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:	6.29.0
Filename:	OVP_VMI_OS_Support_Function_Reference.doc
Project:	OVP VMI OS Support Reference
Last Saved:	Monday, 13 January 2020
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

Functions in this section are useful when creating a *semihost library* that retargets simulated file read or write operations to real file operations on the host machine. This is required so that applications compiled for a target operating system can be run in a simulation environment without requiring the target operating system to be simulated as well.

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

This example is taken from the standard *Newlib* semihost library implementation.

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

//
// open (const char *buf, int flags, int mode)
//
static void doOpen(
    vmiProcessorP processor,
    vmiosObjectP  object,
    const char   *context,
    UnsArch       reent,
    UnsArch       pathnameAddr,
    IntArch       flags,
    IntArch       mode
) {
    // get file name from data domain
    memDomainP  domain   = vmirtGetProcessorDataDomain(processor);
    const char *pathname = vmirtGetString(domain, pathnameAddr);

    // implement open
    Int32 vmiosFlags = (flags & 0x003)
                     | ((flags & TARGET_O_CREAT)  ? VMIOS_O_CREAT  : 0)
```

```
                    | ((flags & TARGET_O_APPEND) ? VMIOS_O_APPEND : 0)
                    | ((flags & TARGET_O_TRUNC)  ? VMIOS_O_TRUNC  : 0);

    Int32 result = vmiosOpen(processor, pathname, vmiosFlags, mode);

    if(result != -1) {

        // save file descriptor in simulated descriptor table
        Int32 fdMap = newFileDescriptor(object, context);
        if (fdMap != -1) {
            object->fileDescriptors[fdMap] = result;
        }

        // Return the fileDescriptor (or error)
        result = fdMap;
    }

    setErrnoAndResult(processor, object, result, reent);
}
```

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 host file descriptors that are not shared by the simulator itself. For example, simulated `stdout` or `stderr` should be redirected to files on the host, not to host `stdout` or `stderr`.

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

This example is taken from the standard *Newlib* semihost library implementation.

```
#include “vmi/vmiOSLib.h”

static void doClose(
    vmiProcessorP processor,
    vmiosObjectP  object,
    UnsArch       reent,
    IntArch       fd
) {
    // implement close
    Int32 fdMap  = mapFileDescriptor(processor, object, fd);
    Int32 result = fdMap != -1 ? vmiosClose(processor, fdMap) : -1;

    // close out the semihosted file descriptor if success
    if(result != -1) {
        object->fileDescriptors[fd] = -1;
    }

    // return result
    setErrnoAndResult(processor, object, result, reent);
}
```

Notes and Restrictions

None.

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

This example is taken from the standard *Newlib* semihost library implementation.

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

static void doRead(
    vmiProcessorP processor,
    vmiosObjectP  object,
    UnsArch       reent,
    IntArch       fd,
    UnsArch       buf,
    UnsArch       count
) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    Int32      fdMap  = mapFileDescriptor(processor, object, fd);
    Int32      result = (fdMap != -1) ?
                        vmiosRead(processor, fdMap, domain, buf, count) :
                        -1;

    setErrnoAndResult(processor, object, result, reent);
}
```

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

This example is taken from the standard *Newlib* semihost library implementation.

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

static void doWrite(
    vmiProcessorP processor,
    vmiosObjectP  object,
    UnsArch       reent,
    IntArch       fd,
    UnsArch       buf,
    UnsArch       count
) {
    memDomainP domain = vmirtGetProcessorDataDomain(processor);
    Int32      fdMap  = mapFileDescriptor(processor, object, fd);
    Int32      result = (fdMap != -)1 ?
                        vmiosWrite(processor, fdMap, domain, buf, count) :
                        -1;

    setErrnoAndResult(processor, object, result, reent);
}
```

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

This example is taken from the standard *Newlib* semihost library implementation.

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

static void doUnlink(
    vmiProcessorP processor,
    vmiosObjectP  object,
    UnsArch       reent,
    UnsArch       pathnameAddr
) {
    // get file name from data domain
    memDomainP  domain   = vmirtGetProcessorDataDomain(processor);
    const char *pathname = vmirtGetString(domain, pathnameAddr);

    // implement unlink
    Int32 result = vmiosUnlink(processor, pathname);

    // return result
    setErrnoAndResult(processor, object, result, reent);
}
```

Notes and Restrictions

None.

