applicable when VMIOS_O_CREAT is included in the flag value and the file does not already exist (that is, when a new file will actually be created), and is ignored in all other cases. The `mode` value is a bit-mask of file permissions read, write, and execute for each of user, group, and other (i.e. standard Unix file permissions).

Example

```
#include "vmi/vmiOSLib.h"

Uns32 fd = vmiosOpen(proc, path, (VMIOS_O_RDWR | VMIOS_O_CREAT), 0666);

if (fd < 0) {
  // failed to open the file
  …
} else {
  // the file was opened successfully
  …
}
```

Notes and Restrictions

If the host is Windows, then only the user permissions of the `mode value` are used when a new file is created, and the remaining bits are ignored.

The simulator and the OS support functions share the same file descriptors. In some cases this may lead to artifacts where the simulator logging and semi-hosting functions using the OS support functions interfere. This is particularly likely when stdout or stderr are closed (simulator logging may be suppressed), or reopened (simulator logging may be redirected to unexpected places).

It is recommended that semi-hosting functions that use file descriptors avoid these situations by mapping file descriptors used by the semi-hosting functions to unique values.

4.2 vmiosClose

Prototype

```
Int32 vmiosClose(vmiProcessorP processor,
                 Int32 fd);
```

Description

This function closes specified file descriptor, and it will no longer be associated with any file. The same file descriptor may be reused by a subsequent call to vmiosOpen().

If the operation succeeds 0 is returned, and if it fails -1 is returned.

Example

```
#include “vmi/vmiOSLib.h”

…

if(vmiosClose(proc, fd) == 0) {
  // the file descriptor was closed successfully
  fd = -1;
  …
} else {
  // failed to close the file descriptor
  …
}
```

Notes and Restrictions

As for vmiosOpen.

4.3 vmiosRead

Prototype

```
Int32 vmiosRead(vmiProcessorP processor,
                Int32 fd,
                memDomainP domain,
                Addr buf,
                Uns32 count);
```

Description

This function attempts to read the number of bytes specified by `count` from file descriptor `fd` into the buffer in simulated memory starting at the address specified by `buf`.

If the operation fails, -1 is returned. Otherwise the actual number of bytes read is returned – this will be 0 if the end of file is encountered before any bytes are read.

If `count` is 0, then no bytes are read, and 0 is returned.

Example

```
#include “vmi/vmiOSLib.h”

…

numBytesRead = vmiosRead(proc, fd, domain, buffer, maxCount);

if (numBytesRead == 0) {
  // unable to read any bytes
  …
}
…
```

Notes and Restrictions

None.

4.4 vmiosWrite

Prototype

```
Int32 vmiosWrite(vmiProcessorP processor,
                 Int32 fd,
                 memDomainP domain,
                 Addr buf,
                 Uns32 count);
```

Description

This function attempts to write `count` bytes from the buffer within simulated memory starting at `buf` to the file referenced by file descriptor fd.

If the operation fails, -1 is returned. Otherwise the number of bytes actually written is returned. If count is 0, then no bytes will be written and 0 is returned.

Example

```
#include “vmi/vmiOSLib.h”

…

numBytesWritten = vmiosWrite(proc, fd, domain, buffer, count);

if (numBytesRead !== count) {
  // write was unsuccessful
  …
}
…
```

Notes and Restrictions

None.

4.5 vmiosUnlink

Prototype

```
Int32 vmiosUnlink(vmiProcessorP processor,
                  Const char *path);
```

Description

This function attempts to delete the file specified by `path`.

If the delete operation succeeds, 0 is returned. If the operation fails, -1 is returned.

Example

```
#include “vmi/vmiOSLib.h”

…

if (vmiosUnlink(proc, pathname) == -1) {
  // unlink failed
  …
}
```

Notes and Restrictions

None.

4.6 vmiosStat

Prototype

```
Int32 vmiosStat(vmiProcessorP processor,
                const char *path,
                vmiosStatBufP buf);
```

Description

This function populated the buffer `buf` with information about the file specified by pathname. The vmiosStatBuf structure is described below.

The value returned is 0 if the operation succeeded and -1 if it failed.

Example

```
#include “vmi/vmiOSLib.h”

vmiosStatBuf buf = {0};

…

retcode = vmiosStat(proc, pathname, &buf);

if (retcode == 0) {
  fileLen = buf.size;
}
```

…

Notes and Restrictions

The vmiosStatBuf structure type contains the following fields:

Type	Name	Description
Uns32	mode	The mode of the file given as a bit-mask: • the low order 9 bits indicates read/write/execute permissions for user/group/other (group/other are not applicable for a Windows host) • VMIOS_S_IFREG indicates a regular file • VMIOS_S_IFDIR indicates a directory
Uns64	size	The size of the file in bytes.
Uns32	blksize	The blocksize used by the device containing the file. A value of 1 will be used for devices that don’t report a blocksize.
Uns32	blocks	The number of blocks occupied by the file. Note that blksize * vmios_blocks may be significantly larger than size.
Uns32	atime	The time of the most recent access of the file as seconds since the Epoc (00:00:00 UTC, January 1, 1970)
Uns32	ctime	The time of the most recent status change of the file as seconds

		since the Epoc.
Uns32	mtime	The time of the most recent modification of the file as seconds since the Epoc.

