OVP VMI Run Time Function Reference

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

Author:	Imperas Software Limited
Version:	6.14.0
Filename:	OVP_VMI_Run_Time_Function_Reference.doc
Project:	OVP VMI Run Time Function Reference
Last Saved:	Thursday, 14 May 2015
Keywords:	

Copyright Notice

Right to Copy Documentation

Destination Control Statement

Disclaimer

Table of Contents

1 Introduction 9

2 QuantumLeap Semantics 10

3 Model Configuration 11

3.1 MODEL PARAMETERS 12
3.2 PARAMETER SPECIFICATION 13

4 Model Hardware Interface 16

4.1 BUS PORT SPECIFICATION 17
4.2 VMIRTGETBUSPORTDOMAIN 20
4.3 NET PORT SPECIFICATION 21
4.4 CONNECTION PORT SPECIFICATION 23

5 Simulator Control 26

5.1 VMIRTYIELD 27
5.2 VMIRTHALT 28
5.3 VMIRTINTERRUPT 29
5.4 VMIRTEXIT 30
5.5 VMIRTFINISH 31
5.6 VMIRTSTOP 32
5.7 VMIRTATOMIC 33
5.8 VMIRTBLOCK 34
5.9 VMIRTISHALTED 35
5.10 VMIRTRESTARTNEXT 36
5.11 VMIRTRESTARTNOW 37
5.12 VMIRTTRACEONAFTER 38
5.13 VMIRTTRACEOFFAFTER 39
5.14 VMIRTFLUSHTARGET 40
5.15 VMIRTFLUSHTARGETMODE 41
5.16 VMIRTFLUSHTARGETMODETAGGED 43
5.17 VMIRTFLUSHDICT 45
5.18 VMIRTFLUSHALLDICTS 46
5.19 VMIRTGETNETPORTHANDLE 47
5.20 VMIRTWRITENETPORT 48
5.21 VMIRTREADNETPORT 49
5.22 VMIRTINSTALLNETCALLBACK 50
5.23 VMIRTDOSYNCHRONOUSINTERRUPT 51
5.24 VMIRTCACHEREGISTER 53

6 Instruction Counting 56

6.1 VMIRTGETPROCESSORIPS 57
6.2 VMIRTGETICOUNT 58
6.3 VMIRTSETICOUNTINTERRUPT 59
6.4 VMIRTCLEARICOUNTINTERRUPT 61
6.5 VMIRTCREATEMODELTIMER 62
6.6 VMIRTCREATEMONOTONICMODELTIMER 65
6.7 VMIRTDELETEMODELTIMER 67
6.8 VMIRTSETMODELTIMER 68
6.9 VMIRTCLEARMODELTIMER 70
6.10 VMIRTISMODELTIMERENABLED 71
6.11 VMIRTGETMODELTIMERCURRENTCOUNT 72
6.12 VMIRTGETMODELTIMEREXPIRYCOUNT 74

7 Simulated Memory Access 75

7.1 vmirtReadNByteDomain 76
7.2 vmirtReadNByteDomainVM 79
7.3 vmirtWriteNByteDomain 81
7.4 vmirtWriteNByteDomainVM 84
7.5 vmirtRead[1248]ByteDomain 85
7.6 vmirtWrite[1248]ByteDomain 87
7.7 vmirtGetReadNByteSrc 89
7.8 vmirtGetWriteNByteDst 91
7.9 vmirtGetString 93

8 Simulated Memory Management 95

8.1 vmirtGetProcessorCodeDomain 99
8.2 vmirtGetProcessorDataDomain 100
8.3 vmirtGetProcessorPhysicalCodeDomain 101
8.4 vmirtGetProcessorPhysicalDataDomain 102
8.5 vmirtGetProcessorInternalCodeDomain 103
8.6 vmirtGetProcessorInternalDataDomain 105
8.7 vmirtSetProcessorInternalCodeDomain 107
8.8 vmirtSetProcessorInternalDataDomain 109
8.9 vmirtSetProcessorCodeDomain 111
8.10 vmirtSetProcessorDataDomain 112
8.11 vmirtSetProcessorCodeDomains 115
8.12 vmirtSetProcessorDataDomains 117
8.13 vmirtGetProcessorCodeEndian 119
8.14 vmirtGetProcessorDataEndian 120
8.15 vmirtNewDomain 121
8.16 vmirtAliasMemory 123
8.17 vmirtUnaliasMemory 125
8.18 vmirtProtectMemory 126
8.19 vmirtGetDomainAddressBits 128
8.20 vmirtGetDomainPrivileges 129
8.21 vmirtIsExecutable 131
8.22 vmirtGetDomainMapped 133
8.23 vmirtMapVAToPA 134
8.24 vmirtMapMemory 135
8.25 vmirtMapCallbacks 136
8.26 vmirtMapNativeMemory 139
8.27 vmirtAddReadCallback 141
8.28 vmirtAddWriteCallback 143
8.29 vmirtAddFetchCallback 145
8.30 vmirtRemoveReadCallback 147
8.31 vmirtRemoveWriteCallback 149
8.32 vmirtRemoveFetchCallback 151
8.33 vmirtAliasMemoryVM 153
8.34 vmirtUnaliasMemoryVM 156
8.35 vmirtGetDomainMappedASID 158
8.36 vmirtSetProcessorASID 160
8.37 vmirtGetProcessorASID 161
8.38 vmirtGetMRUStateTable 162
8.39 vmirtGetNthStateIndex 163
8.40 vmirtAddPCCallback 164
8.41 vmirtRemovePCCallback 166

9 Accessing and Setting the Program Counter 168

9.1 vmirtGetPC ... 169
9.2 vmirtGetPCDS ... 170
9.3 vmirtSetPC ... 172
9.4 vmirtSetPCDS ... 173
9.5 vmirtSetPCException ... 174
9.6 vmirtSetPCFlushTarget ... 175
9.7 vmirtSetPCFlushDict ... 177

10 Processor Modes ... 178

10.1 vmirtSetMode ... 179
10.2 vmirtGetMode ... 181
10.3 vmirtSetBlockMask ... 182
10.4 vmirtGetBlockMask ... 185

11 Floating Point Operations ... 186

11.1 vmirtConfigureFPU ... 187
11.2 vmirtGetFPControlWord ... 191
11.3 vmirtSetFPControlWord ... 193
11.4 vmirtRestoreFPState ... 195

12 Connection Operations ... 196

12.1 vmirtConnGetInput ... 197
12.2 vmirtConnGetOutput ... 198
12.3 vmirtConnGetInputInfo ... 199
12.4 vmirtConnGetOutputInfo ... 200
12.5 vmirtConnGet ... 201
12.6 vmirtConnPut ... 203
12.7 vmirtConnNotifyGet ... 205
12.8 vmirtConnNotifyPut ... 206

13 Simulation Environment Access ... 207

13.1 vmirtGetCurrentProcessor ... 208
13.2 vmirtCPUId ... 209
13.3 vmirtGetProcessorForCPUId ... 210
13.4 vmirtProcessorFlags ... 211
13.5 vmirtProcessorName ... 212
13.6 vmirtSetProcessorName ... 213
13.7 vmirtProcessorType ... 214
13.8 vmirtProcessorStringAttribute ... 215
13.9 vmirtProcessorBoolAttribute ... 217
13.10 vmirtProcessorUns32Attribute ... 218
13.11 vmirtProcessorUns64Attribute ... 219
13.12 vmirtProcessorFlt64Attribute ... 221

14 SMP Processor Functions ... 222

14.1 SMP Processor Hierarchies ... 223
14.2 Specifying SMP Attributes in the Processor Constructor ... 224
14.3 vmirtGetSMPParent ... 228
14.4 vmirtSetSMPParent ... 229
14.5 vmirtGetSMPChild ... 231
14.6 vmirtGetSMPPrevSibling ... 232
14.7 vmirtGetSMPNextSibling ... 233
14.8 vmirtGetSMPActiveSibling ... 234
14.9 vmirtGetSMPIndex ... 235
14.10 vmirtSetSMPIndex ... 236
14.11 vmirtGetSMPCpuType ... 237

14.12 vmirtIterAllChildren ... 239
14.13 vmirtIterAllDescendants ... 240
14.14 vmirtIterAllProcessors ... 242

15 Communication Between Objects ... 244

15.1 vmirtFindAddSharedData ... 245
15.2 vmirtFindSharedData ... 246
15.3 vmirtGetSharedDataValue ... 247
15.4 vmirtSetSharedDataValue ... 248
15.5 vmirtRemoveSharedData ... 249
15.6 vmirtWriteListeners ... 250
15.7 vmirtRegisterListener ... 251
15.8 vmirtUnregisterListener ... 253

16 Address Range Hash Utilities ... 254

16.1 vmirtNewRangeTable ... 255
16.2 vmirtFreeRangeTable ... 256
16.3 vmirtInsertRangeEntry ... 257
16.4 vmirtRemoveRangeEntry ... 258
16.5 vmirtGetFirstRangeEntry ... 259
16.6 vmirtGetNextRangeEntry ... 260
16.7 vmirtGetRangeEntryLow ... 261
16.8 vmirtGetRangeEntryHigh ... 262
16.9 vmirtGetRangeEntryUserData ... 263
16.10 vmirtSetRangeEntryUserData ... 264

17 Application Program Symbol Table Access ... 265

17.1 vmirtAddressLookup ... 266
17.2 vmirtSymbolLookup ... 268
17.3 vmirtAddSymbolFile ... 269
17.4 vmirtNextSymbolFile ... 270
17.5 vmirtGetSymbolFileName ... 272
17.6 vmirtGetSymbolByName ... 273
17.7 vmirtGetSymbolByAddr ... 274
17.8 vmirtNextSymbolByName ... 275
17.9 vmirtNextSymbolByAddr ... 277
17.10 vmirtPrevSymbolByAddr ... 279
17.11 vmirtNextSymbolByNameFile ... 280
17.12 vmirtNextSymbolByAddrFile ... 282
17.13 vmirtPrevSymbolByAddrFile ... 284
17.14 vmirtGetSymbolName ... 286
17.15 vmirtGetSymbolAddr ... 287
17.16 vmirtGetSymbolLoadAddr ... 288
17.17 vmirtGetSymbolType ... 289
17.18 vmirtGetSymbolBinding ... 290
17.19 vmirtGetSymbolSize ... 291

18 Application Dwarf Line Number Information Access ... 292

18.1 vmirtGetFLByAddr ... 293
18.2 vmirtNextFLByAddr ... 294
18.3 vmirtPrevFLByAddr ... 295
18.4 vmirtNextFLByAddrFile ... 296
18.5 vmirtPrevFLByAddrFile ... 297
18.6 vmirtGetFLFileName ... 298
18.7 vmirtGetFLLineNumber ... 299
18.8 vmirtGetFLAddr ... 300

19 Licensing 301

19.1 vmirtGetLicense 302
19.2 vmirtGetLicenseErrString 303

20 View Provider Interface 304

20.1 vmirtGetProcessorViewObject 305
20.2 vmirtAddViewObject 306
20.3 vmirtSetViewObjectConstValue 307
20.4 vmirtSetViewObjectRefValue 308
20.5 vmirtSetViewObjectValueCallback 309
20.6 vmirtAddViewAction 310
20.7 vmirtAddViewEvent 311
20.8 vmirtNextViewEvent 312
20.9 vmirtTriggerViewEvent 313
20.10 vmirtDeleteViewObject 314

21 Runtime Commands 315

21.1 vmirtAddCommand 316
21.2 vmirtAddCommandParse 317
21.3 vmirtAddArg 321
21.4 Command attributes 323

22 Processor Register, Mode, Exception and Port Access Utilities 324

22.1 vmirtGetRegGroupByName 325
22.2 vmirtGetNextRegGroup 326
22.3 vmirtGetRegByName 327
22.4 vmirtGetNextReg 328
22.5 vmirtGetNextRegInGroup 329
22.6 vmirtRegRead 330
22.7 vmirtRegWrite 331
22.8 vmirtGetCurrentMode 332
22.9 vmirtGetNextMode 333
22.10 vmirtGetCurrentException 334
22.11 vmirtGetNextException 335
22.12 vmirtGetBusPortByName 336
22.13 vmirtGetNextBusPort 337
22.14 vmirtGetNetPortByName 338
22.15 vmirtGetNextNetPort 339
22.16 vmirtGetFifoPortByName 340
22.17 vmirtGetNextFifoPort 341

23 Semihost and Debugger Integration Support Functions 342

23.1 vmirtDisassemble 343

24 Debugger Integration Support Functions 346

24.1 vmirtEvaluateGDBExpression 347
24.2 vmirtEvaluateCodeLocation 348
24.3 vmirtInstructionBytes 349
24.4 vmirtAddRegisterWatchCallback 350
24.5 vmirtDeleteRegisterWatchCallback 351

25 Shared Object Access 352

25.1 vmirtDLOpen 353
25.2 vmirtDLClose 355
25.3 vmirtDLSymbol 356

25.4 VMIRTDLERROR 358

1 Introduction

This is reference documentation for **version 6.14.0** of the VMI *run time* function interface, defined in `ImpPublic/include/host/vmi/vmiRt.h`.

The functions in this interface are generally used within *embedded calls*[1] from translated native code implementing a processor model, generated by the VMI *morph time* interface. See the *VMI Morph Time Function Reference Guide* for more information about the VMI morph time interface and how to use it to create embedded calls.

Functions in the run time interface have the prefix `vmirt`.