4.6 vmiosStat

Prototype

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

Description

This function populates the buffer `buf` with information about the file specified by `path`. 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 Epoch (00:00:00 UTC, January 1, 1970)
`Uns32`	`ctime`	The time of the most recent status change of the file as seconds since the Epoch.
`Uns32`	`mtime`	The time of the most recent modification of the file as seconds since the Epoch.

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

Example

This example is taken from the standard *Newlib* semihost library implementation.

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

static void doStat(
    vmiProcessorP processor,
    vmiosObjectP  object,
    UnsArch       reent,
    UnsArch       file_nameAddr,
    UnsArch       bufAddr
) {
    // get file name from data domain
    memDomainP  domain    = vmirtGetProcessorDataDomain(processor);
    const char *file_name = vmirtGetString(domain, file_nameAddr);
```

```
    // implement stat
    vmiosStatBuf statBuf = {0};
    Int32        result  = vmiosStat(processor, file_name, &statBuf);

    // write back results
    if (result != -1) {
        transcribeStatData(processor, bufAddr, &statBuf);
    }

    setErrnoAndResult(processor, object, result, reent);
}
```

Notes and Restrictions

None.

4.7 vmiosLStat

Prototype

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

Description

This function returns information about the file specified by `path`.

If `path` 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.

See section 4.6 for a description of the format of the `vmiosStatBuf` structure.

Example

This example is taken from the standard *Newlib* semihost library implementation.

```
#include “vmi/vmiOSLib.h”

static void doLstat(
    vmiProcessorP processor,
    vmiosObjectP  object,
    UnsArch       reent,
    UnsArch       file_nameAddr,
    UnsArch       bufAddr
) {
    // get file name from data domain
    memDomainP  domain    = vmirtGetProcessorDataDomain(processor);
    const char *file_name = vmirtGetString(domain, file_nameAddr);

    // implement stat
    vmiosStatBuf statBuf = {0};
    Int32        result  = vmiosLStat(processor, file_name, &statBuf);

    // write back results
    if (result != -1) {
        transcribeStatData(processor, bufAddr, &statBuf);
    }
    setErrnoAndResult(processor, object, result, reent);
}
```

Notes and Restrictions

None.

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.

See section 4.6 for a description of the format of the `vmiosStatBuf` structure.

Example

This example is taken from the standard *Newlib* semihost library implementation.

```
#include “vmi/vmiOSLib.h”

static void doFstat(
    vmiProcessorP processor,
    vmiosObjectP  object,
    UnsArch       reent,
    IntArch       filedes,
    UnsArch       bufAddr
) {
    // implement fstat
    vmiosStatBuf statBuf = {0};
    Int32        fdMap   = mapFileDescriptor(processor, object, filedes);
    Int32        result  = (fdMap != -1) ?
                           vmiosFStat(processor, fdMap, &statBuf) :
                           -1;

    // write back results
    if (result != -1) {
        transcribeStatData(processor, bufAddr, &statBuf);
    }

    setErrnoAndResult(processor, object, result, reent);
}
```

Notes and Restrictions

None.

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

This example is taken from the standard *Newlib* semihost library implementation.

