OVP VMI Memory Model Component Function Reference

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

Author:	Imperas Software Limited
Version:	6.9.1
Filename:	OVP_VMI_Memory_Model_Component_Function_Reference.doc
Project:	OVP VMI MMC Reference
Last Saved:	February 19, 2019
Keywords:	MMC

Table of Contents

1 Introduction

This is reference documentation for **version 6.9.1** of the MMC function interface, defined in `ImpPublic/include/host/vmi/vmiMmc.h`.

The functions in this interface are used to model *memory model components*, such as instruction and data caches, that supplement Imperas processor models.

There are two distinct kinds of memory model component model: *full* and *transparent*. Full models implement storage and so can be used to accurately model components such as caches that are incoherent with main memory. Transparent models do not implement storage (so cannot be incoherent) but can be used to create very fast performance monitors. As an example, a transparent cache model would model only the cache tags and use this information to count hits and misses.

Functions in this interface have the prefix `vmimmc`.

1.1 Transparent and Full MMC models

MMCs operate in two modes:
In Full Mode the model must implement read and write requests delivered to it. Transactions can be satisfied locally (to simulate a cache-hit for example) or routed to other models (to simulate a cache-miss). Since the model is simulating the storage and recovery of data, cache coherency can be modeled in this mode.

Full mode has a detrimental effect on simulator performance especially when used with very high speed processor models.

In Transparent Mode, the simulator automatically routes the read and write requests around the model to whatever is connected beyond it. The model is notified of every access by a callback function. The function is used to record data but does not implement the reads and writes. Since data is not read or written by the model (other than for performance analysis), cache coherency cannot be modeled in this mode, so cache behavior will appear full coherent.

Transparent Mode has less effect on simulation performance.

The simulator chooses the mode of operation. It is recommended that both full and transparent modes are implemented so that a model can be used in both cases.

2 Example

A simple module that uses an MMC to model a cache is in

```
$IMPERAS_HOME/Examples/PlatformConstruction/fullMMC.
```

The module uses an OVP model from

```
$IMPERAS_HOME/ImperasLib/source/ovpworld.org/mmc/wb_1way_32byteline_2048tags/1.0
```

The source of this model is in

```
$IMPERAS_HOME/ImperasLib/source/ovpworld.org/mmc/support/1.0/include/cache.h
```

Note: Run-time tests in MMCs that would be required if the model were written to have its configuration specified at run time would significantly affect the performance of the model. Therefore this include file is parameterized to construct separate cache models that are configured at compile time, thus saving the overhead of configuration computations at run time.

2.1 Model attributes table

An MMC model defines a table that contains the addresses of entry functions required by the simulator. The table is defined by the type `vmimmcAttrs in vmimmcAttrs.h` and must be called `modelattrs.`

```
vmimmcAttr modelAttrs = {

    .versionString       = VMI_VERSION,
    .modelType           = VMI_MMC_MODEL,
    .componentSize       = sizeof(cacheObject),

    .constructorCB       = cacheConstructor,
    .linkCB              = cacheLink,
    .destructorCB        = cacheDestructor,

    .refreshCB           = cacheRefresh,

    .readNFullCB         = readNFull,
    .writeNFullCB        = writeNFull,

    .readNTransparentCB  = readNTransparent,
    .writeNTransparentCB = writeNTransparent,

    .paramSpecsCB        = paramSpecs,
    .paramValueSizeCB    = paramValueSize,

    .busPortSpecsCB      = nextBusPortSpec
};
```

The fields must be initialized as follows:

2.2 Construction and specification

`.versionString`
Must be set to the macro `VMI_VERSION` defined in `vmiVersion.h`

`.modelType`
Must be set to `VMI_MMC_MODEL` defined in `vmiTypes.h`

`.componentSize`
Must be set to the total size (in bytes) of the data object defined by your model. In the example `cache.h`, data is stored in the structure `cacheObject`.

`.constructorCB`
Use the macro `VMIMMC_CONSTRUCTOR_FN` to define your constructor function. This is used to construct and initialize data associated with the MMC instance.

`.destructorCB`
Use the macro `VMIMMC_DESTRUCTOR_FN` to define your destructor function. This is used to free any resources allocated by your model, and to report statistics and other information to the use at the end of simulation.