[1] An *embedded call* is a function that is called from a JIT-translated code block to implement behavior that cannot easily be described using morpher primitives.

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.

When creating a processor model for use with QuantumLeap, be careful to avoid situations where a processor model relies on a cached value of a shared data item that could be updated by another processor in an embedded call, of the form:

```
void vmic_Callback(void) {
    tmp = <shared_value>;
    <vmi synchronizing call>;
    fn(tmp);
}
```

The above pseudo-example is potentially dangerous when QuantumLeap is active if `<shared_value>` can be modified by another processor while the current thread is suspended. To avoid problems like this, try to restructure routines so that the synchronizing call is made before any shared variable access:

```
void vmic_Callback(void) {
    <vmi synchronizing call>;
    tmp = <shared_value>;
    fn(tmp);
}
```

If it is not possible to modify the embedded call in this manner, use function `vmimtAtomic` when emitting the embedded call to ensure that the call is made with all other threads suspended. See the *VMI Morph Time Function Reference Guide* for more information about this function.

3 Model Configuration

A processor 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 `VMI_<type>_PARAM`, which reserves space for the value itself and also 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	32bit signed
VMI_UNS32_PARAM	32bit unsigned
VMI_UNS54_PARAM	64bit 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	32bit signed	specified min / max
VMI_UNS32_PARAM_SPEC	32bit unsigned	specified min / max
VMI_UNS54_PARAM_SPEC	64bit 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 `vmiAttrs.h`:

Prototype

```
#define VMI_PROC_PARAM_SPECS_FN(_NAME) vmiParameterP _NAME ( \
    vmiProcessorP processor,    \
    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 processor pointer, so can include or exclude parameters according to the current configuration.

Example

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

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

    VMI_ENUM_PARAM(variant);           // enumeration specifying the variant
    VMI_ENDIAN_PARAM(endian);          // endian parameter
    VMI_BOOL_PARAM(verbose);           // boolean parameter
    VMI_UNS32_PARAM(nInts);            // 32bit unsigned parameter
    VMI_UNS64_PARAM(rstVec);           // 64bit parameter used as an address
    VMI_STRING_PARAM(dataFile);        // string param used as a file name
    VMI_PTR_PARAM(data);               // pointer to platform provided data struct

} pVals, *pValsP;

//
// Define the list of legal variants
//
vmiEnumParameter variantList[] = {
    // name        value   description
    { "variant1",  1,  "first  variant" },
    { "variant2",  2,  "second variant" },
    { "variant3",  3,  "third  variant" },
};
```

```
//
// Define the parameters
//
static vmiParameter formals[] = {
    VMI_ENUM_PARAM_SPEC(  pVals, variant,  variantList, "processor variant"),
    VMI_ENDIAN_PARAM_SPEC(pVals, endian,                "processor endianness"),
    VMI_BOOL_PARAM_SPEC(  pVals, verbose,  0,           "Enable text output"),
    VMI_UNS32_PARAM_SPEC( pVals, nInts,    1, 0, 255,   "Number of int ports"),
    VMI_UNS64_PARAM_SPEC( pVals, rstVec,   0, 0, VMI_MAXU64, "Reset vec addr"),
    VMI_STRING_PARAM_SPEC(pVals, dataFile, "",         "dataFile file name "),
    VMI_PTR_PARAM_SPEC(   pVals, data,     0,          "Address  of data struct"),


    // 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
//
VMI_CONSTRUCTOR_FN(modelConstructor) {
    myProcP myProc = (processorP)processor;
    pValsP params  = (pValsP)parameterValues; // cast to my type

    ...
    // use the parameter values
    if (params->SETBIT(rstVec)) {
        ... params->rstVec ....
    }

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

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

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

4 Model Hardware Interface

An OVP model must provide to the simulator a specification of its hardware interface, so that the simulator can check the connections to the platform as they are made, and so that other tools can use the ICM API to interrogate a model and discover its interface without referring to the model's source code. The hardware interface specification separately covers Bus Ports, Net Ports and Connection Ports (also known as FIFO Ports).

4.1 Bus Port Specification

A bus port connects to a bus. A bus transmits read or write requests from a bus master to a bus slave (OVP does not represent details of bus timing, contention, or individual signals that comprise a real bus implementation). The model must provide an iterator function which returns the first or subsequent bus port specifications, or 0 at the end of the list. The function must be registered in the model attributes table and should be defined using this macro from `vmiPorts.h`:

Prototype

```
#define VMI_BUS_PORT_SPECS_FN(_NAME) vmiBusPortP _NAME ( \
    vmiProcessorP processor,    \
    vmiBusPortP   prev          \
)
```

Note that the iterator is also supplied with the processor pointer, so can include or exclude bus ports according to the current configuration.

Each specification includes:

- Bus port name;
- Bus port type (*master*, *slave* or *both*);
- Bus domain type (*code*, *data* or *other*);
- Minimum and maximum address bits supported on the bus;
- A Boolean indicating whether the port must be connected or may be left unconnected;
- Optional description fields;

The bus port structure also includes a *domain* field that is filled by the simulator with the memory domain object for any bus that it connected to the port.

The bus port specification is defined in `vmiPorts.h`.

Definition

```
typedef enum vmiBusPortTypeE {
    vmi_BP_MASTER,
    vmi_BP_SLAVE,
    vmi_BP_MASTER_SLAVE
} vmiBusPortType;

typedef enum vmiDomainTypeE {
    vmi_DOM_CODE,     // code domain port
    vmi_DOM_DATA,     // data domain port
    vmi_DOM_OTHER     // other domain port
} vmiDomainType;

typedef struct vmiBusPortS {

    const char      *name;
    vmiBusPortType   type;
    vmiDomainType    domainType;
    struct           {Uns8 min; Uns8 max; Uns8 unset;} addrBits;
    Bool             mustBeConnected;

    // space for documentation
    const char      *description;
    void            *descriptionDom;
```

```
    // domain is non-NULL if port is connected
    memDomainP      domain;

} vmiBusPort;
```

(Note: `addrBits.unset` and `descriptionDom` exist for historical reasons and are not used)

A processor model will typically have two bus master ports, of domain type code and data, named "INSTRUCTION" and "DATA". Other ports may be present for more complex processor models.

Because the *domain* field is filled by the simulator when the port is connected, processor models must implement a separate copy of the bus port list for each instance of a processor. This typically means that the list must be allocated and a pointer to it held on the processor structure. The following example shows how this is done for the standard OR1K processor model.

Example

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

//
// Return number of members of an array
//
#define NUM_MEMBERS(_A) (sizeof(_A)/sizeof((_A)[0]))

//
// Template bus port list
//
const static vmiBusPort busPorts[] = {
    {"INSTRUCTION", vmi_BP_MASTER, vmi_DOM_CODE, {32,32}, 1},
    {"DATA"       , vmi_BP_MASTER, vmi_DOM_DATA, {32,32}, 0},
};

//
// Allocate bus port specifications
//
static void newBusPorts(or1kP or1k) {

    Uns32 i;

    or1k->busPorts = STYPE_CALLOC_N(vmiBusPort, NUM_MEMBERS(busPorts));

    for(i=0; i<NUM_MEMBERS(busPorts); i++) {
        or1k->busPorts[i] = busPorts[i];
    }
}

//
// Free bus port specifications
//
static void freeBusPorts(or1kP or1k) {

    if(or1k->busPorts) {
        STYPE_FREE(or1k->busPorts);
        or1k->busPorts = 0;
    }
}

//
// Get the next bus port
//
VMI_BUS_PORT_SPECS_FN(or1kGetBusPortSpec) {
```

```
    or1kP or1k = (or1kP)processor;

    if(!prev) {

        // first port
        return or1k->busPorts;

    } else {

        // port other than the first
        Uns32 prevIndex = (prev-or1k->busPorts);
        Uns32 thisIndex = prevIndex+1;

        return (thisIndex<NUM_MEMBERS(busPorts)) ? &or1k->busPorts[thisIndex] : 0;
    }
}

//
// OR1K processor constructor
//
VMI_CONSTRUCTOR_FN(or1kConstructor) {

    or1kP or1k = (or1kP)processor;

    . . .

    // create bus port specifications
    newBusPorts(or1k);
}

//
// OR1K processor destructor
//
VMI_DESTRUCTOR_FN(or1kDestructor) {

    or1kP or1k = (or1kP)processor;

    . . .

    // free bus port specifications
    freeBusPorts(or1k);
}

//
// Register the function in the model attribute table
//
const vmiIASAttr modelAttrs = {
    ...
    .busPortSpecsCB = or1kGetBusPortSpec,
    ...
};
```

4.2 vmirtGetBusPortDomain

Prototype

```
memDomainP vmirtGetBusPortDomain(
    vmiProcessorP processor,
    vmiBusPortP   port
);
```

Description

This function is deprecated. Use `port->domain` to obtain the memory domain object connected to a port.

QuantumLeap Semantics

Non-self Synchronizing

4.3 Net Port Specification

A net port connects to a net. A net usually carries a boolean value, but can be used to carry a 32-bit value. The model must provide an iterator function which returns the first or subsequent net port specifications, or 0 at the end of the list. The function must be registered in the model attributes table and should use this macro from `vmiPorts.h`:

Prototype

```
#define VMI_NET_PORT_SPECS_FN(_NAME) vmiNetPortP _NAME ( \
    vmiProcessorP processor,    \
    vmiNetPortP   prev          \
)
```

Note that the iterator is also supplied with the processor pointer, so can use data associated with the instance.

Each specification includes:

- Net port name.
- Net port type (*input*, *output* or *in-out*).
- A callback function and user-data field (for an input).
- Pointer to a handle (used for output).
- Optional description.
- A Boolean indicating whether the port must be connected or may be left unconnected.

The net port specification object is defined in `vmiPorts.h`.

Definition

```
typedef struct vmiNetPortS {

    char            *name;
    vmiNetPortType   type;
    void            *userData;

    // callback for input or I/O net change
    vmiNetChangeFn   netChangeCB;
    Uns32           *handle;

    // space for documentation
    const char      *description;
    void            *descriptionDom;

    Bool             mustBeConnected;

} vmiNetPort;
```

A processor model will typically use net ports for reset, external interrupt inputs and for integration with an external interrupt controller. Note that if the net port is an output, the output handle must be allocated per port, per processor instance and that the handle pointer in the `vmiNetPort` structure must be set to point to this handle. The simulator will use this pointer to set the handle before simulation begins. The model then uses the handle to update the output net port when required.

Example

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

//
// Define (part of) the model instance data structure
//
typedef struct myProcS {
    ...
    Uns32       handle;  // handle for output net
    vmiNetPortP nets;    // definitions
    ...
} myProc, *myProcP;

//
// Callback function called when the input net is written from outside
//
VMI_NET_CHANGE_FN(extInt) {
    ...
}

//
// Call this to drive the ack net
//
static writeAck(myProcP proc, Bool value) {
    vmirtWriteNetPort(vmiProcessorP)proc, proc->handle, value);
}

//
// Call this from the constructor (once per instance)
//
static void makePorts(myProcP proc) {

    proc->nets = STYPE_CALLOC(vmiNetPort, 3);
    proc->nets[0] = {"intr0", vmi_NP_INPUT,  0, extInt, 0,  "interrupt input" };
    proc->nets[1] = {"ack",   vmi_NP_OUTPUT, 0, 0, &proc->handle, "ack output" };
    proc->nets[2] = { 0 };

}

//
// Callback to get the next net port
//
VMI_NET_PORT_SPECS_FN(getNetPortSpec) {
    myProcP proc = (myProcP)processor;
    if (!prev) {
        return proc->nets;
    }
    prev++;
    if (prev->name) {
        return prev;
    }
    return 0;
}

//
// Register the function in the model attribute table
//
const vmiIASAttr modelAttrs = {
    ...
    .netPortSpecsCB     = getNetPortSpec,
    ...
};
```

4.4 Connection port specification

If a model uses connection ports, it must define an iterator function that gives access to a list of connection port specifications. The iterator function is specified in the model attributes table. It returns the first or subsequent connection port specification structures, or 0 at the end of the list. The function must be registered in the model attributes table and should use this macro from `vmiPorts.h`:

Prototype

```
#define VMI_FIFO_PORT_SPECS_FN(_NAME) vmiFifoPortP _NAME ( \
    vmiProcessorP processor,    \
    vmiFifoPortP  prev          \
)
```

Note that the iterator is also supplied with the processor pointer, so can include or exclude bus ports according to the current configuration.

Each specification includes:

- Port name.
- Direction.
- Data width in bits.
- Offset of the port handle in the model's main data structure (the model writer is responsible for allocating a handle for each connection).
- An optional description.

The specification structure is defined in `vmiPorts.h`.

Definition

```
typedef struct vmiFifoPortS {

    const char      *name;
    vmiFifoPortType  type;
    Uns32            bits;
    void           **handle;
    void            *userData;

    // space for documentation
    const char      *description;
    void            *descriptionDom;

} vmiFifoPort;
```

During initialization, the simulator uses the iterator function to get the list of ports and set the port handles. The model can then use the following functions to send data through the connections.

This shows how the function might be defined and installed, and how the handles are created and referenced.

Example

```
//
// Add the handles to model instance structure
//
```

```
typedef struct myProcS {
    ...
    vmiFifoPortP   fifoPorts;       // fifo port descriptions
    ...
    vmiConnInputP  inputConn;       // input FIFO connection
    vmiConnOutputP outputConn;      // output FIFO connection
    ...
} myProc, *myProcP;


////////////////////////////////////////////////////////////////////////////////
// CONSTRUCTOR
////////////////////////////////////////////////////////////////////////////////

VMI_CONSTRUCTOR_FN(or1kConstructor) {
    ...