```
#include “vmi/vmiOSLib.h”

static void doLseek(
    vmiProcessorP processor,
    vmiosObjectP  object,
    UnsArch       reent,
    IntArch       fd,
    IntArch       offset,
    IntArch       whence
) {
    // implement lseek
    Int32 fdMap  = mapFileDescriptor(processor, object, fd);
    Int32 result = fdMap != -1 ? vmiosLSeek(processor, fdMap, offset, whence) : 1;

    // return result
    setErrnoAndResult(processor, object, result, reent);
}
```

Notes and Restrictions

None.

5 Simulation Environment

Functions in this section are used to obtain resources and settings from the simulator, and to control some aspects of interaction with the simulator.

5.1 vmiosGetExtensionName

Prototype

```
const char *vmiosGetExtensionName(vmiosObjectP object);
```

Description

This function returns the name of a plugin. It is typically used in error messages.

Example

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

#define LICENSE_NAME "OS_FEATURE"

static void licenseCheck(vmiosObjectP object) {
    if(!vmiosGetLicenseFeature(LICENSE_NAME)) {
        vmiMessage(
            "F", PLUGIN_PREFIX,
            "%s: Unable to obtain tool license:\n%s\n",
            vmiosGetExtensionName(object),
            vmiosGetLicenseFeatureErrString(LICENSE_NAME)
        );
    }
}
```

Notes and Restrictions

None.

5.2 vmiosGetStdin, vmiosGetStdout, vmiosGetStderr

Prototypes

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

Description

These functions return suitable file descriptors to use for `stdin`, `stdout` and `stderr` in a semihost library. They will typically be used in the intercept library constructor.

Example

This example is taken from the standard *Newlib* semihost library implementation.

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

//
// Intercept library object
//
typedef struct vmiosObjectS {

    Int32 fileDescriptors[FILE_DES_NUM];

} vmiosObject

//
// Intercept library initialization
//
static void setupNewlib(vmiosObjectP object, vmiProcessorP processor) {

    . . .

    // initialize stdin, stderr and stdout
    object->fileDescriptors[0] = vmiosGetStdin(processor);
    object->fileDescriptors[1] = vmiosGetStdout(processor);
    object->fileDescriptors[2] = vmiosGetStderr(processor);

    . . .
}
```

Notes and Restrictions

None.

5.3 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 the Epoch (00:00:00 UTC, January 1970). The `vmiosTimeBuf` structure type contains the following fields:

Type	Name	Description
`Uns32`	`sec`	Seconds since the epoch
`Uns32`	`usec`	Microseconds in addition to seconds

The time since the Epoch is the sum of seconds and microseconds.

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

Example

This example is taken from the standard *Newlib* semihost library implementation.

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

static void doGettimeofday(
    vmiProcessorP processor,
    vmiosObjectP  object,
    UnsArch       reent,
    UnsArch       tvAddr
) {
    // implement gettimeofday
    vmiosTimeBuf timeBuf = {0};
    Int32        result  = vmiosGetTimeOfDay(processor, &timeBuf);

    // write back results
    if (result != -1 && tvAddr) {
        transcribeTimeData(processor, tvAddr, &timeBuf);
    }

    setErrnoAndResult(processor, object, result, reent);
}
```

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.

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.

5.4 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 implemented by a different intercept library.

Intercept notifiers are useful when writing plugins that analyze call/return addresses as an application runs. When a called function is intercepted (by a semihost library, for example) the body of that function will *not* be executed: the simulator will behave as if there was an implicit return from the function after a single instruction. This can confuse a plugin tool that is decoding instructions to understand the call/return flow though a program. By installing a notifier, the plugin can make sure it is informed about this behavior and take appropriate action.

The intercept notifier function is defined using the `VMIOS_INTERCEPT_NOTIFIER_FN` macro (defined in `vmiTypes.h`):

```
#define VMIOS_INTERCEPT_NOTIFIER_FN(_NAME) void _NAME( \
    vmiProcessorP processor,    \
    vmiosObjectP  object,       \
    const char   *context,      \
    void         *userData      \
)
```

The callback is passed the processor object and the intercept library that installed the notifier as arguments, and a client data pointer (`userData`). Argument `context` is the name of the intercepted function.