`.linkCB`
This function is used when this is a transparent cache, to link to another cascaded cache.

`.paramSpecsCB`
This function is used by the simulator to get a list of formal parameters read by this model. Parameters are set in the platform and can be read by the model to change its behavior. Parameter usage is described in more detail in the next section.

`.paramValueSizeCB`
This function is used by the simulator to get the size of a parameter block needed to hold parameters for the MMC model. Parameter usage is described in more detail in the next section.

`.busPortSpecsCB`
This function is used by the simulator to get a list of bus ports used by the model.

2.3 Refresh

`.refreshCB`
This function is called before an MMC model becomes active (with state `RS_RUN`) and before an MMC model becomes inactive (with state `RS_SUSPEND`). The purpose of the callback is to allow the MMC model to update its state to take account of changes in the platform while it has been inactive (if `RS_RUN`) or to propagate state changes from the model to the wider platform (if `RS_SUSPEND`).

2.4 Read and Write callbacks – full MMC mode

`.readNFullCB`

This function is called to implement the reading of data by the bus master connected to this model. It can supply the data from a local source or might choose to read data from another source via its bus slave port.

`.writeNFullCB`

This function is called to implement the writing of data by the bus master. It can store the data locally or might choose to write it to another destination via a master port.

2.5 Read and Write callbacks – transparent MMC mode

`.readNTransparentCB`
`.writeNTransparentCB`

These functions do not implement the read and write callbacks – the simulator reads and writes data automatically from whatever is connected to the model's slave ports. Instead these called are used to record whatever information is required by the transparent model.

3 Model Configuration

A model can have optional features that can be configured by the platform during construction. Configuration is controlled by parameters which form part of the model's interface.

3.1 Model Parameters

Parameters are specified to the simulator by an iterator function and a size function specified in the model's attributes table. A parameter specification specifies the data type and bounding conditions of the parameter so the model does not need to check for trivial errors. The model must define a structure which contains value fields for each parameter. It should use the provided macros of the form `VMI_<type>_PARAM`, which reserve space for the value and for a Boolean which is true if the parameter has been set by the platform, false otherwise.

The supported parameter types are described below:

macro	data type
`VMI_BOOL_PARAM`	boolean
`VMI_INT32_PARAM`	32 bit signed
`VMI_INT64_PARAM`	64 bit signed
`VMI_UNS32_PARAM`	32 bit unsigned
`VMI_UNS64_PARAM`	64 bit unsigned
`VMI_DBL_PARAM`	floating point
`VMI_STRING_PARAM`	0 terminated string
`VMI_ENUM_PARAM`	0 terminated string
`VMI_ENDIAN_PARAM`	0 terminated string
`VMI_PTR_PARAM`	native host pointer

During initialization, the simulator uses the iterator function to get the list of parameters for the model. Then it allocates the model's parameter structure (using the size function) and fills in the correct values. This structure is then passed to the model's constructor where the model can use the values.

3.2 Parameter Specification

The parameter specification structure is defined in `vmiParameters.h` and should be initialized using these macros:

macro	data type	limits
`VMI_BOOL_PARAM_SPEC`	boolean	0 or 1
`VMI_INT32_PARAM_SPEC`	32 bit signed	specified min / max
`VMI_INT64_PARAM_SPEC`	64 bit signed	specified min / max
`VMI_UNS32_PARAM_SPEC`	32 bit unsigned	specified min / max
`VMI_UNS64_PARAM_SPEC`	64 bit unsigned	specified min / max
`VMI_DBL_PARAM_SPEC`	floating point	specified min / max
`VMI_STRING_PARAM_SPEC`	0 terminated string	any string (or 0 if not specified)

VMI_ENUM_PARAM_SPEC	0 terminated string	string must be a member of the specified list
VMI_ENDIAN_PARAM_SPEC	0 terminated string	"big" or "little"
VMI_PTR_PARAM_SPEC	native host pointer	none

The iterator function must be supplied by the model and should use the provided macro from `vmiMmcAttrs.h`:

Prototype

```
#define VMIMMC_PARAM_SPECS_FN(_NAME) vmiParameterP _NAME ( \
    vmimmcComponentP component, \
    vmiParameterP    prev       \
)
```

It should return the first or subsequent parameter specification or 0 if at the end of the list. Note that the iterator is also supplied with the MMC component pointer, so can include or exclude parameters in an instance-specific way if required.