    // construct the FIFOs
    or1k->fifoPorts = newFifoPorts(or1k);
    // note that the destructor should free this structure.
    ...
}

////////////////////////////////////////////////////////////////////////////////
// FIFO PORTS
////////////////////////////////////////////////////////////////////////////////

//
// Allocate FIFO port structure. This is call from the model constructor.
//
static vmiFifoPortP newFifoPorts(or1kP or1k) {

    // declare temporary port structure (including terminator)
    vmiFifoPort template[] = {
        {"fifoIn" , vmi_FIFO_INPUT,  32, (void *)&or1k->inputConn },
        {"fifoOut", vmi_FIFO_OUTPUT, 32, (void *)&or1k->outputConn},
        VMI_END_FIFO_PORTS
    };

    // count members
    Uns32 numMembers = NUM_MEMBERS(template);
    Uns32 i;

    // allocate permanent port structure (including terminator)
    vmiFifoPortP result = STYPE_CALLOC_N(vmiFifoPort, numMembers);

    // fill permanent port structure
    for(i=0; i<numMembers; i++) {
        result[i] = template[i];
    }

    return result;
}


//
// iterator function to get the next fifo port
//
VMI_FIFO_PORT_SPECS_FN(or1kGetFifoPortSpec) {

    or1kP or1k = (or1kP)processor;

    if(!prev) {
        return or1k->fifoPorts;
    }

    prev++;

    if(prev->name) {
        return prev;
    }
```

```
    return 0;
}

//
// Make the function available to the simulator, by adding an entry to the
// modelAttrs structure.
//
const vmiIASAttr modelAttrs = {
    ...
    .fifoPortSpecsCB = getFifoPortSpec,
    ...
};
```

5 Simulator Control

This section describes simulator control functions for simulation exit, restart of processors in a halted state, processor mode change, and simulated interrupt handling. It also discusses some special support for instruction counting.

5.1 vmirtYield

Prototype

```
void vmirtYield(vmiProcessorP processor);
```

Description

In a multiprocessor simulation, this function causes the simulator to cease simulation of the passed processor and resume with the next runnable processor.

Example

```
#include “vmi/vmiRt.h”

// embedded call made on move to control status register
static void vmic_MoveToCSR(cpuxP cpux, Uns32 csrId, Uns32 value) {

    if((csrId==CPUX_YIELD_CSR) && (value==1)) {
        vmirtYield((vmiProcessorP)cpux);
    } else {
        // handle other CSR writes
        // … etc …
    }
}
```

Notes and Restrictions

1. This function has no effect for processors other than the currently-executing processor.

QuantumLeap Semantics

Non-self-synchronizing

5.2 vmirtHalt

Prototype

```
void vmirtHalt(vmiProcessorP processor);
```

Description

This function halts the passed processor. No more instructions will be executed by that processor unless it is restarted by a call to `vmirtRestartNext` or `vmirtRestartNow`. This could be done by another processor (in a multiprocessor simulation) or by an exception (for example, a countdown timer expiring or an external event signal being activated).

If the processor is the currently executing processor, it will halt on completion of the current instruction.

Example

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

// embedded call made on move to control status register
static void vmic_MoveToCSR(cpuxP cpux, Uns32 csrId, Uns32 value) {

    if(csrId==CPUX_HALT_CSR) {
        // handle write to halt CSR
        vmirtHalt((vmiProcessorP)cpux);
    } else {
        // handle other CSR writes
        // … etc …
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

5.3 vmirtInterrupt

Prototype

```
void vmirtInterrupt(vmiProcessorP processor);
```

Description

This function causes simulation of the passed processor to stop on completion of the current simulated instruction and return from the calling context (the `icmSimulate` or `icmSimulatePlatform` invocation) with a stop reason of `ICM_SR_INTERRUPT`.

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

5.4 vmirtExit

Prototype

```
void vmirtExit(vmiProcessorP processor);
```

Description

This function ends execution for the passed processor. In a multiprocessor simulation, other processors will continue execution until they also exit or simulation is finished (see `vmirtFinish`).

If the processor is the currently executing processor, it will exit on completion of the current instruction.

This function is typically used to terminate simulations when special pseudo-opcodes are detected.

Example

```
#include “vmi/vmiRt.h”

// embedded call made on move to control status register
static void vmic_MoveToCSR(cpuxP cpux, Uns32 csrId, Uns32 value) {

    if(csrId==CPUX_END_SIM_CSR) {
        // handle write to special end simulation pseudo-CSR
        vmiPrintf(
            “Processor %u exits with result %u\n”,
            vmirtCPUId),
            Value
        );
        vmirtExit((vmiProcessorP)cpux);
    } else {
        // handle normal CSR write
        // … etc …
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.5 vmirtFinish

Prototype

```
void vmirtFinish(Int32 status);
```

Description

This function causes simulation to finish. An example use might be within a store exception handler to detect a write to a magic address that should cause simulation termination.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

// structure field accessor macros
#define MAGIC_TERM_ADDR 0x10001000

VMI_WRITE_EXCEPT_FN(cpuxWriteExceptCB) {
    if(address==MAGIC_TERM_ADDR) {
        ASSERT(bytes==4, “expected 4-byte write to term address\n”);
        vmiPrintf(“Test ended with status: %u\n”, *(Uns32*)value);
        vmirtFinish(0);
    } else {
        . . . etc . . .
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.6 vmirtStop

Prototype

```
void vmirtStop();
```

Description

This function causes simulation to be interrupted. If the Imperas MP Debugger is available the debug shell shall be entered. If there is no Imperas MP Debugger available the function behaves in the same way as `vmirtFinish(0)` and the simulation is finished. An example use might be within a store exception handler to detect a write to a magic address that should cause simulation to stop or terminate.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

// structure field accessor macros
#define MAGIC_TERM_ADDR 0x10001000

VMI_WRITE_EXCEPT_FN(cpuxWriteExceptCB) {
    if(address==MAGIC_TERM_ADDR) {
        ASSERT(bytes==4, “expected 4-byte write to term address\n”);
        vmiPrintf(“Test ended with status: %u\n”, *(Uns32*)value);
        vmirtStop();
    } else {
        . . . etc . . .
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.7 vmirtAtomic

Prototype

```
void vmirtAtomic();
```

Description

In a parallel simulation, this function causes all other parallel threads to be suspended before it returns. The parallel threads will remain suspended throughout the execution of the simulated instruction which has called `vmirtAtomic` (typically, an embedded call or a function called from an embedded call). If the simulation is not parallel the function has no effect.

This function is typically used before access is made to some data that is shared between parallel threads.

Example

This example is taken from the OVP ARM processor model. In this model, there is some shared data accessible via the `root` pointer on the processor instance. To ensure consistency, all accesses to the root pointer are preceded by a call to `vmirtAtomic`.

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

inline static armP getClusterRoot(armP arm) {
    vmirtAtomic();
    return arm->root;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.8 vmirtBlock

Prototype

```
void vmirtBlock(vmiProcessorP processor);
```

Description

This function halts the passed processor. No more instructions will be executed by that processor unless it is restarted by a call to `vmirtRestartNext` or `vmirtRestartNow`. This could be done by another processor (in a multiprocessor simulation) or by an exception (for example, a countdown timer expiring or an external event signal being activated).

If the processor is the currently executing processor, the current instruction will be terminated; when the processor restarts, the current instruction will be re-executed.

Example

```
#include “vmi/vmiRt.h”

//
// Get from connection, block if data not available
//
static void vmic_ConnPutOrBlock(or1kP or1k, Uns32 thisPC, Uns32 rb) {

    if(!vmirtConnPut(or1k->outputConn, &or1k->regs[rb])) {

        // halt this processor and re-execute this instruction on restart
        vmirtBlock((vmiProcessorP)or1k);

        // install notifier to be called on connection event
        vmirtConnNotifyGet(or1k->outputConn, restartAfterBlock);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

5.9 vmirtIsHalted

Prototype

```
Bool vmirtIsHalted(vmiProcessorP processor);
```

Description

Processor execution may be halted wither by a call to `vmirtHalt` (described elsewhere in this section) or by the VMI morph time function `vmimtHalt` (defined in `vmiMt.h`). While in the halted state, a processor will execute no instructions.

This function enables a model to determine whether a processor is currently in a halted state. This is typically used in conjunction with `vmirtRestart` to restart a halted processor on some event, often from within an external interrupt callback function as shown in the example below.

Example

```
#include "vmi/vmiRt.h"
#include "vmi/vmiTypes.h"

// external interrupt request handler for cpux processor type
VMI_NET_CHANGE_FN(cpuxExternalInterrupt) {

    // cast to specific processor type from generic type
    cpuxP cpux = (cpuxP)processor;

    // call model-specific routine to determine if there is a pending
    // interrupt
    Bool interruptPending = cpuxInterruptPending(cpux, userData);

    // if the processor is halted, a pending interrupt will cause it
    // to restart and handle the interrupt
    if(interruptPending && vmirtIsHalted(processor)) {
        vmirtRestart(processor);
    }

    return interruptPending;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

5.10 vmirtRestartNext

Prototype

```
void vmirtRestartNext(vmiProcessorP processor);
```

Description

Processor execution may be halted wither by a call to `vmirtHalt` (described elsewhere in this section) or by the VMI morph time function `vmimtHalt` (defined in `vmiMt.h`). While in the halted state, a processor will execute no instructions.

This function enables a model to restart a processor that is in a halted state. This is typically used from within an external interrupt callback function as shown in the example below.

The simulator runs each processor in a multiprocessor simulation for a time slice. This function will restart the halted processor at the start of the next time slice; use `vmirtRestartNow` to restart the halted processor immediately.

Example

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

// external interrupt request handler for cpux processor type
VMI_NET_CHANGE_FN(cpuxExternalInterrupt) {

    // cast to specific processor type from generic type
    cpuxP cpux = (cpuxP)processor;

    // call model-specific routine to determine if there is a pending
    // interrupt
    Bool interruptPending = cpuxInterruptPending(cpux, userData);

    // if the processor is halted, a pending interrupt will cause it
    // to restart and handle the interrupt
    if(interruptPending && vmirtIsHalted(processor)) {
        vmirtRestartNext(processor);
    }

    return interruptPending;
}
```

Notes and Restrictions

1. This function has no effect if the target processor is not in the halted state.

QuantumLeap Semantics

Non-self-synchronizing

5.11 vmirtRestartNow

Prototype

```
void vmirtRestartNow(vmiProcessorP processor);
```

Description

Processor execution may be halted wither by a call to `vmirtHalt` (described elsewhere in this section) or by the VMI morph time function `vmimtHalt` (defined in `vmiMt.h`). While in the halted state, a processor will execute no instructions.

This function enables a model to restart a processor that is in a halted state. This is typically used from within an external interrupt callback function as shown in the example below.

The simulator runs each processor in a multiprocessor simulation for a time slice. This function will restart the halted processor immediately; use `vmirtRestartNext` to restart the halted processor at the start of the next time slice.

Example

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

// external interrupt request handler for cpux processor type
VMI_NET_CHANGE_FN(cpuxExternalInterrupt) {

    // cast to specific processor type from generic type
    cpuxP cpux = (cpuxP)processor;

    // call model-specific routine to determine if there is a pending
    // interrupt
    Bool interruptPending = cpuxInterruptPending(cpux, userData);

    // if the processor is halted, a pending interrupt will cause it
    // to restart and handle the interrupt
    if(interruptPending && vmirtIsHalted(processor)) {
        vmirtRestartNow(processor);
    }

    return interruptPending;
}
```

Notes and Restrictions

1. This function has no effect if the target processor is not in the halted state.

QuantumLeap Semantics

Non-self-synchronizing

5.12 vmirtTraceOnAfter

Prototype

```
void vmirtTraceOnAfter(vmiProcessorP processor, Uns64 delta);
```

Description

This function turns on instruction tracing after the given number of instructions. After the given number each instruction executed by the processor will disassembled into the simulator output log.

Calling `vmirtTraceOnAfter` a second time before the elapsed number of instructions will restart the count at the new value.