4.7 vmiosLStat

Prototype

```
Int32 vmiosLStat(vmiProcessorP processor,
                 const char *path,
                 vmiosStatBufP buf);
```

Description

This function returns information about the file specified by pathname.

If pathname refers to a symbolic-link, then information about the link itself (rather than the file to which the link refers) is returned. If the host is Windows, this function behaves exactly like `vmiosStat()` since symbolic-links are not available

The value returned is 0 if the operation succeeded and -1 if it failed.

Example

```
#include “vmi/vmiOSLib.h”

vmiosStatBuf buf = {0};

…

retcode = vmiosLStat(proc, pathname, &buf);

if (retcode == 0) {
  fileLen = buf.size;
}
```

…

Notes and Restrictions

See `vmiosStat()`.

4.8 vmiosFStat

Prototype

```
Int32 vmiosFstat(vmiProcessorP processor,
                 Int32 fd,
                 vmiosStatBufP buf);
```

Description

This function returns information about the file referenced by file descriptor fd.

The value returned is 0 if the operation succeeded and -1 if it failed.

Example

```
#include “vmi/vmiOSLib.h”

vmiosStatBuf buf = {0};

…

retcode = vmiosFStat(proc, fd, &buf);

if (retcode == 0) {
  fileLen = buf.size;
}
```

…

Notes and Restrictions

See vmiosStat().

4.9 vmiosLSeek

Prototype

```
Int32 vmiosLSeek(vmiProcessorP processor,
                 Int32 fd,
                 Int32 offset
                 Int32 whence);
```

Description

This function repositions the position of the file descriptor fd to the specified signed offset from a position indicated by the value of whence.

Whence must be one of the following constants:

VMIOS_SEEK_SET offset is from the start of the file
VMIOS_SEEK_CUR offset is from the current position within the file
VMIOS_SEEK_END offset is from the end of the file

If the operation succeeds, the resulting offset in bytes from the start of the file is returned. If the operation fails, -1 is returned.

Example

```
#include “vmi/vmiOSLib.h”

…

fileLen = vmiosLSeek(processor, fd, 0, VMIOS_SEEK_END);

…
```

Notes and Restrictions

None.

5 Environment

5.1 vmiosGetArgsLen

This function has been removed.

5.2 vmiosGetArgs

This function has been removed.

5.3 vmiosGetStdin

Prototype

```
Int32 vmiosGetStdin (vmiProcessorP processor);
```

Description

Returns the file descriptor to use for stdin.

5.4 vmiosGetStdout

Prototype

```
Int32 vmiosGetStdout (vmiProcessorP processor);
```

Description

Returns the file descriptor to use for stdout.

5.5 vmiosGetStderr

Prototype

```
Int32 vmiosGetStderr (vmiProcessorP processor);
```

Description

Returns the file descriptor to use for stderr.

5.6 vmiosGetTimeOfDay

Prototype

```
Int32 vmiosGetTimeOfDay(vmiProcessorP processor,
                        vmiosTimeBufP timebuf);
```

Description

This function populates the buffer pointed to by `timebuf` with the (approximate) number of seconds and microseconds since 00:00:00 UTC, January 1970.

Example

```
#include “vmi/vmiOSLib.h”

vmiosTimeBuf buffer;

Int32 retval;

…

retval = vmiosTimeofday(processor, &buffer);

if (retval == 0) {
  Uns32 seconds = buffer.vmios_sec;
  Uns32 microseconds = buffer.usec;
  …
}
```

Notes and Restrictions

On a Windows host the microseconds value is only given to millisecond precision.

Since the value returned is based on the host system’s clock, time within a simulation determined using this function may appear to advance at an unrealistic rate.

Also, in a multi-processor simulation, there is the possibility of times returned by this function to appear out of sequence due to the size of the quanta by which each processor is advanced.

6 Miscellaneous

6.1 Runtime Commands

A runtime command is part of a plugin which can be executed by the simulator. Its typical use is to change the mode of operation of the plugin or to print information from inside it. The functions `vmiosAddCommand` and `vmiosAddCommandParse` are similar to the functions `vmirtAddCommand` and `vmirtAddCommandParse` which are comprehensively documented in *OVP_VMI_Run_Time_Function_Reference.doc*. Please refer to that manual for more detail.