Example

```
// VMI header files
#include "vmi/vmiMmcAttrs.h"
#include "vmi/vmParameters.h"

//
// Define the parameter structure
//
typedef struct paramValuesS {

    VMI_BOOL_PARAM(verbose);           // boolean parameter
    VMI_UNS32_PARAM(numSlavePorts);    // 32 bit unsigned parameter

} pVals, *pValsP;

//
// Define the parameters
//
static vmiParameter formals[] = {

    VMI_BOOL_PARAM_SPEC(  pVals, verbose,       0,        "Enable text output"),
    VMI_UNS32_PARAM_SPEC( pVals, numSlavePorts, 1, 1, 8,  "Slave port number"),

    // Add entry with name==NULL to terminate list
    VMI_END_PARAM
};

//
// Function to iterate the parameter specs
//
VMI_PROC_PARAM_SPECS_FN(getParamSpec) {
    if(!prev) {
        return formals;
    } else {
        prev++;
        if (prev->name)
            return prev;
        else
            return 0;
    }
}

//
// Get the size of the parameter values table
//
VMI_PROC_PARAM_TABLE_SIZE_FN(paramValueSize) {
    return sizeof(pVals);
```

```
}

//
// model constructor
//
VMIMMC_CONSTRUCTOR_FN(modelConstructor) {

    myMMCP myMMC  = (myMMCP)component;
    pValsP params = (pValsP)parameterValues; // cast to my type

    // use the parameter values
    myMMC->numSlavePorts = params->numSlavePorts;

    if (params->verbose) {
       vmiPrintf(....);
    }
    ...
}

//
// Add functions to the model attributes table
//
const vmiMMCAttr modelAttrs = {
    ...
    .constructorCB    = modelConstructor,
    .paramSpecsCB     = getParamSpec,
    .paramValueSizeCB = paramValueSize,
    ...
};
```

Restrictions

The parameter structure exists only for the life of the constructor function.

4 Functions

4.1 vmimmcGetName

Prototype

```
const char *vmimmcGetName(vmimmcComponentP component);
```

Description

This function returns the name of an MMC component. Typically this will be used in debug messages or in summary messages.

The name returned is that of the component only, as defined in the platform file. The related function `vmimmcGetHierarchicalName` returns the name including the full instantiation path.

Example

```
#include “vmi/vmiMmc.h”
#include “vmi/vmiMmcAttrs.h”
#include “vmi/vmiMessage.h”

// Cache object destructor
static VMIMMC_DESTRUCTOR_FN(cacheDestructor) {

    vmiPrintf(
        "\n%s called for %s\n",
        FUNC_NAME,
        vmimmcGetName(component)
    );

    cacheObjectP cache = (cacheObjectP)component;

    debugDestructor(cache);
}
```

Notes and Restrictions

None.

4.2 vmimmcGetHierarchicalName

Prototype

```
const char *vmimmcGetHierarchicalName(vmimmcComponentP component);
```

Description

This function returns the name of a memory model component, including the full instantiation path, as defined in the platform file.

The related function `vmimmcGetName` returns the component base name omitting the instantiation path.

Example

```
#include “vmi/vmiMmc.h”
#include “vmi/vmiMmcAttrs.h”
#include “vmi/vmiMessage.h”

// Cache object destructor
static VMIMMC_DESTRUCTOR_FN(cacheDestructor) {

    vmiPrintf(
        "\n%s called for %s\n",
        FUNC_NAME,
        vmimmcGetHierarchicalName(component)
    );

    cacheObjectP cache = (cacheObjectP)component;

    debugDestructor(cache);
}
```

Notes and Restrictions

None.

4.3 vmimmcGetNextPort

Prototype

```
vmimmcPortP vmimmcGetNextPort(
    vmimmcComponentP component,
    const char      *portName
);
```

Description

Given an MMC component and an output port name for that component, this function returns the input port of any subsequent MMC component connected to that port.

Typically, this is used in combination with `vmimmcGetPortAttrs` to establish connectivity in the `VMIMMC_LINK_FN` callback, as shown in the following example.

This function returns non-NULL only if the current MMC component is *transparent*. If the component implements a full MMC model, this function will return NULL, but `vmimmcGetNextDomain` will return the connected domain.