Example

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

// external interrupt request handler for cpux processor type
VMIOS_INTERCEPT_FN(turnOnTracing) {

    vmirtTraceOnAfter(processor, 5);
    vmirtTraceOffAfter(processor, 105);
}
```

Notes and Restrictions

1. See `vmirtTraceOffAfter`.
2. This command will override instruction tracing controlled from the simulator command line, and any previously made calls to `icmTraceOffAfter`.

QuantumLeap Semantics

Non-self-synchronizing

5.13 vmirtTraceOffAfter

Prototype

```
void vmirtTraceOffAfter(vmiProcessorP processor, Uns64 delta);
```

Description

This function turns off instruction tracing after the given number of instructions. After the given number instruction disassembly will be disabled.

Calling `vmirtTraceOffAfter` a second time before the elapsed number of instructions will restart the count at the new value.

Example

```
#include “vmi/vmiRt.h”

// external interrupt request handler for cpux processor type
VMIOS_INTERCEPT_FN(turnOnTracing) {

    vmirtTraceOnAfter(processor, 5);
    vmirtTraceOffAfter(processor, 105);
}
```

Notes and Restrictions

1. See vmirtTraceOnAfter.
2. This command will override instruction tracing controlled from the simulator command line, and any previously made calls to `icmTraceOffAfter`.

QuantumLeap Semantics

Non-self-synchronizing

5.14 vmirtFlushTarget

Prototype

```
void vmirtFlushTarget(vmiProcessorP processor, Addr flushPC);
```

Description

This routine flushes any code block from the *current* dictionary at the passed address. It is required when the processor has undergone some state change that would make any previously-generated code block in the current dictionary at that address potentially invalid.

Within processor models, it is usually much more efficient to implement modal behavior of this kind by having multiple code dictionaries to represent different modes (see the section *Simulated Memory Access* elsewhere in this document for a detailed explanation of code dictionaries and their interaction with processor modes). However, the ability to flush a target address is often useful in *intercept libraries*. As an example, an intercept library may require to install functionality that is executed on *return* from a function. It can do this using `vmirtFlushTarget` as follows:

1. Intercept the first instruction of the function of interest;
2. Within the primary intercept callback, determine the function return address (for example, by examining the ABI stack pointer register);
3. Use `vmirtFlushTarget` to delete any code block that already exists for that return address;
4. In the intercept library morpher callback, emit an embedded call to perform the action required.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiOSAttrs.h”
#include “vmi/vmiOSLib.h”

VMIOS_INTERCEPT_FN(traceOnReturn) {

    vmiRegInfoCP info = vmiosGetRegDesc(processor, “lr”);
    Uns32        ra;

    // read return address (32-bit on this processor)
    vmiosRegRead(processor, info, &ra);

    // flush any code block at the return address
    vmirtFlushTarget(processor, ra);
}
```

Notes and Restrictions

1. See related function `vmirtFlushTargetMode`, which enables code blocks to be flushed in a different processor mode to the current mode.

QuantumLeap Semantics

Synchronizing

5.15 vmirtFlushTargetMode

Prototype

```
void vmirtFlushTargetMode(
    vmiProcessorP processor,
    Addr          flushPC,
    Uns32         mode
);
```

Description

This routine flushes any code block from the dictionary indicated by the passed mode at the passed address. It is required when the processor has undergone some state change that would make any previously-generated code block in that dictionary at that address potentially invalid.

The ability to flush a target address is often useful in *intercept libraries*. As an example, an intercept library may require to install functionality that is executed in some circumstances on *return from an exception*. It can do this using `vmirtFlushTargetMode` as follows:

1. Intercept the first instruction of the exception handler of interest (in the *kernel mode* dictionary);
2. Within the primary intercept callback, determine the system call return address and processor mode (by examining the processor privileged register state);
3. Use `vmirtFlushTargetMode` to delete any code block that already exists for that return address *in the return mode*;
4. In the intercept library morpher callback, emit an embedded call to perform the action required on return the old mode.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiOSAttrs.h”
#include “vmi/vmiOSLib.h”

VMIOS_INTERCEPT_FN(traceOnRFE) {

    vmiRegInfoCP epc = vmiosGetRegDesc(processor, “epc”); // saved PC
    vmiRegInfoCP esr = vmiosGetRegDesc(processor, “esr”); // saved status register
    Uns32        ra;
    Uns32        sr;

    // read return address (32-bit on this processor)
    vmiosRegRead(processor, epc, &ra);

    // read saved status register (32-bit on this processor) and extract old mode
    // (model-specific)
    vmiosRegRead(processor, esr, &sr);
    Uns32 oldMode = extractMode(sr);

    // flush any code block at the return address
    vmirtFlushTargetMode(processor, ra, oldMode);
}
```

Notes and Restrictions

1. See related function `vmirtFlushTarget` which allows flushing of code blocks in the current mode.

QuantumLeap Semantics

Synchronizing

5.16 vmirtFlushTargetModeTagged

Prototype

```
void vmirtFlushTargetMode(
    vmiProcessorP processor,
    Addr          flushPC,
    Uns32         mode,
    vmiBlockTag   tagMask,
    vmiBlockTag   tagValue,
    Bool          flushIfEqual
);
```

Description

This routine conditionally flushes code blocks from the dictionary indicated by the passed mode at the passed address. It is required when the processor has undergone some state change that could make a previously-generated code block in that dictionary at that address potentially invalid.

Whether a block is flushed depends on the `tagMask`, `tagValue` and `flushIfEqual` arguments. If `flushIfEqual` is `True`, then a block will be flushed only if:

```
((block->tag & tagMask) == tagValue)
```

If `flushIfEqual` is `False`, then a block will be flushed only if:

```
((block->tag & tagMask) != tagValue)
```

In these expressions, `block->tag` is the tag associated with the block when it was constructed. See function `vmimtTagBlock` in the *VMI Morph Time Function Reference Manual* for more information.

Example

This example is from the OVP ARC processor model. The ARC processor implements a *zero-overhead loop* construct in which a code block can only be reused if a limiting address held in a register holds a certain value. Setting the register to a new address must invalidate any blocks at that address unless they implement zero-overhead loop behavior, in which case they can be preserved.

Unnecessary block flushes are suppressed by using tagged blocks as follows:

```
#include “vmi/vmiMt.h”
#include “vmi/vmiTypes.h”

void arcEmitZeroOverheadLoop(arcMorphStateP state) {

    if(state->atZOL) {

        // when executing this block, validate that lp_end has the same value
        // that it has when code for the block was generated
        vmimtValidateBlockMaskR(ARC_GPR_BITS, ARC_AUX_REG(lp_end), -1);

        // tag this block to avoid unnecessary deletion when lp_end changes
```

```
        vmimtTagBlock(VBT_1);

        . . .
    }
}
```

In a callback that is activated when the `lp_end` register changes, code blocks at the end address implied by the new value of the `lp_end` register are flushed using this idiom:

```
static void flushTargetZOL(arcP arc, Uns32 lpEnd) {

    for(mode=0; mode<ARC_MODE_LAST; mode++) {
        vmirtFlushTargetModeTagged(
            (vmiProcessorP)arc, lpEnd, mode, VBT_1, VBT_1, False
        );
    }
}
```

The call to `vmirtFlushTargetModeTagged` in this case will flush any code block at address `lpEnd` for which the expression

```
((block->tag & VBT_1) == VBT_1)
```

is `False`. This flushes any code block that was not tagged as a zero-overhead loop block when it was constructed.

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

5.17 vmirtFlushDict

Prototype

```
void vmirtFlushDict(vmiProcessorP processor);
```

Description

This routine flushes all code blocks from the current dictionary. It is required when the processor has undergone some state change that would make any previously-generated code blocks in the current dictionary potentially invalid.

It is usually much more efficient to implement modal behavior of this kind by having multiple code dictionaries to represent different modes (see the section *Simulated Memory Access* elsewhere in this document for a detailed explanation of code dictionaries and their interaction with processor modes). However, it occasionally makes sense to implement mode changes by flushing a dictionary instead. An example might be simulation of a processor with a hardware mode that enables a trace buffer to record a sequence of recent branch addresses. In almost all normal operation, the trace buffer is disabled. In the very rare case that the trace buffer is enabled and must be simulated, every branch must be preceded in translated native code by an embedded call to simulate the buffer. In these circumstances, it is often more convenient to simply flush the current dictionary when the trace buffer is enabled or disabled by a processor state change instead of incurring the overhead of another dictionary to model the trace buffer.

Example

```
#include “vmi/vmiRt.h”

// embedded call made on status register (CPUX_SR) change
static void vmic_TestTraceModeChange(cpuxP cpux) {

    Bool simTraceEnabled = cpux->regs[CPUX_SR] & HW_TRACE_MODE;

    if(simTraceEnabled!=cpux->simTraceEnabled) {
        vmiProcessorP processor = (vmiProcessorP)cpux;
        vmirtFlushDict((vmiProcessorP)cpux);
        cpux->simTraceEnabled = simTraceEnabled;
    }
}
```

Notes and Restrictions

1. `vmirtFlushDict` is very slow because all code blocks in a dictionary must be discarded and retranslated. Use `vmirtFlushDict` only in very rare circumstances as described above.
2. See also the related routine `vmirtSetPCFlushDict`, which allows the current program counter to be set while the dictionary is flushed.

QuantumLeap Semantics

Synchronizing

5.18 vmirtFlushAllDicts

Prototype

```
void vmirtFlushAllDicts(vmiProcessorP processor);
```

Description

This routine flushes all code blocks from the all dictionaries of the passed processor. It is required when the processor has undergone some state change that would make any previously-generated code blocks in the any dictionary potentially invalid.

This function is typically used in intercept libraries that are enabled or disabled at run time and which provide a morpher callback that overrides the normal processor code generation when enabled. In such cases, when the intercept library is either enabled or disabled, all existing code blocks for the processor must be regenerated.

Example

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

static void enableIntercept(
    vmiosObjectP  object,
    vmiProcessorP processor,
    Bool          enable
) {
    if(enable!=object->enable) {
        vmirtFlushAllDicts(processor);
        object->enable = enable;
    }
}
```

Notes and Restrictions

1. `vmirtFlushAllDicts` is very slow because all code blocks in a dictionary must be discarded and retranslated. Use `vmirtFlushAllDicts` only in very rare circumstances as described above.

QuantumLeap Semantics

Synchronizing

5.19 vmirtGetNetPortHandle

Prototype

```
void vmirtGetNetPortHandle(
    vmiProcessorP processor,
    const char   *netPortName
);
```

Description

Deprecated. See section 4 for more information about the improved methodology for processor net port connection.

QuantumLeap Semantics

Non-self-synchronizing

5.20 vmirtWriteNetPort

Prototype

```
void vmirtWriteNetPort(
    vmiProcessorP  processor,
    Uns32          handle,
    Uns32          value
);
```

Description

This function writes a value to a processor output net port, the handle of which was set up in the net port specification. See section 4 for more information about the methodology for processor net port connection.

Example

```
#include “vmi/vmiRt.h”

// processor structure definition
typedef struct cpuxS {
    Uns32 dataHandle;
    Uns32 watchdogHandle;
} cpux, *cpuxP;

// trigger watchdog signal
static void writeWatchdog(cpuxP cpux) {
    vmirtWriteNetPort((vmiProcessorP)cpux, cpux->watchdogHandle, 1);
}

// read data input
static Uns32 readInput(cpuxP cpux) {
    return vmirtReadNetPort((vmiProcessorP)cpux, cpux->dataHandle);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.21 vmirtReadNetPort

Prototype

```
Uns32 vmirtReadNetPort(
    vmiProcessorP  processor,
    Uns32          handle
);
```

Description

This function reads a value from a processor input net port, the handle of which was which was set up in the net port specification. See section 4 for more information about the methodology for processor net port connection.

Example

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

// processor structure definition
typedef struct cpuxS {
    Uns32 dataHandle;
    Uns32 watchdogHandle;
} cpux, *cpuxP;

// trigger watchdog signal
static void writeWatchdog(cpuxP cpux) {
    vmirtWriteNetPort((vmiProcessorP)cpux, cpux->watchdogHandle, 1);
}

// read data input
static Uns32 readInput(cpuxP cpux) {
    return vmirtReadNetPort((vmiProcessorP)cpux, cpux->dataHandle);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

5.22 vmirtInstallNetCallback

Prototype

```
Uns32 vmirtInstallNetCallback(
    vmiProcessorP  processor,
    const char    *netPortName,
    vmiNetChangeFn netChangeCB,
    void          *userData
);
```

Description

This function adds a notifier to a net port, called when the net is written.

Example

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

struct context {
    …
} myContext;

VMI_NET_CHANGE_FN(netNotify) {
    vmiPrintf("Net written with %u\n", newValue);
}

    vmirtInstallNetCallback(processor, "myPort", netNotify, &myContext);
```

Notes and Restrictions

1. This function is only for use in an intercept library that is monitoring a model; see section 4 for information about the methodology for processor net port connection.
2. Notifiers may only be installed on net ports that are connected to nets in the platform.
3. Notifiers may be installed on both output and input net ports.

QuantumLeap Semantics

Synchronizing

5.23 vmirtDoSynchronousInterrupt

Prototype

```
void vmirtDoSynchronousInterrupt(vmiProcessorP processor);
```

Description

When modeling processor status registers, it is often the case that a change to a status register will cause a processor interrupt to happen immediately after the instruction completes. For example, if a processor has an instruction specifically to enable interrupts, then any previously pending interrupt should be taken immediately after the enable.

This function should be used in an embedded call to indicate that there are exceptions to be processed immediately *after* this simulated instruction completes.

Example

```
#include "vmi/vmiRt.h"
#include "vmi/vmiMt.h"

// callback when the status register SR is set
static void vmic_SetSR(cpuxP cpux, Uns32 newFlags) {

    // it is never possible to clear the fixed-one (FO) bit
    newFlags |= SPR_SR_FO;

    // set the SR status register
    cpux->regs[CPUX_REG_SR] = newFlags;

    // schedule an interrupt if required
    Uns32 enabled = newFlags & cpux->pendingIntMask;
    if(isInterruptHigh(cpux) && isEnabledIEE(cpux)) {
        vmirtDoSynchronousInterrupt((vmiProcessorP)cpux);
    }

    // set the current branch flag, carry flag and overflow flag from the SR
    cpux->branchFlag   = GET_BIT(newFlags, SPR_SR_F);
    cpux->carryFlag    = GET_BIT(newFlags, SPR_SR_CY);
    cpux->overflowFlag = GET_BIT(newFlags, SPR_SR_OV);
}

// emit code to model return from exception
static void emitRFE(Uns32 instr) {
    vmimtArgProcessor();
    vmimtArgReg(32, CPUX_REG(CPUX_REG_ESR));
    vmimtCall((vmiCallFn)vmic_SetSR);
    vmimtUncondJumpReg(0, CPUX_EPCR, VMI_NOREG, vmi_JH_RETURNINT);
}
```

Notes and Restrictions

1. The interrupt will be performed only *after* all actions for the current simulated instruction have completed. In the example above, the processor will be just about to execute the instruction at the address previously in register `CPUX_EPCR` when the interrupt is performed.
2. Currently, the interrupt will always be taken before the next instruction. There is no support for an interrupt after more than one instruction.