Example

This example is extracted from a tool that may record stack depth by intercepting paired call and return instructions as an application runs. When a function is intercepted, there will be a call with no apparent return; to allow for this, the call count is decremented implicitly whenever an opaque intercept occurs.

```
#include “vmi/vmiOSLib.h”

static VMIOS_INTERCEPT_NOTIFIER_FN(opaqueInterceptCalled) {
    object->callCount--;
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    object->callCount = 0;

    . . .
```

```
    // install callback whenever an opaque intercept occurs
    // (to ensure stack push is cancelled)
    vmiosInstallInterceptNotifier(object, opaqueInterceptCalled, NULL);
}
```

Notes and Restrictions

None.

5.5 vmiosGetLicenseFeature

Prototype

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

Description

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

Example

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

#define LICENSE_NAME "OS_FEATURE"

static void licenseCheck(vmiosObjectP object) {
    if(!vmiosGetLicenseFeature(LICENSE_NAME)) {
        vmiMessage(
            "F", PLUGIN_PREFIX,
            "%s: Unable to obtain tool license:\n%s\n",
            vmiosGetExtensionName(object),
            vmiosGetLicenseFeatureErrString(LICENSE_NAME)
        );
    }
}
```

Notes and Restrictions

None.

5.6 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"

static void licenseCheck(vmiosObjectP object) {
    if(!vmiosGetLicenseFeature(LICENSE_NAME)) {
        vmiMessage(
            "F", PLUGIN_PREFIX,
            "%s: Unable to obtain tool license:\n%s\n",
            vmiosGetExtensionName(object),
            vmiosGetLicenseFeatureErrString(LICENSE_NAME)
        );
    }
}
```

Notes and Restrictions

None.

6 Scope Constraints

It is sometimes desirable to restrict a semihost library or plugin so that it is activated only when a processor is in a particular state, indicated by a combination of *processor operating mode* (e.g. user or kernel) and *virtual address state* (typically defined by ASID or VMID). For example, when writing a tool to analyze the behavior of a user application running under a simulated Linux operating system it will be necessary to ensure the tool is only activated when that particular user process is running.

To facilitate implementation of this kind of analysis tool, it is possible to set a scope on a plugin or intercept library, and use this to control operations of that tool.

6.1 vmiosUpdateScope

Prototype

```
Bool vmiosUpdateScope(vmiosObjectP object, const char *scope);
```

Description

This function modifies the effective scope of an intercept library, to restrict its activity to a particular simulated processor state. The argument scope specifies the new effective scope, and must be a string in one of these forms:

```
clear
[<mode>][:<ASID>]
+[<mode>][:<ASID>]
-[<mode>][:<ASID>]
```

The value `clear` indicates that all scope constraints should be removed. The value `<mode>:<ASID>` indicates that the intercept library should be activated only when the processor is operating with that particular mode/ASID combination. The value `+<mode>:<ASID>` indicates that the intercept library should be activated when the processor is operating with that particular mode/ASID combination *in addition to* any previously-specified scope constraints. The value `-<mode>:<ASID>` indicates that the intercept library should *not* be activated when the processor is operating with that particular mode/ASID combination, but any other previously-specified scope constraints specified should be preserved. Either mode or ASID may be omitted – see the examples below.

The values of mode and ASID are processor dependent, but both are 32-bit unsigned values. The currently-effective scope of a processor can be found by `vmirtGetProcessorScope`.

In the simplest case, mode and ASID values will be unsigned 32-bit numbers and will therefore specify a precise operating mode and ASID. It is also possible to specify a mask with either value, using the following syntax:

```
value&mask
```

where both value and mask are 32-bit numbers. When a mask is specified, the scope is deemed to match if:

```
(current_processor_value & mask) == value
```

Scope String Examples

This section gives some examples of scope strings to clarify the allowed syntax.

1. `vmiosUpdateScope(object, “clear”);`
Remove all scope constraints on the semihost library or plugin.

2. `vmiosUpdateScope(object, “3:24”);`
Indicate the plugin is active only when operating in processor mode 3 with ASID 24. Mode and ASID specifications are processor-specific.

3. `vmiosUpdateScope(object, “3”);`
Indicate the plugin is active only when operating in processor mode 3, with any ASID.

4. `vmiosUpdateScope(object, “:24”);`
Indicate the plugin is active only when operating with ASID 24, in any mode.