Example

```
// Cache object
typedef struct cacheObjectS {
                                                // MODELING ARTIFACTS
    vmimmcPortP     nextPort;                   // next port (TRANSPARENT)
    memDomainP      nextDomain;                 // next domain (FULL)
    memRegionP      lastRegion;                 // last accessed (FULL)
    Uns32           mruKey;                     // access optimization
    cacheLineP      mruLine;                    // access optimization
    cacheAccessInfo readInfo;                   // read access recording
    cacheAccessInfo writeInfo;                  // write access recording

                                                // TRUE CACHE CONTENTS
    Uns32      keys[CACHE_TAGS][CACHE_WAYS];    // set of keys for cache
    cacheLineP index[CACHE_TAGS][CACHE_WAYS];   // index into cache lines
    cacheLine  lines[CACHE_TAGS][CACHE_WAYS];   // set of lines for cache

} cacheObject, *cacheObjectP;

// Cache object link
static VMIMMC_LINK_FN(cacheLink) {

    vmiPrintf(
        "\n%s called for %s\n",
        FUNC_NAME,
        vmimmcGetHierarchicalName(component)
    );

    cacheObjectP cache      = (cacheObjectP)component;
    vmimmcPortP  nextPort   = vmimmcGetNextPort(component, "mp1");
    memDomainP   nextDomain = vmimmcGetNextDomain(component, "mp1");

    // sanity check that we know whether we are in transparent or full
    // mode
    VMI_ASSERT(
        !(nextPort && nextDomain),
        "%s: expected either nextPort (transparent) "
        "or nextDomain (full), not both",
        FUNC_NAME
    );
```

```
    // set the next connected MMC model port
    cache->nextPort   = nextPort;
    cache->nextDomain = nextDomain;

    if(nextPort) {

        vmimmcAttrCP attrs = vmimmcGetPortAttrs(nextPort);

        // set transparent functions to call on a miss
        if(attrs) {
            cache->readInfo.missCB  = attrs->readNTransparentCB;
            cache->writeInfo.missCB = attrs->writeNTransparentCB;
        }
    }
}
```

Notes and Restrictions

None.

4.4 vmimmcGetNextDomain

Prototype

```
memDomainP vmimmcGetNextDomain(
    vmimmcComponentP component,
    const char      *portName
);
```

Description

Given an MMC component and an output port name for that component, this function returns any memory domain object connected to that port.

This function returns NULL if the current MMC component is *transparent*. In this case, `vmimmcGetNextPort` should be used to obtain information about the next port.

The domain is typically used in full model callback functions. As an example, a model of a cache may need to call `vmirtReadNByteDomain` or `vmirtWriteNByteDomain` in the case of a cache miss.

Example

```
// Cache object
typedef struct cacheObjectS {
                                                // MODELING ARTIFACTS
    vmimmcPortP     nextPort;                   // next port (TRANSPARENT)
    memDomainP      nextDomain;                 // next domain (FULL)
    memRegionP      lastRegion;                 // last accessed (FULL)
    Uns32           mruKey;                     // access optimization
    cacheLineP      mruLine;                    // access optimization
    cacheAccessInfo readInfo;                   // read access recording
    cacheAccessInfo writeInfo;                  // write access recording

                                                // TRUE CACHE CONTENTS
    Uns32      keys[CACHE_TAGS][CACHE_WAYS];    // set of keys for cache
    cacheLineP index[CACHE_TAGS][CACHE_WAYS];   // index into cache lines
    cacheLine  lines[CACHE_TAGS][CACHE_WAYS];   // set of lines for cache

} cacheObject, *cacheObjectP;

// Cache object link
static VMIMMC_LINK_FN(cacheLink) {

    vmiPrintf(
        "\n%s called for %s\n",
        FUNC_NAME,
        vmimmcGetHierarchicalName(component)
    );

    cacheObjectP cache      = (cacheObjectP)component;
    vmimmcPortP  nextPort   = vmimmcGetNextPort(component, "mp1");
    memDomainP   nextDomain = vmimmcGetNextDomain(component, "mp1");

    // sanity check that we know whether we are in transparent or full
    // mode
    VMI_ASSERT(
        !(nextPort && nextDomain),
        "%s: expected either nextPort (transparent) "
        "or nextDomain (full), not both",
        FUNC_NAME
    );
```

```
    // set the next connected MMC model port
    cache->nextPort   = nextPort;
    cache->nextDomain = nextDomain;

    if(nextPort) {

        vmimmcAttrCP attrs = vmimmcGetPortAttrs(nextPort);

        // set transparent functions to call on a miss
        if(attrs) {
            cache->readInfo.missCB  = attrs->readNTransparentCB;
            cache->writeInfo.missCB = attrs->writeNTransparentCB;
        }
    }
}
```

Notes and Restrictions

None.