QuantumLeap Semantics
Non-self-synchronizing

5.24 vmirtCacheRegister

Prototype

```
const char *vmirtCacheRegister( \
    vmiProcessorP        processor,
    vmiCacheType         cacheType,
    Bool                 enable,
    vmiCacheHitFn        hitCB,
    vmiCacheMissFn       missCB,
    vmiCacheInvalidateFn invalidateCB,
    void                *userData
);
```

Description

On processor models that implement a cache model (such as the mips32 model) this function may be used to enable/disable the cache model and register call back functions for cache events.

When enable is true the cache is enabled, when false it is disabled.

The types `vmiCacheType`, `vmiCacheHitFn`, `vmiCacheMissFn` and `vmiCacheInvalidateFn` are defined in `vmiCacheAttrs.h`:

```
typedef enum vmiCacheTypeE {
    VMI_CT_I = 0,  // level 1 instruction
    VMI_CT_D = 1,  // level 1 data
    VMI_CT_LAST    // KEEP LAST - May be used for array size
} vmiCacheType, *vmiCacheTypeP;

#define VMI_CACHE_HIT_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,      \
    Uns32          rowIndex,       \
    Addr           hitAddr,        \
    void          *userData        \
)
typedef VMI_CACHE_HIT_FN((*vmiCacheHitFn));

#define VMI_CACHE_MISS_FN(_NAME) void *_NAME( \
    vmiProcessorP  processor,      \
    Uns32          rowIndex,       \
    Addr           missAddr,       \
    Addr           victimAddr,     \
    void          *cacheContext,   \
    void          *userData        \
)
typedef VMI_CACHE_MISS_FN((*vmiCacheMissFn));

#define VMI_CACHE_INVALIDATE_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,      \
    Uns32          rowIndex,       \
    Addr           invalidateAddr, \
    Addr           victimAddr,     \
    void          *cacheContext,   \
    void          *userData        \
)
typedef VMI_CACHE_INVALIDATE_FN((*vmiCacheInvalidateFn));
```

`cacheType` is used to specify either the instruction or data cache. (Only level 1 caches are supported)

The `hitCB`, `missCB` and `invalidateCB` function pointers, if not NULL, specify the function to be called when that event occurs in the cache.

The `vmiCacheMissFn` returns a `void *` value that will be associated with the cache line that is added as a result of the miss. If that line is evicted this value will be passed as the `cacheContext` value on the miss or invalidate callback. This can be useful for tracking the context (e.g. process and/or function) that was active when a line was added to the cache.

When called the functions are passed the following arguments:

- `processor`: The processor making the access that caused the event
- `rowIndex`: The row of the cache being accessed
- `hitAddr`: The address accessed on a cache hit
- `missAddr`: The address accessed on a cache miss
- `invalidateAddr`: The address accessed on a cache invalidate
- `victimAddr`: The address accessed when victim cache line was added
- `cacheContext`: Value returned by `cacheMiss()`when the victim cache line was added
- `userData`: The value provided as the last argument to `vmirtCacheRegister()`

`vmirtCacheRegister()` returns a unique name for the cache if successful. If the same cache is connected to multiple processors they will each get the same name returned.

If the call is not successful NULL will be returned. This may occur if the processor model does not support a cache model.

Example

```
static VMI_CACHE_HIT_FN(notifyHit) {
    ...
}
static VMI_CACHE_MISS_FN(notifyMiss) {
    ...
}
static VMI_CACHE_INVALIDATE_FN(notifyInvalidate) {
    ...
}

const char *mipsCacheEnable(
    vmiosObjectP  object,
    vmiCacheType  type,
    Bool          enable)
{

    VMI_ASSERT(type < VMI_CT_LAST, "Illegal Cache type");

    const char *cacheName =
        vmirtCacheRegister(
            object->processor,
            type,
            enable,
            notifyHit,
            notifyMiss,
            notifyInvalidate,
```

```
            cache
        );

    if (!cacheName) {

        // Cache not supported
        vmiMessage (
            "I", PLUGIN_PREFIX,
            "%s : unable to register %s\n",
            vmirtProcessorName(object->processor),
            vmiCacheTypeName(type)
        );
   }

   Return cacheName;
}
```

Notes and Restrictions

1. When a cache is shared between multiple processors (e.g. in a multi-threaded processor such as the MIPS 34K) callbacks must be registered separately on every processor connected to the cache.

QuantumLeap Semantics

Non-self-synchronizing

6 Instruction Counting

The VMI Run-Time Function API interface defines functions to:

1. Obtain the nominal IPS (instructions-per-second) rate for a processor;
2. Obtain the current instruction count for a processor;
3. Set or clear instruction count callbacks to be triggered after a specified number of instructions have been executed by a processor or SMP group (see section 14 for more information on SMP processor modeling).

Prior to VMI version 2.0.17, each processor allowed only a single instruction counter callback to be specified, within the processor model itself

In VMI versions 2.0.17 and later, multiple instruction counter callbacks can be specified, and it is now possible to specify and use instruction counter callbacks from within intercept libraries.

In VMI versions 2.0.20 and later, it was possible to create a timer at any level in an SMP processor hierarchy. Prior to this version, timers could only be created at leaf level, and were shared by all members of an SMP group.

In VMI versions 6.1.0 and later, it is possible to create *monotonic timers*, which present a consistent view of monotonically-increasing time for all timers in a platform. With this version, creation of timers at processor levels above individually-schedulable processors is *no longer supported* (timers at this level were used to emulate monotonic timers in earlier VMI versions).

6.1 vmirtGetProcessorIPS

Prototype

```
Flt64 vmirtGetProcessorIPS(vmiProcessorP processor);
```

Description

This function call returns the nominal *instructions-per-second* speed of the given processor. This is the instructions-per-second implied by the `mips` processor attribute given when the processor was created. This information can be useful when modeling timers that expire after a specified *time interval* rather than a cycle delay (it provides a method to convert from a time delay to an equivalent number of instructions).

For simulations under full control of the built-in scheduler (for example, launched by `icmSimulatePlatform`) the nominal instructions-per-second is directly related to the current simulation time. For simulations under external scheduler control (for example, a SystemC simulation using `icmSimulate` to simulate a processor) it is the external scheduler's responsibility to ensure that the processor simulation rate corresponds to the specified MIPS rate.

Example

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

static void vmic_DebugIPS(cpuxP cpux) {
    vmiPrintf("IPS: %f\n", vmirtGetProcessorIPS((vmiProcessorP)cpux));
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

6.2 vmirtGetICount

Prototype

```
Uns64 vmirtGetICount(vmiProcessorP processor);
```

Description

The simulator maintains a count of executed instructions for every processor in a simulation. This function call returns the count for a specific processor.

For a non-SMP processor, or for an SMP leaf processor that is not a member of an SMP group, the count is the sum of instructions actually executed on the processor and instructions *for which the processor was halted*. This means that the return value can conveniently be used to implement instruction count interrupts, as the value returned is equivalent to a cycle count, if every instruction is assumed to have taken a single cycle.

For an SMP leaf level processor that is a member of an SMP group, the count is the number of instructions executed on the processor only. Halted instructions are not counted at this level.

For an SMP group, the count is the sum of the instructions executed by leaf level processors when members of this group and the number of instructions for which the group was halted (i.e. for which there were no runnable members). The return value can therefore conveniently be used to implement instruction count interrupts at the SMP group level.

For SMP containers that are not groups, the count is the sum of the instructions executed by all current children, according to the definitions above.

Example

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

static void vmic_DebugICount(cpuxP cpux) {
    vmiPrintf("ICount: %llu\n", vmirtGetICount((vmiProcessorP)cpux));
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

6.3 vmirtSetICountInterrupt

Prototype

```
void vmirtSetICountInterrupt(vmiProcessorP processor, Uns64 iDelta);
```

Description

The simulator maintains a count of executed instructions for every processor in a simulation (see the description of `vmirtGetICount` for exact details on how instructions are counted). This function causes the *default instruction count interrupt callback* to be called when the processor instruction count equals the current processor instruction count plus the count specified with the `iDelta` argument.

The instruction count interrupt callback is defined with the `VMI_ICOUNT_FN` macro and installed using the `icountExceptCB` field of the processor `vmiIASAttr` attribute structure (see `vmiAttrs.h`). The `VMI_ICOUNT_FN` macro is defined in `vmiTypes.h` as:

```
#define VMI_ICOUNT_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,   \
    vmiModelTimerP timer,       \
    Uns64          iCount,      \
    void          *userData     \
)
typedef VMI_ICOUNT_FN((*vmiICountFn));
```

The `processor` argument is the processor on which the timer has expired. The `timer` argument is the implicit model timer that is created automatically for every processor model (see function `vmirtCreateModelTimer` in this section for information about how to create explicit timers if more than one timer is required). The `iCount` argument is the current instruction count. The `userData` argument is always `NULL` for the default instruction count callback.

The callback should analyze the processor state and indicate that an interrupt is pending by calling `vmirtDoSynchronousInterrupt` if required. The `processor` argument to `vmirtDoSynchronousInterrupt` may be the current processor or any other processor in the simulation. Every processor subject to a synchronous interrupt call will immediately stop simulating, and the simulator will call the processor's instruction fetch handler to determine what to do next. The processor instruction fetch exception handler is responsible for arbitrating between the instruction count exception and any other pending exceptions and taking appropriate action (for example, entering kernel mode and setting the program counter to a simulated exception handler).

Example

This example shows the `COUNT`/`COMPARE` timer implementation from the OVP MIPS model. Note that this model implements a microthreaded SMP processor, and the timer is at the SMP group level (the VPE object in MIPS parlance).

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

static void scheduleTimerInterrupt(
```

```
    mipsP       vpe,
    const char *reason,
    Bool        atExpiry
) {
    if(!COP0_FIELD(vpe, Cause, DC)) {

        // counter enabled - schedule the next interrupt
        Uns32 delta32 = getCountCompareDelta(vpe);
        Uns64 delta64 = !delta32 && atExpiry ? 0x100000000ULL : delta32;

        // use default instruction timer interrupt
        vmirtSetICountInterrupt((vmiProcessorP)vpe, delta64);

        // emit debug output if required
        if(MIPS32_DEBUGTIMER(vpe)) {
            vmiMessage(
                "I", CPU_PREFIX,
                "%s: %s (Compare=%u Count=%u CauseDC=%u) - "
                "schedule timer interrupt after "FMT_64u" (0x"FMT_64x")",
                GET_NAME(vpe), reason, getCompare(vpe), getCount(vpe),
                COP0_FIELD(vpe, Cause, DC), delta64, delta64
            );
        }

    } else if(!atExpiry) {

        . . .
    }
}
```

Notes and Restrictions

1. If `vmirtSetICountInterrupt` is called multiple times then each call overrides the count specified with the previous call.

QuantumLeap Semantics

Non-self-synchronizing

6.4 vmirtClearICountInterrupt

Prototype

```
void vmirtClearICountInterrupt(vmiProcessorP processor);
```

Description

Function `vmirtClearICountInterrupt` clears any instruction count interrupt callback that uses the *default instruction count interrupt callback* previously enabled with `vmirtSetICountInterrupt`.

Example

This example shows the `COUNT`/`COMPARE` timer implementation from the OVP MIPS model. Note that this model implements a microthreaded SMP processor, and the timer is at the SMP group level (the VPE object in MIPS parlance).

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

static void scheduleTimerInterrupt(
    mipsP     vpe,
    const char *reason,
    Bool        atExpiry
) {
    if(!COP0_FIELD(vpe, Cause, DC)) {

        . . . .

    } else if(!atExpiry) {

        // counter disabled - no interrupt should be scheduled
        vmirtClearICountInterrupt((vmiProcessorP)vpe);

        // emit debug output if required
        if(MIPS32_DEBUGTIMER(vpe)) {
            vmiMessage(
                "I", CPU_PREFIX,
                "%s: %s (Compare=%u Count=%u CauseDC=%u) - "
                "clear timer interrupt",
                GET_NAME(vpe), reason, getCompare(vpe), getCount(vpe),
                COP0_FIELD(vpe, Cause, DC)
            );
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

6.5 vmirtCreateModelTimer

Prototype

```
vmiModelTimerP vmirtCreateModelTimer(
    vmiProcessorP processor,
    vmiICountFn   icountCB,
    Uns32         scale,
    void         *userData
);
```

Description

Function `vmirtCreateModelTimer` creates and returns a new instruction timer for the passed processor. When the timer expires, function `icountCB` is called. This *instruction count interrupt callback* is defined with the `VMI_ICOUNT_FN` macro from `vmiTypes.h` as:

```
#define VMI_ICOUNT_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,   \
    vmiModelTimerP timer,       \
    Uns64          iCount,      \
    void          *userData     \
)
typedef VMI_ICOUNT_FN((*vmiICountFn));
```

The `processor` argument is the processor on which the timer has expired. The `timer` argument is the timer object returned by `vmirtCreateModelTimer`. The `iCount` argument is the current instruction count. The `userData` argument is the value that was originally passed as the `userData` argument to `vmirtCreateModelTimer` when the timer was created.

If the callback is implementing a feature of a processor model, it should analyze the processor state and indicate that an interrupt is pending by calling `vmirtDoSynchronousInterrupt` if required. The `processor` argument to `vmirtDoSynchronousInterrupt` may be the current processor or any other processor in the simulation. Every processor subject to a synchronous interrupt call will immediately stop simulating, and the simulator will call the processor's instruction fetch handler to determine what to do next. The processor instruction fetch exception handler is responsible for arbitrating between the instruction count exception and any other pending exceptions and taking appropriate action (for example, entering kernel mode and setting the program counter to a simulated exception handler).

The timer created by `vmirtCreateModelTimer` expires when the processor has executed *exactly the number of instructions specified*. In a multiprocessor simulation, in which each processor is simulated in turn for a time slice, this can give the appearance of time moving backwards in some circumstances. See related function `vmirtCreateMonotonicModelTimer` for information on monotonic timers, which all present a consistent view of simulation time which does not appear to move backwards.

This function may also be used in intercept libraries to install periodic callbacks for analysis purposes. In this case, the callback should directly manipulate data associated with the intercept library, and not call `vmirtDoSynchronousInterrupt`.

The instruction timer returned by `vmirtCreateModelTimer` is initially disabled. It should be enabled for a specific instruction count by calling `vmirtSetModelTimer`.

The `scale` argument allows the timer rate to be specified as a fraction of the processor instruction rate. For example, a `scale` of 10 implies that the counter increments once every 10 processor instructions. A `scale` of 0 is treated as if 1 was specified (the counter will increment at every processor instruction).

Example

This example shows the CMP module watchdog timer implementation from the OVP MIPS model. Note that this model implements a microthreaded SMP processor, and the timer is at the SMP group level (the VPE object in MIPS parlance).

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

static VMI_ICOUNT_FN(iCountPendingCB) {

    . . .
}

static mipsCMPVLocalP allocCMPVPELocals(mipsP tc) {

    mipsP          vpe          = VPE_FOR_TC(tc);
    mipsCMPVLocalP cmpVPELocals = vpe->cmpVPELocals;

    if(!cmpVPELocals) {

        cmpVPELocals = STYPE_CALLOC(mipsCMPVLocal);

        // link cmpVPELocals to VPE
        vpe->cmpVPELocals = cmpVPELocals;

        . . .

        // create timer for CMP count/compare exceptions on this VPE
        cmpVPELocals->pintTimer = vmirtCreateMonotonicModelTimer(
            (vmiProcessorP)vpe, iCountPendingCB, 0, 0
        );

        // create WatchDog timer on this VPE
        cmpVPELocals->wdTimer = vmirtCreateModelTimer(
            (vmiProcessorP)vpe, wdPendingCB, 0, 0
        );
    }

    return cmpVPELocals;
}
```

Notes and Restrictions

1. Explicit model timers (created with this call) can coexist with the implicit model timer, used by the default instruction count interrupt callback (see `vmirtSetICountInterrupt`): models may have either, or both.
2. The `scale` argument exists only from VMI version 5.7.0.

3. Timers cannot be created by `vmirtCreateModelTimer` on SMP containers where any descendant is *individually schedulable*. They may only be created on *schedulable processors* or *a member of an individually-schedulable SMP group*.

QuantumLeap Semantics

Synchronizing

6.6 vmirtCreateMonotonicModelTimer

Prototype

```
vmiModelTimerP vmirtCreateMonotonicModelTimer(
    vmiProcessorP processor,
    vmiICountFn   icountCB,
    Uns32         scale,
    void         *userData
);
```

Description

Function `vmirtCreateMonotonicModelTimer` is identical to `vmirtCreateModelTimer` except that:

1. The timer created presents a *monotonically-increasing view of time*, when considered together with all other monotonic timers in a simulation; and:
2. Monotonic timers may only be created at the schedulable processor level. *They may not be created for SMP group members*.

The algorithm for determining the timer value in a multiprocessor simulation is as follows:

1. When the timer is accessed, the *implied current processor time* is determined. This is derived from the *current processor instruction count* combined with the current processor nominal *instructions-per-second* speed.
2. The *implied current processor time* is then compared with the *current monotonic minimum time*.
3. If the implied current processor time is *earlier* than the current monotonic minimum time, the timer count value is recalculated, based on the current monotonic minimum time.
4. If the implied current processor time is *later* than the current monotonic minimum time, the current monotonic minimum time is updated to the implied current processor time.

Example

This example shows the CMP module count/compare timer implementation from the OVP MIPS model. Note that this model implements a microthreaded SMP processor, and the timer is at the SMP group level (the VPE object in MIPS parlance).

```
#include “vmi/vmiRt.h”

static VMI_ICOUNT_FN(iCountPendingCB) {

    . . .
}

static mipsCMPVLocalP allocCMPVPELocals(mipsP tc) {

    mipsP          vpe          = VPE_FOR_TC(tc);
    mipsCMPVLocalP cmpVPELocals = vpe->cmpVPELocals;

    if(!cmpVPELocals) {

        cmpVPELocals = STYPE_CALLOC(mipsCMPVLocal);
```

```
        // link cmpVPELocals to VPE
        vpe->cmpVPELocals = cmpVPELocals;

        . . .

        // create timer for CMP count/compare exceptions on this VPE
        cmpVPELocals->pintTimer = vmirtCreateMonotonicModelTimer(
            (vmiProcessorP)vpe, iCountPendingCB, 0, 0
        );

        // create WatchDog timer on this VPE
        cmpVPELocals->wdTimer = vmirtCreateModelTimer(
            (vmiProcessorP)vpe, wdPendingCB, 0, 0
        );
    }

    return cmpVPELocals;
}
```

Notes and Restrictions

1. Explicit model timers (created with this call) can coexist with the implicit model timer, used by the default instruction count interrupt callback (see `vmirtSetICountInterrupt`): models may have either, or both.
2. The `scale` argument exists only from VMI version 5.7.0.
3. Timers may only be created by `vmirtCreateMonotonicModelTimer` on *individually schedulable* processors.

QuantumLeap Semantics

Synchronizing

6.7 vmirtDeleteModelTimer

Prototype

```
void vmirtDeleteModelTimer(vmiModelTimerP modelTimer);
```

Description

Function `vmirtDeleteModelTimer` deletes any timer previously created with `vmirtCreateModelTimer`. If the timer is active, it is disabled prior to deletion.

Example

This example shows the CMP module `COUNT`/`COMPARE` timer implementation from the OVP MIPS model. Note that this model implements a microthreaded SMP processor, and the timer is at the SMP group level (the VPE object in MIPS parlance).

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

static void freeCMPVPELocals(mipsP mips) {

    mipsCMPVLocalP cmpVPELocals = mips->cmpVPELocals;

    if(
        cmpVPELocals &&
        (IS_VPE(mips) || (IS_TC(mips) && (VPE_FOR_TC(mips)==mips)))
    ) {
        vmirtDeleteModelTimer(cmpVPELocals->pintTimer);
        vmirtDeleteModelTimer(cmpVPELocals->wdTimer);
        STYPE_FREE(cmpVPELocals);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

6.8 vmirtSetModelTimer

Prototype

```
void vmirtSetModelTimer(vmiModelTimerP modelTimer, Uns64 delta);
```

Description

This function causes the *instruction count interrupt callback* associated with the passed timer to be called when the *timer tick count* equals the *current timer tick count* plus the count specified with the `delta` argument. The timer tick count is the instruction count of the processor on which the timer is installed divided by the scale factor specified for the timer when `vmirtCreateModelTimer` was called. See the description of `vmirtGetICount` for information about how instructions are counted for different kinds of processor.

The instruction count interrupt callback is defined with the `VMI_ICOUNT_FN` macro and passed as the `icountCB` argument to `vmirtCreateModelTimer`. The `VMI_ICOUNT_FN` macro is defined in `vmiTypes.h` as:

```
#define VMI_ICOUNT_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,   \
    vmiModelTimerP timer,       \
    Uns64          iCount,      \
    void          *userData     \
)
typedef VMI_ICOUNT_FN((*vmiICountFn));
```

The `processor` argument is the processor on which the timer has expired. The `timer` argument is the timer previously returned by `vmirtCreateModelTimer`. The `iCount` argument is the current instruction count (note that this will be a multiple of the tick count). The `userData` argument is the value that was originally passed as the `userData` argument to `vmirtCreateModelTimer` when the timer was created.

The callback should analyze the processor state and indicate that an interrupt is pending by calling `vmirtDoSynchronousInterrupt` if required. The `processor` argument to `vmirtDoSynchronousInterrupt` may be the current processor or any other processor in the simulation. Every processor subject to a synchronous interrupt call will immediately stop simulating, and the simulator will call the processor's instruction fetch handler to determine what to do next. The processor instruction fetch exception handler is responsible for arbitrating between the instruction count exception and any other pending exceptions and taking appropriate action (for example, entering kernel mode and setting the program counter to a simulated exception handler).

Example

This example shows the CMP module `COUNT`/`COMPARE` timer implementation from the OVP MIPS model. Note that this model implements a microthreaded SMP processor, and the timer is at the SMP group level (the VPE object in MIPS parlance).

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

```
static void scheduleTimerInterrupt(
    mipsP       vpe,
    const char *reason,
    Bool        atExpiry
) {
    vmiModelTimerP modelTimer = vpe->cmpVPELocals->pintTimer;

    if(!CMPG_FIELD(vpe, GIC_SH_CONFIG, COUNTSTOP)) {

        // counter enabled - schedule the next interrupt
        Uns64 delta = getLocalCompareDelta(vpe);
        delta = !delta && atExpiry ? -1ULL : delta;

        // set CMP timer
        vmirtSetModelTimer(modelTimer, delta);

        // emit debug output if required
        if(MIPS32_DEBUGCMP(vpe)) {
            vmiMessage(
                "I", CPU_PREFIX,
                "%s: %s (GIC_VPE_Compare=0x"FMT_64x" GIC_Counter=0x"FMT_64x") - "
                "schedule CMP timer interrupt after "FMT_64u" (0x"FMT_64x")",
                GET_NAME(vpe), reason, getCompare(vpe), getCount(vpe),
                delta, delta
            );
        }

    } else if(!atExpiry) {

        . . .
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Non-self-synchronizing (*self* is processor associated with the timer when it was created)

6.9 vmirtClearModelTimer

Prototype

```
void vmirtClearModelTimer(vmiModelTimerP modelTimer);
```

Description

Function `vmirtClearModelTimer` disables the passed model timer. It will remain disabled until a subsequent call to `vmirtSetModelTimer`.

Example

This example shows the CMP module `COUNT`/`COMPARE` timer implementation from the OVP MIPS model. Note that this model implements a microthreaded SMP processor, and the timer is at the SMP group level (the VPE object in MIPS parlance).

```
#include “vmi/vmiRt.h”

static void scheduleTimerInterrupt(
    mipsP       vpe,
    const char *reason,
    Bool        atExpiry
) {
    vmiModelTimerP modelTimer = vpe->cmpVPELocals->pintTimer;

    if(!CMPG_FIELD(vpe, GIC_SH_CONFIG, COUNTSTOP)) {

        . . . .

    } else if(!atExpiry) {

        // counter disabled - no interrupt should be scheduled
        vmirtClearModelTimer(modelTimer);

        // emit debug output if required
        if(MIPS32_DEBUGCMP(vpe)) {
            vmiMessage(
                "I", CPU_PREFIX,
                "%s: %s (GIC_VPE_Compare=0x"FMT_64x" GIC_Counter=0x"FMT_64x") - "
                "clear CMP timer interrupt",
                GET_NAME(vpe), reason, getCompare(vpe), getCount(vpe)
            );
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing (*self* is processor associated with the timer when it was created)

6.10 vmirtIsModelTimerEnabled

Prototype

```
Bool vmirtIsModelTimerEnabled(vmiModelTimerP modelTimer);
```

Description

Function `vmirtIsModelTimerEnabled` returns a boolean indicating if the passed timer is currently enabled.

Example

This example shows how the OVP MIPS model uses `vmirtIsModelTimerEnabled` to set a timer to trigger a debug single-step exception if required. The timer should be set only if it is not already activated.

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

void mipsEnableDebugSingleStepException(mipsP tc) {

    if(
        debugSingleStepExceptionEnabled(tc) &&
        !vmirtIsModelTimerEnabled(tc->sstTimer)
    ) {
        vmirtSetModelTimer(tc->sstTimer, 1);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing (*self* is processor associated with the timer when it was created)

6.11 vmirtGetModelTimerCurrentCount

Prototype

```
Uns64 vmirtGetModelTimerCurrentCount(
    vmiModelTimerP modelTimer,
    Bool           ticks
);
```

Description

Function `vmirtGetModelTimerCurrentCount` returns the current value of the passed timer. If `ticks` is `False`, the returned value is the *instruction count* of the processor with which the timer is associated. If `ticks` is `True`, the returned value is the equivalent tick count, calculated by dividing the instruction count by the scale specified when the timer was created.