5. `vmiosUpdateScope(object, “+3:30”);`
Indicate the plugin is should be active when operating in processor mode 3 with ASID 30, in addition to any previously-specified scope.

6. `vmiosUpdateScope(object, “-3:12”);`
Indicate the plugin is should be not be active when operating in processor mode 3 with ASID 12, but preserve any other active scope.

7. `vmiosUpdateScope(object, “3:12&0xff”);`
Indicate the plugin is should be active when operating in processor mode 3 when the processor ASID masked to 8 bits is 12 (in other words, ignore the top 24 bits of the ASID).

Example

This example shows how scopes might be used to constrain an intercept library callback that monitors a function in a Linux user application so that it is activated only when one particular process runs. The example is somewhat contrived because it constrains the intercept library so that it is tied to the *first user process that executes at the intercepted function address*, whether or not this is in fact running the target application. In reality, the scope constraint would be specified by an OS-aware helper library that monitored the creation and deletion of user tasks.

```
#include “vmi/vmiOSLib.h”

static VMIOS_INTERCEPT_FN(doFib) {

    Uns32 index;

    getArg(processor, object, 0, &index);

    if(index>=30) {

        const char *context = vmirtGetProcessorScope(processor);

        // constrain intercept library to one scope
        if(!object->initialized) {
            object->initialized = True;
            vmiosUpdateScope(object, context);
            vmiMessage(
                "I", CPU_PREFIX "_SCOPE",
```

```
                "set intercept scope %s",
                vmiosGetScope(object)
            );
        }

        if(vmiosMatchScope(processor, object)) {

            // matching scope
            vmiMessage(
                "I", CPU_PREFIX "_MATCH",
                "(scope %s): index=%u",
                context,
                index
            );

        } else {

            // mismatched scope
            vmiMessage(
                "I", CPU_PREFIX "_IGNRE",
                "(scope %s ignored)",
                Context
            );
        }
    }
}
```

Notes and Restrictions

None.

6.2 vmiosGetScope

Prototype

```
const char *vmiosGetScope(vmiosObjectP object);
```

Description

This function returns the current effective scope of an intercept library or plugin. This is a string of the form:

```
clear
[!]mode:ASID ([and|or] [!]mode:ASID)*
```

(where `mode:ASID` follows the description given in section 6.1).

The value `clear` indicates that no scope constraint is active.

An exclamation mark in front of a scope implies negation: foe example, the value:

`"!3:12"`

indicates that the scope is active when a processor is not in mode 3 with ASID 12.

Multiple active scopes are shown concatenated by and and or primitives. Foe example, the value:

`"3:12 or 3:15"`

indicates that the scope is active when a processor is in mode 3 with ASID 12 or 15.

Example

This example shows how scopes might be used to constrain an intercept library callback that monitors a function in a Linux user application so that it is activated only when one particular process runs. The example is somewhat contrived because it constrains the intercept library so that it is tied to the *first user process that executes at the intercepted function address*, whether or not this is in fact running the target application. In reality, the scope constraint would be specified by an OS-aware helper library that monitored the creation and deletion of user tasks.

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

static VMIOS_INTERCEPT_FN(doFib) {

    Uns32 index;

    getArg(processor, object, 0, &index);

    if(index>=30) {

        const char *context = vmirtGetProcessorScope(processor);

        // constrain intercept library to one scope
        if(!object->initialized) {
            object->initialized = True;
            vmiosUpdateScope(object, context);
            vmiMessage(
                "I", CPU_PREFIX "_SCOPE",
```

```
                "set intercept scope %s",
                vmiosGetScope(object)
            );
        }

        if(vmiosMatchScope(processor, object)) {

            // matching scope
            vmiMessage(
                "I", CPU_PREFIX "_MATCH",
                "(scope %s): index=%u",
                context,
                index
            );

        } else {

            // mismatched scope
            vmiMessage(
                "I", CPU_PREFIX "_IGNRE",
                "(scope %s ignored)",
                Context
            );
        }
    }
}
```

Notes and Restrictions

None.

6.3 vmiosMatchScope

Prototype