4.5 vmimmcGetPortAttrs

Prototype

```
vmimmcAttrCP vmimmcGetPortAttrs(vmimmcPortP port);
```

Description

Given an MMC input port, this function returns the attribute structure associated with that port. The attribute structure defines the behavior of the input port (for example, callback functions to be activated when data is read or written).

Typically, this is used in combination with `vmimmcGetNextPort` to establish connectivity in the `VMIMMC_LINK_FN` callback, as shown in the following example.

Example

```
// Cache object
typedef struct cacheObjectS {
                                                // MODELING ARTIFACTS
    vmimmcPortP     nextPort;                   // next port (TRANSPARENT)
    memDomainP      nextDomain;                 // next domain (FULL)
    memRegionP      lastRegion;                 // last accessed (FULL)
    Uns32           mruKey;                     // access optimization
    cacheLineP      mruLine;                    // access optimization
    cacheAccessInfo readInfo;                   // read access recording
    cacheAccessInfo writeInfo;                  // write access recording

                                                // TRUE CACHE CONTENTS
    Uns32      keys[CACHE_TAGS][CACHE_WAYS];    // set of keys for cache
    cacheLineP index[CACHE_TAGS][CACHE_WAYS];   // index into cache lines
    cacheLine  lines[CACHE_TAGS][CACHE_WAYS];   // set of lines for cache

} cacheObject, *cacheObjectP;

// Cache object link
static VMIMMC_LINK_FN(cacheLink) {

    vmiPrintf(
        "\n%s called for %s\n",
        FUNC_NAME,
        vmimmcGetHierarchicalName(component)
    );

    cacheObjectP cache      = (cacheObjectP)component;
    vmimmcPortP  nextPort   = vmimmcGetNextPort(component, "mp1");
    memDomainP   nextDomain = vmimmcGetNextDomain(component, "mp1");

    // sanity check that we know whether we are in transparent or full
    // mode
    VMI_ASSERT(
        !(nextPort && nextDomain),
        "%s: expected either nextPort (transparent) “
        “or nextDomain (full), not both",
        FUNC_NAME
    );

    // set the next connected MMC model port
    cache->nextPort   = nextPort;
    cache->nextDomain = nextDomain;

    if(nextPort) {

        vmimmcAttrCP attrs = vmimmcGetPortAttrs(nextPort);
```

```
        // set transparent functions to call on a miss
        if(attrs) {
            cache->readInfo.missCB  = attrs->readNTransparentCB;
            cache->writeInfo.missCB = attrs->writeNTransparentCB;
        }
    }
}
```

Notes and Restrictions

None.

4.6 vmimmcRefreshTransparent

Prototype

```
vmimmcAttrCP vmimmcRefreshTransparent(
    vmimmcPortP    port,
    vmiIASRunState state
);
```

Description

When an MMC's refresh callback is called, the model must refresh any cascaded MMCs.

Typically, this is used in combination with vmimmcRefreshFull so that either transparent or full models are refreshed.

Example

```
// Cache refresh callback

static VMIMMC_REFRESH_FN(cacheRefresh) {
    // get cascaded models (which must be held on the model object).
    cacheObjectP cache      = (cacheObjectP)component;
    vmimmcPortP  nextPort   = cache->nextPort;
    memDomainP   nextDomain = cache->nextDomain;

    if(nextPort) {
        vmimmcRefreshTransparent(nextPort, state);
    else if (nextDomain) {
        vmimmcRefreshFull(nextDomain, state);
    } else {
        // there are no cascaded components
    }

    // now perform local refresh actions.
}

vmimmcAttr modelAttrs = {
    ...
    cacheDestructor,
    cacheRefresh,       // install the refresh callback
    readNFull,
    ...
};
```

Notes and Restrictions

The refresh callback is updated at the start of each time-slice. It is typically used in a full cache model to synchronize its state with any changes to the memory it is caching that were caused by other processors.