Example

This example shows how the OVP ARM model uses `vmirtGetModelTimerCurrentCount` when printing debug messages describing timer updates. The debug message prints both the current instruction count and the count at which an interrupt is scheduled to occur.

```
#include “vmi/vmiRt.h”

inline static Uns64 getTimerCurrentICount(vmiModelTimerP timer) {
    return vmirtGetModelTimerCurrentCount(timer, False);
}

inline static Uns64 getTimerExpiryICount(vmiModelTimerP timer) {
    return vmirtGetModelTimerExpiryCount(timer, False);
}

static void scheduleIntLT(armP arm, armLTimerP timer, const char *reason) {

    Int64 delta = instructionsToWrapLT(timer);

    delta -= timer->SR.scale;

    if(delta<=0) {
        delta += timer->SR.period;
    }

    vmirtSetModelTimer(timer->vmiTimer, delta);

    MP_INFO(arm, "_STI",
        "%s - icount="FMT_64u" schedule %s timer interrupt at "FMT_64u,
        reason,
        getTimerCurrentICount(timer->vmiTimer),
        timer->name,
        getTimerExpiryICount(timer->vmiTimer)
    );
}
```

Notes and Restrictions

1. This function exists only in VMI version 5.7.0 and above.

QuantumLeap Semantics

Non-self-synchronizing (*self* is processor associated with the timer when it was created)

6.12 vmirtGetModelTimerExpiryCount

Prototype

```
Uns64 vmirtGetModelTimerExpiryCount(
    vmiModelTimerP modelTimer,
    Bool           ticks
);
```

Description

Function `vmirtGetModelTimerExpiryCount` returns the count at which the passed timer is scheduled to expire (i.e. call the instruction count interrupt function associated with the timer). If `ticks` is `False`, the returned value is the *instruction count* at which the timer will expire. If `ticks` is `True`, the returned value is the equivalent tick count, calculated by dividing the instruction count by the scale specified when the timer was created.

Example

This example shows how the OVP ARM model uses `vmirtGetModelTimerExpiryCount` when printing debug messages describing timer updates. The debug message prints both the current instruction count and the count at which an interrupt is scheduled to occur.

```
#include “vmi/vmiRt.h”

inline static Uns64 getTimerCurrentICount(vmiModelTimerP timer) {
    return vmirtGetModelTimerCurrentCount(timer, False);
}

inline static Uns64 getTimerExpiryICount(vmiModelTimerP timer) {
    return vmirtGetModelTimerExpiryCount(timer, False);
}

static void scheduleIntLT(armP arm, armLTimerP timer, const char *reason) {

    Int64 delta = instructionsToWrapLT(timer);

    delta -= timer->SR.scale;

    if(delta<=0) {
        delta += timer->SR.period;
    }

    vmirtSetModelTimer(timer->vmiTimer, delta);

    MP_INFO(arm, "_STI",
        "%s - icount="FMT_64u" schedule %s timer interrupt at "FMT_64u,
        reason,
        getTimerCurrentICount(timer->vmiTimer),
        timer->name,
        getTimerExpiryICount(timer->vmiTimer)
    );
}
```

Notes and Restrictions

1. This function exists only in VMI version 5.7.0 and above.

QuantumLeap Semantics

Non-self-synchronizing (*self* is processor associated with the timer when it was created)

7 Simulated Memory Access

The VMI morph time interface (described in `vmiMt.h`) allows simple memory accesses such as load and store instructions to be modeled very efficiently by translated native code. See the *VMI Morph Time Function Reference* for more details about this.

Sometimes the required behavior is more complex and cannot easily or efficiently be modeled using the morph time interface. For example, a processor might implement a single instruction that is able to move many bytes of data in memory – for example, the x86 `movs` instruction. In this case, run time functions for modeling memory accesses are available, described in this section.

7.1 vmirtReadNByteDomain

Prototype

```
memMapped vmirtReadNByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    void          *buffer,
    Addr           size,
    memRegionPP    cachedRegion,
    memAccessAttrs attrs
);
```

Description

This function reads `size` bytes starting at address `simAddress` from the passed domain into the buffer. Typically, this will be called from an embedded function call (created by `vmimtCall` or similar at morph time).

If `cachedRegion` is non-`NULL`, then it should point to a *memory region cache*, typically in the processor structure. This region cache is an opaque type, `memRegionP`, which can be used in combination with `vmirtGetReadNByteSrc` for faster access when the same area of memory is used by successive reads.

The `attrs` argument is an enumeration defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

If the value is `MEM_AA_TRUE`, then this is a *true processor read/write* that should generate a call to the processor simulated exception handler on an access violation. For example, if a function has been written to simulate a processor block move instruction, then this is the value that should be used.

If the value is `MEM_AA_FALSE`, then this is an *artifact read/write* and not a true processor access. For example, if data is being accessed to semi-host some system function, this value should be used.

If the value is `MEM_AA_TRUE|MEM_AA_FETCH`, then this is a *true processor fetch* that should generate a call to the processor simulated exception handler on an access violation.

If the value is `MEM_AA_FALSE|MEM_AA_FETCH`, then this is an *artifact fetch* and not a true processor access (usually, an access made by the JIT code generator).

Additionally, the argument controls the value passed as the `processor` argument to read callback functions: see `vmirtAddReadCallback` for more details.

The return value of the function is a member of the `memMapped` enumeration:

```
typedef enum memMappedE {
    MEM_MAP_NONE = 0,               // no addresses in the range mapped
    MEM_MAP_PART = 1,               // some addresses in the range mapped
    MEM_MAP_FULL = 2,               // all addresses in the range mapped
} memMapped;
```

If no address in the range is readable, the function returns `MEM_MAP_NONE`; if all addresses in the range are readable, the function returns `MEM_MAP_FULL`; otherwise, the function returns `MEM_MAP_PART`.

Example

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

// processor structure definition
typedef struct x86S {
    memRegionP srcRegion;       // source region cache
    memRegionP dstRegion;       // destination region cache
} x86, *x86P;

static void doMovSCommon(
    x86P  x86,
    Uns32 fromAddress,
    Uns32 toAddress,
    Uns32 bytesToMove
) {
    Uns8 *src;
    Uns8 *dst;

    if(
        (src=vmirtGetReadNByteSrc(x86->srcRegion,fromAddress,bytesToMove,True)) &&
        (dst=vmirtGetWriteNByteDst(x86->dstRegion,toAddress,bytesToMove,True))
    ) {

        // optimized move between the same regions as previously
        memcpy(dst, src, bytesToMove);

    } else {

        // case requiring an intermediate buffer
        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)x86);
        Uns8       buffer[bytesToMove];

        vmirtReadNByteDomain(
            domain, fromAddress, buffer, bytesToMove, &x86->srcRegion, MEM_AA_TRUE
        );
        vmirtWriteNByteDomain(
            domain, toAddress, buffer, bytesToMove, &x86->dstRegion, MEM_AA_TRUE
        );
    }
}
```

Notes and Restrictions

1. There is no automatic validation that the target buffer is large enough to hold `size` bytes. It is the calling function's responsibility to verify this.
2. If the value size is 1, 2, 4 or 8 bytes, use the appropriate `vmirtRead[1248]ByteDomain` function instead (which also allows the endianness of the value in memory to be specified).
3. `memAccessAttrs` value `MEM_AA_IGNORE_PRIV` previously supported by the VMI API is now deprecated and redefined to be equivalent to `MEM_AA_TRUE`.

QuantumLeap Semantics
Synchronizing

7.2 vmirtReadNByteDomainVM

Prototype

```
memMapped vmirtReadNByteDomainVM(
    memDomainP     domain,
    Addr           simAddress,
    void          *buffer,
    Addr           size,
    memRegionPP    cachedRegion,
    Addr           VA,
    memAccessAttrs attrs
);
```

Description

This function is identical to `vmirtReadNByteDomain`, except that it additionally takes a nominal virtual address argument (`VM`). If callbacks of type `vmiMemReadFn` or `vmiMemWatchFn` (or equivalent platform callbacks) are activated by this call, then this nominal virtual address will be passed as the `VM` argument to those callbacks. By contrast, `vmirtReadNByteDomain` always passes the same value as both `address` and `VA` to such callbacks.

This function is intended to be used when structures such as cache models are implemented using the VMI API. It allows virtual addresses reported as the `VA` argument of a function of type `vmiMemReadFn` that implements the cache to passed on to structures beyond the cache in the memory subsystem.

For details of other arguments to this function, see notes for function `vmirtReadNByteDomain`.

Example

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

static void readMemTrue(
    cacheInfoP cInfo,
    Addr       pa,
    Addr       va,
    void      *data,
    Uns32      nBytes,
    Bool       isFetch
) {
    memAccessAttrs attrs;
    memDomainP     domain;
    memRegionPP    rCache;

    if(isFetch) {
        attrs  = MEM_AA_TRUE | MEM_AA_FETCH;
        domain = cInfo->cacheBackDomain.d[MPT_CODE];
        rCache = &cInfo->rCacheCode;
    } else {
        attrs  = MEM_AA_TRUE;
        domain = cInfo->cacheBackDomain.d[MPT_DATA];
        rCache = &cInfo->rCacheData;
    }

    vmirtReadNByteDomainVA(domain, pa, data, nBytes, rCache, va, attrs);
}
```

Notes and Restrictions
See notes for function `vmirtReadNByteDomain`.

QuantumLeap Semantics
Synchronizing

7.3 vmirtWriteNByteDomain

Prototype

```
memMapped vmirtWriteNByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    const void    *buffer,
    Addr           size,
    memRegionPP    cachedRegion,
    memAccessAttrs attrs
);
```

Description

This function writes `size` bytes to address `simAddress` in the passed domain from the buffer. Typically, this will be called from an embedded function call (created by `vmimtCall` or similar at morph time).

If `cachedRegion` is non-`NULL`, then it should point to a *memory region cache*, typically in the processor structure. This region cache is an opaque type, `memRegionP`, which can be used in combination with `vmirtGetWriteNByteDst` for faster access when the same area of memory is used by successive writes.

The `attrs` argument is an enumeration defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

If the value is `MEM_AA_TRUE`, then this is a *true processor read/write* that should generate a call to the processor simulated exception handler on an access violation. For example, if a function has been written to simulate a processor block move instruction, then this is the value that should be used.

If the value is `MEM_AA_FALSE`, then this is an *artifact read/write* and not a true processor access. For example, if data is being accessed to semi-host some system function, this value should be used.

If the value is `MEM_AA_TRUE|MEM_AA_FETCH`, then this is a *true processor fetch* that should generate a call to the processor simulated exception handler on an access violation.

If the value is `MEM_AA_FALSE|MEM_AA_FETCH`, then this is an *artifact fetch* and not a true processor access (usually, an access made by the JIT code generator).

Additionally, the argument controls the value passed as the `processor` argument to write callback functions: see `vmirtAddWriteCallback` for more details.

The return value of the function is a member of the `memMapped` enumeration:

```
typedef enum memMappedE {
    MEM_MAP_NONE = 0,               // no addresses in the range mapped
    MEM_MAP_PART = 1,               // some addresses in the range mapped
    MEM_MAP_FULL = 2,               // all addresses in the range mapped
} memMapped;
```

If no address in the range is writable, the function returns `MEM_MAP_NONE`; if all addresses in the range are writable, the function returns `MEM_MAP_FULL`; otherwise, the function returns `MEM_MAP_PART`.

Example

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

// processor structure definition
typedef struct x86S {
    memRegionP srcRegion;       // source region cache
    memRegionP dstRegion;       // destination region cache
} x86, *x86P;

static void doMovSCommon(
    x86P  x86,
    Uns32 fromAddress,
    Uns32 toAddress,
    Uns32 bytesToMove
) {
    Uns8 *src;
    Uns8 *dst;

    if(
        (src=vmirtGetReadNByteSrc(x86->srcRegion,fromAddress,bytesToMove,True)) &&
        (dst=vmirtGetWriteNByteDst(x86->dstRegion,toAddress,bytesToMove,True))
    ) {

        // optimized move between the same regions as previously
        memcpy(dst, src, bytesToMove);

    } else {

        // case requiring an intermediate buffer
        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP) x86);
        Uns8       buffer[bytesToMove];

        vmirtReadNByteDomain(
            domain, fromAddress, buffer, bytesToMove, &x86->srcRegion, MEM_AA_TRUE
        );
        vmirtWriteNByteDomain(
            domain, toAddress, buffer, bytesToMove, &x86->dstRegion, MEM_AA_TRUE
        );
    }
}
```

Notes and Restrictions

1. There is no automatic validation that the source buffer is large enough to hold `size` bytes. It is the calling function's responsibility to verify this.
2. If the value size is 1, 2, 4 or 8 bytes, use the appropriate `vmirtWrite[1248]ByteDomain` function instead (which also allows the endianness of the value in memory to be specified).
3. `memAccessAttrs` value `MEM_AA_IGNORE_PRIV` previously supported by the VMI API is now deprecated and redefined to be equivalent to `MEM_AA_TRUE`.

QuantumLeap Semantics
Synchronizing

7.4 vmirtWriteNByteDomainVM

Prototype

```
memMapped vmirtWriteNByteDomainVM(
    memDomainP     domain,
    Addr           simAddress,
    const void    *buffer,
    Addr           size,
    memRegionPP    cachedRegion,
    Addr           VM,
    memAccessAttrs attrs
);
```

Description

This function is identical to `vmirtWriteNByteDomain`, except that it additionally takes a nominal virtual address argument (`VM`). If callbacks of type `vmiMemWriteFn` or `vmiMemWatchFn` (or equivalent platform callbacks) are activated by this call, then this nominal virtual address will be passed as the `VM` argument to those callbacks. By contrast, `vmirtWriteNByteDomain` always passes the same value as both `address` and `VA` to such callbacks.

This function is intended to be used when structures such as cache models are implemented using the VMI API. It allows virtual addresses reported as the `VA` argument of a function of type `vmiMemWriteFn` that implements the cache to passed on to structures beyond the cache in the memory subsystem.

For details of other arguments to this function, see notes for function `vmirtWriteNByteDomain`.

Example

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

static void writeMemTrue(
    cacheInfoP  cInfo,
    Addr        pa,
    Addr        va,
    const void *data,
    Uns32       nBytes
) {
    memAccessAttrs attrs  = MEM_AA_TRUE;
    memDomainP     domain = cInfo->cacheBackDomain.d[MPT_DATA];
    memRegionPP    rCache = &cInfo->rCacheData;

    vmirtWriteNByteDomainVA(domain, pa, data, nBytes, rCache, va, attrs);
}
```

Notes and Restrictions

See notes for function `vmirtWriteNByteDomain`.