```
Bool vmiosMatchScope(vmiProcessorP processor, vmiosObjectP object);
```

Description

This function returns a Boolean indicating whether the current processor scope matches the active intercept or plugin scope. This can be used to modify the control flow of the intercept library so that it is activated only when scopes match.

Example

This example shows how scopes might be used to constrain an intercept library callback that monitors a function in a Linux user application so that it is activated only when one particular process runs. The example is somewhat contrived because it constrains the intercept library so that it is tied to the *first user process that executes at the intercepted function address*, whether or not this is in fact running the target application. In reality, the scope constraint would be specified by an OS-aware helper library that monitored the creation and deletion of user tasks.

```
#include “vmi/vmiOSLib.h”

static VMIOS_INTERCEPT_FN(doFib) {

    Uns32 index;

    getArg(processor, object, 0, &index);

    if(index>=30) {

        const char *context = vmirtGetProcessorScope(processor);

        // constrain intercept library to one scope
        if(!object->initialized) {
            object->initialized = True;
            vmiosUpdateScope(object, context);
            vmiMessage(
                "I", CPU_PREFIX "_SCOPE",
                "set intercept scope %s",
                vmiosGetScope(object)
            );
        }

        if(vmiosMatchScope(processor, object)) {

            // matching scope
            vmiMessage(
                "I", CPU_PREFIX "_MATCH",
                "(scope %s): index=%u",
                context,
                index
            );

        } else {

            // mismatched scope
            vmiMessage(
                "I", CPU_PREFIX "_IGNRE",
                "(scope %s ignored)",
                Context
            );
```

```
        }
    }
}
```

Notes and Restrictions

None.

7 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.

7.1 vmiosAddCommand

Prototype

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

Description

This function adds a command with the specified `name` to the passed `vmiosObject`. When the command is executed, the function `commandCB` will be called. The argument `exampleArguments` is used by the help system and should list the arguments required by the command. Function commandCB should be defined using the `VMIOS_COMMAND_FN` macro (defined in file `vmiCommand.h`):

```
#define VMIOS_COMMAND_FN(_NAME) const char *_NAME( \
    vmiosObjectP object,    \
    Int32        argc,      \
    const char  *argv[]     \
)
```

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.

Argument `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.

7.2 vmiosAddCommandParse

Prototype

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

Description

This function adds a command with the specified `name` to the passed `vmiosObject`. When the command is called, the arguments will first be parsed according to the argument specifications provided by `vmirtAddArg`() and then the function `commandCB` will be called in the model. The argument `exampleArguments` 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*.

Argument `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

This example shows how to add a command `logtofile` with parsed arguments on, off and filename to a plugin. The command is created first and then arguments are added to it by repeatedly calling `vmirtAddArg`.

```
void constructParser(vmiosObjectP object, cmdArgValuesP argValues) {

    vmiCommandP cmd;

    cmd = vmiosAddCommandParse(
        object,
        "logtofile",
        "Enable/disable writing messages from this library to its own log file",
        logtofileCB,
        VMI_CT_QUERY|VMI_CO_CPU|VMI_CA_TRACE
    );

    vmirtAddArg(
        cmd,
        "on",
        "Turns on trace output to the log file (default)",
        VMI_CA_FLAG,
        VMI_CAA_MENU,
        0,
        &(argValues->logtofileStr.on)
    );

    vmirtAddArg(
```

```
        cmd,
        "off",
        "Turns off trace output to the log file (initial)",
        VMI_CA_FLAG,
        VMI_CAA_DEFAULT,
        0,
        &(argValues->logtofileStr.off)
    );

    vmirtAddArg(
        cmd,
        "filename",
        "Filename for logfile for this library (default is name based on library
instance name)",
        VMI_CA_STRING,
        VMI_CAA_MENU,
        0,
        &(argValues->logtofileStr.filename)
    );
}
```

Notes and Restrictions

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