6.2 vmiosAddCommand

Prototype

```
void vmiosAddCommand (
         vmiosObjectP    object,
         const char     *name,
         const char     *help,
         vmiosCommandFn  commandCB,
         vmiCommandAttrs attrs,
);
```

Description

Adds a command with the specified `name` to the passed `vmiosObject`.
When the command is called, the function `commandCB` will be called. The argument `help` is used by the help system and should list the arguments required by the command.

The installed function is passed the object context, the number of arguments (argc) and an array of string arguments argv[]. argv[0] is the command name.

`attrs` may be used to control how the command appears in a graphical interface. See the VMI Run Time Reference Manual for information on the values that may be passed.

Example

```
#include “vmi/vmiOSLib.h”

static VMIOS_COMMAND_FN(myCommand) {
    vmiPrintf("command %s was called, with args...\n", argv[0]);

    int i;
    for(i= 1; i < argc) {
        vmiPrintf("   arg %d=%s\n", i, argv[i]);
    }
    ...
}


  VMIOS_CONSTRUCTOR_FN(constructor) {
      ...
    vmiosAddCommand(
        object, “myCommand”, "-myArg <string>", myCommand, VMI_CT_DEFAULT
    );
    ...
}
```

Notes and Restrictions

The arguments passed in argv[] do not persist after the function call is complete.

6.3 vmiosAddCommandParse

Prototype

```
vmiCommandP vmiosAddCommandParse(
        vmiosObjectPP       object,
        const char         *name,
        const char         *help,
        vmiosCommandParseFn commandCB,
        vmiCommandAttrs     attrs
);
```

Description

Adds a command with the specified `name` to the passed `vmiosObject`. When the command is called, the arguments will be parsed according to the argument specifications provided by `vmirtAddArg`() then the function `commandCB` will be called in the model. The argument `help` is used by the help system and should list the arguments required by the command.

The called function is passed the object context, the number of arguments specified with `vmirtAddArg`() and an array of argument value structures filled with the parsed values, in the order they were originally specified. See `vmirtAddArg`() in the VMI Run Time Reference Manual.

`attrs` may be used to control how the command appears in a graphical interface. See the VMI Run Time Reference Manual for information on the values that may be passed.

6.4 vmiosInstallInterceptNotifier

Prototype

```
void vmiosInstallInterceptNotifier (
         vmiosObjectP     object,
         vmiosNotifierFn  notifierCB,
         void             *userData
);
```

Description

This function installs an opaque intercept notifier for the intercept library. This is called whenever an opaque function intercept is performed, even if the intercept is handled by a different library.

Example

```
#include “vmi/vmiOSLib.h”
…
static VMIOS_INTERCEPT_NOTIFIER_FN(getCount) {
    object->stackPointer--;
}
…
vmiosInstallNotifier(object, opaqueInterceptCalled, NULL);
```

Notes and Restrictions

None.

6.5 vmiosGetLicenseFeature

Prototype

```
Bool vmiosGetLicenseFeature(const char *feature);
```

Description

This routine attempts to check out a license feature with the passed name. The `Bool` return code indicates whether the license was successfully checked out. This function will typically be called from within the constructor.

Example

```
#include “vmi/ vmiOSLib.h”
#include “vmi/vmiMessage.h”

#define LICENSE_NAME    "OS_FEATURE"
#define LICENSE_VERSION "1.0"

//
// Processor constructor
//
VMI_CONSTRUCTOR_FN(cpuxConstructor) {

    // Check for valid CPU license
    if (!vmiosGetLicenseFeature(LICENSE_NAME, LICENSE_VERSION)) {
        vmiMessage("F", "LIC_NA2", "%s",
            vmiosGetLicenseFeatureErrString(LICENSE_NAME)
        );
    }

    . . . etc . . .
}
```

Notes and Restrictions

None.

6.6 vmiosGetLicenseFeatureErrString

Prototype

```
const char *vmiosGetLicenseFeatureErrString(const char *feature);
```

Description

When function `vmiosGetLicenseFeature` fails, this function can be called to get an error string indicating why the checkout failed. Typically, the result should be used in a call to `vmiMessage` with the fatal message identifier `"F"`: this will cause a *fatal* message to be printed and terminate simulation.

Example

```
#include "vmi/ vmiOSLib.h"
#include "vmi/vmiMessage.h"

#define LICENSE_NAME    "OS_FEATURE"
#define LICENSE_VERSION "1.0"

//
// Processor constructor
//
VMI_CONSTRUCTOR_FN(cpuxConstructor) {

    // Check for valid CPU license
    if (!vmiosGetLicenseFeature(LICENSE_NAME, LICENSE_VERSION)) {
        vmiMessage("F", "LIC_NA2", "%s",
            vmiosGetLicenseFeatureErrString(LICENSE_NAME)
        );
    }

    . . . etc . . .
}
```

Notes and Restrictions

None.

##