4.7 vmimmcRefreshFull

Prototype

```
vmimmcAttrCP vmimmcRefreshFull(
    vmimmcPortP port,
    memDomainP  nextDomain
);
```

Description

When an MMC's refresh callback is called, the model must refresh any cascaded MMCs.

Typically, this is used in combination with vmimmcRefreshTransparent so that either transparent or full models are refreshed.

Example

```
// Cache refresh callback

static VMIMMC_REFRESH_FN(cacheRefresh) {
    // get cascaded models (which must be held on the model object).
    cacheObjectP cache      = (cacheObjectP)component;
    vmimmcPortP  nextPort   = cache->nextPort;
    memDomainP   nextDomain = cache->nextDomain;

    if(nextPort) {
        vmimmcRefreshTransparent(nextPort, state);
    else if (nextDomain) {
        vmimmcRefreshFull(nextDomain, state);
    } else {
        // there are no cascaded components
    }

    // now perform local refresh actions.
}

vmimmcAttr modelAttrs = {
    ...
    cacheDestructor,
    cacheRefresh,       // install the refresh callback
    readNFull,
    ...
};
```

Notes and Restrictions

The refresh callback is updated at the start of each time-slice. It is typically used in a full cache model to synchronize its state with any changes to the memory it is caching that were caused by other processors.

4.8 vmimmcAddCommand

Prototype

```
void vmimmcAddCommand(
    vmimmcComponentP component,
    const char       *name,
    const char       *exampleArguments,
    vmimmcCommandFn  commandCB
);
```

Description

This function adds a command that can be called from the simulator. The command is typically used to change the mode of the MMC or to report something about its internal state.

Please use *vmimmcAddCommandParse* in preference to this function.

Example

```
static VMIMMC_COMMAND_FN(enableCallback) {
    if (argc == 2) {
        const char *firstArg = argv[1];
        ... // use the argument
        return "OK";
    } else {
        vmiPrintf("Error calling command '%s'", argv[0]);
        return "";
    }
}

// Cache constructor callback

static VMIMMC_REFRESH_FN(cacheconstructor) {
    ...
    vmimmcAddCommand(
        component,
        "enable",
        "-on | -off"    // this is used by the help system
        enableCallback
    );
```

Notes and Restrictions

The string returned by the command callback is passed to the tcl interpreter (if an interpreter is active).

4.9 *vmimmcAddCommandParse*

Prototype

```
void vmimmcAddCommandParse(
    vmimmcComponentP     component,
    const char          *name,
    const char*          help,
    vmimmcCommandParseFn commandCB,
    vmiCommandAttrs      attrs
);
```

Description

This function adds a command that can be called from the simulator. The command is typically used to change the mode of the MMC or to report something about its internal state.

Example

```
static VMIMMC_COMMAND_PARSE_FN(enableCallback) {
    cacheObjectP cache = (cacheObjectP) component;

    cache->enabled = True;

    return “”;
}

// Cache constructor callback

static VMIMMC_REFRESH_FN(cacheconstructor) {
    ...
    vmimmcAddCommandParse(
        component,
        "enable",
        "Enable the cache",
        enableCallback,
        VMI_CT_MODE|VMI_CO_CACHE|VMI_CA_CONTROL
    );
```

Notes and Restrictions

The string returned by the command callback is passed to the tcl interpreter (if an interpreter is active).

5 Model Design Considerations

This section is intended to provide some Questions and Answer that may aid in the efficient design and use of the MMC as a modeling component.

5.1 What is the difference between Transparent and Full

In transparent mode the data transfers are not modified, the MMC is called back to say what has happened but transfers are direct between processor and other models and memory i.e. there is no change.

In full mode the MMC is providing cached data/instructions, it is the responsibility of the cache model to correctly fill and provide the data to the processor model. Any errors in the coding of the cache model could cause issues with program execution.

5.2 What limitations are there using an MMC for cache modeling

When using an MMC as a cache model you do not get the action of the cache instructions, such as flush etc, applied to the cache memory. This can also have an effect on the program execution depending how the application code is written.