QuantumLeap Semantics

Synchronizing

7.5 vmirtRead[1248]ByteDomain

Prototypes

```
Uns8 vmirtRead1ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memAccessAttrs attrs
);
Uns16 vmirtRead2ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    memAccessAttrs attrs
);
Uns32 vmirtRead4ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    memAccessAttrs attrs
);
Uns64 vmirtRead8ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    memAccessAttrs attrs
);
```

Description

These four routines read (respectively) 1, 2, 4 and 8 byte data words from the passed address in the passed memory domain. For the 2, 4 and 8 byte versions, the read is performed with the specified endianness.

The `attrs` argument is an enumeration defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

If the value is `MEM_AA_TRUE`, then this is a *true processor read/write* that should generate a call to the processor simulated exception handler on an access violation. For example, if a function has been written to simulate a processor block move instruction, then this is the value that should be used.

If the value is `MEM_AA_FALSE`, then this is an *artifact read/write* and not a true processor access. For example, if data is being accessed to semi-host some system function, this value should be used.

If the value is `MEM_AA_TRUE|MEM_AA_FETCH`, then this is a *true processor fetch* that should generate a call to the processor simulated exception handler on an access violation.

If the value is MEM_AA_FALSE|MEM_AA_FETCH, then this is an *artifact fetch* and not a true processor access (usually, an access made by the JIT code generator).

Additionally, the argument controls the value passed as the processor argument to read callback functions: see vmirtAddReadCallback for more details.

These functions are very useful in intercept libraries since they allow intercept code to be written that is independent of processor endianness (see example).

Example

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

// Read 4 byte word written by processor at the passed address
Uns32 read4Byte(processorP processor, Addr simAddress) {

    memEndian  endian = vmirtGetProcessorDataEndian(processor);
    memDomainP domain = vmirtGetProcessorDataDomain(processor);

    return vmirtRead4ByteDomain(domain, simAddress, endian, MEM_AA_FALSE);
}
```

Notes and Restrictions

1. In general, routines from the morph time interface (defined in the *VMI Morph Time Function Reference*) should be used in preference to these functions in processor models wherever possible because they are much faster. As an example, refer see function vmimtLoadRRO which generates optimized code to perform a memory read.
2. memAccessAttrs value MEM_AA_IGNORE_PRIV previously supported by the VMI API is now deprecated and redefined to be equivalent to MEM_AA_TRUE.

QuantumLeap Semantics

Synchronizing

7.6 vmirtWrite[1248]ByteDomain

Prototypes

```
void vmirtWrite1ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    Uns8           value,
    memAccessAttrs attrs
);
void vmirtWrite2ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    Uns16          value,
    memAccessAttrs attrs
);
void vmirtWrite4ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    Uns32          value,
    memAccessAttrs attrs
);
void vmirtWrite8ByteDomain(
    memDomainP     domain,
    Addr           simAddress,
    memEndian      endian,
    Uns64          value,
    memAccessAttrs attrs
);
```

Description

These four routines write (respectively) 1, 2, 4 and 8 byte data words to the passed address in the passed memory domain. For the 2, 4 and 8 byte versions, the write is performed with the specified endianness.

The `attrs` argument is an enumeration defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

If the value is `MEM_AA_TRUE`, then this is a *true processor read/write* that should generate a call to the processor simulated exception handler on an access violation. For example, if a function has been written to simulate a processor block move instruction, then this is the value that should be used.

If the value is `MEM_AA_FALSE`, then this is an *artifact read/write* and not a true processor access. For example, if data is being accessed to semi-host some system function, this value should be used.

If the value is MEM_AA_TRUE|MEM_AA_FETCH, then this is a *true processor fetch* that should generate a call to the processor simulated exception handler on an access violation.

If the value is MEM_AA_FALSE|MEM_AA_FETCH, then this is an *artifact fetch* and not a true processor access (usually, an access made by the JIT code generator).

Additionally, the argument controls the value passed as the processor argument to write callback functions: see vmirtAddWriteCallback for more details.

These functions are very useful in intercept libraries since they allow intercept code to be written that is independent of processor endianness (see example).

Example

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

// Write 4 byte word at the passed address in appropriate endianness
void write4Byte(processorP processor, Addr simAddress, Uns32 value) {

    memEndian  endian = vmirtGetProcessorDataEndian(processor);
    memDomainP domain = vmirtGetProcessorDataDomain(processor);

    vmirtWrite4ByteDomain(domain, simAddress, endian, value, MEM_AA_FALSE);
}
```

Notes and Restrictions

1. In general, routines from the morph time interface (defined in the *VMI Morph Time Function Reference*) should be used in preference to these functions in processor models wherever possible because they are much faster. As an example, refer see function vmimtStoreRRO which generates optimized code to perform a memory write.
2. memAccessAttrs value MEM_AA_IGNORE_PRIV previously supported by the VMI API is now deprecated and redefined to be equivalent to MEM_AA_TRUE.

QuantumLeap Semantics

Synchronizing

7.7 vmirtGetReadNByteSrc

Prototype

```
void *vmirtGetReadNByteSrc(
    memRegionP region,
    Addr       simLow,
    Addr       size,
    Bool       trueAccess
);
```

Description

Given a memory region, this function determines whether the addresses from `simLow` to `simLow+size-1` are represented in that region and are readable using native C functions (for example, `memcpy`). If so, it returns a pointer to the native memory from which the data bytes can be read. If not, it returns `NULL`.

The purpose of this routine is to accelerate memory reads where the same region of memory is read repeatedly. The memory region (represented by the opaque type `memRegionP`) is normally a cache variable in the processor structure. It is updated by a call to `vmirtReadNByteDomain`.

The returned pointer has limited lifetime: it is invalidated by any subsequent call that may modify the simulator's simulated memory mapping. In particular, any call to the following functions will cause invalidation:

```
vmirtReadNByteDomain
vmirtWriteNByteDomain
vmirtGetString
vmirtAliasMemory
vmirtUnaliasMemory
vmirtProtectMemory
vmirtAddReadCallback
vmirtAddWriteCallback
vmirtAddFetchCallback
vmirtRemoveReadCallback
vmirtRemoveWriteCallback
vmirtRemoveFetchCallback
```

The `trueAccess` argument indicates whether this function is being called to model actual processor behavior (`True`) or an artifact of simulation (`False`). For example, if the call is being used to model a block move instruction, the argument should be `True`; if it is being called to move data in order to semi-host some system function, the argument should be `False`. This information is required when the processor model is simulated in conjunction with memory model components such as caches: only transactions corresponding to actual processor behavior should modify the cache state.

Example

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

// processor structure definition
typedef struct x86S {
    memRegionP srcRegion;       // source region cache
    memRegionP dstRegion;       // destination region cache
```

```
} x86, *x86P;

static void doMovSCommon(
    x86P  x86,
    Uns32 fromAddress,
    Uns32 toAddress,
    Uns32 bytesToMove
) {
    Uns8 *src;
    Uns8 *dst;

    if(
        (src=vmirtGetReadNByteSrc(x86->srcRegion,fromAddress,bytesToMove,True)) &&
        (dst=vmirtGetWriteNByteDst(x86->dstRegion,toAddress,bytesToMove,True))
    ) {

        // optimized move between the same regions as previously
        memcpy(dst, src, bytesToMove);

    } else {

        // unoptimized case requiring an intermediate buffer
        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP) x86);
        Uns8       buffer[bytesToMove];

        vmirtReadNByteDomain(
            domain, fromAddress, buffer, bytesToMove, &x86->srcRegion, MEM_AA_TRUE
        );
        vmirtWriteNByteDomain(
            domain, toAddress, buffer, bytesToMove, &x86->dstRegion, MEM_AA_TRUE
        );
    }
}
```

Notes and Restrictions

1. If `vmirtGetReadNByteSrc` returns `NULL` it does not imply that memory is not readable: it just means that the address range does not lie in the cached region and a call to the slow read routine `vmirtReadNByteDomain` is required to perform the read. This routine merely improves performance for typical accesses.
2. Models must take great care not to use the pointer returned by this function beyond its valid lifetime. To do so may cause obscure errors or simulator crashes.

QuantumLeap Semantics

Synchronizing

7.8 vmirtGetWriteNByteDst

Prototype

```
void *vmirtGetWriteNByteDst(
    memRegionP region,
    Addr       simLow,
    Addr       size,
    Bool       trueAccess
);
```

Description

Given a memory region, this function determines whether the addresses from `simLow` to `simLow+size-1` are represented in that region and are writable using native C functions (for example, `memcpy`). If so, it returns a pointer to the native memory to which the data bytes can be written. If not, it returns `NULL`.

The purpose of this routine is to accelerate memory reads where the same region of memory is written repeatedly. The memory region (represented by the opaque type `memRegionP`) is normally a cache variable in the processor structure, updated by a call to `vmirtWriteNByteDomain`.

The returned pointer has limited lifetime: it is invalidated by any subsequent call that may modify the simulator's simulated memory mapping. In particular, any call to the following functions will cause invalidation:

```
vmirtReadNByteDomain
vmirtWriteNByteDomain
vmirtGetString
vmirtAliasMemory
vmirtUnaliasMemory
vmirtProtectMemory
vmirtAddReadCallback
vmirtAddWriteCallback
vmirtAddFetchCallback
vmirtRemoveReadCallback
vmirtRemoveWriteCallback
vmirtRemoveFetchCallback
```

The `trueAccess` argument indicates whether this function is being called to model actual processor behavior (`True`) or an artifact of simulation (`False`). For example, if the call is being used to model a block move instruction, the argument should be `True`; if it is being called to move data in order to semi-host some system function, the argument should be `False`. This information is required when the processor model is simulated in conjunction with memory model components such as caches: only transactions corresponding to actual processor behavior should modify the cache state.

Example

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

// processor structure definition
typedef struct x86S {
    memRegionP srcRegion;      // source region cache
    memRegionP dstRegion;      // destination region cache
```

```
} x86, *x86P;

static void doMovSCommon(
    x86P  x86,
    Uns32 fromAddress,
    Uns32 toAddress,
    Uns32 bytesToMove
) {
    Uns8 *src, *dst;

    if(
        (src=vmirtGetReadNByteSrc(x86->srcRegion,fromAddress,bytesToMove,True)) &&
        (dst=vmirtGetWriteNByteDst(x86->dstRegion,toAddress,bytesToMove,True))
    ) {

        // optimized move between the same regions as previously
        memcpy(dst, src, bytesToMove);

    } else {

        // unoptimized case requiring an intermediate buffer
        memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)x86);
        Uns8       buffer[bytesToMove];

        vmirtReadNByteDomain(
            domain, fromAddress, buffer, bytesToMove, &x86->srcRegion, MEM_AA_TRUE
        );
        vmirtWriteNByteDomain(
            domain, toAddress, buffer, bytesToMove, &x86->dstRegion, MEM_AA_TRUE
        );
    }
}
```

Notes and Restrictions

1. If `vmirtGetWriteNByteDst` returns `NULL` it does not imply that memory is not writable: it just means that the address range does not lie in the cached region and a call to the slow write routine `vmirtWriteNByteDomain` is required to perform the write. This routine merely improves performance for typical accesses.
2. Models must take great care not to use the pointer returned by this function beyond its valid lifetime. To do so may cause obscure errors or simulator crashes.

QuantumLeap Semantics

Synchronizing

7.9 vmirtGetString

Prototype

```
char *vmirtGetString(memDomainP domain, Addr simAddress);
```

Description

This routine returns a pointer to a `NULL`-terminated string starting at address `simAddress` in the passed memory domain. A typical use is for implementing processor pseudo-opcodes which provide debug text output capability.

The returned pointer has limited lifetime: it is invalidated by any subsequent call that may modify the simulator's simulated memory mapping. In particular, any call to the following functions will cause invalidation:

```
vmirtReadNByteDomain
vmirtWriteNByteDomain
vmirtGetString
vmirtAliasMemory
vmirtUnaliasMemory
vmirtProtectMemory
vmirtAddReadCallback
vmirtAddWriteCallback
vmirtAddFetchCallback
vmirtRemoveReadCallback
vmirtRemoveWriteCallback
vmirtRemoveFetchCallback
```

Example

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

#define CPUX_GREGS 32       // number of GPRs

// processor structure definition
typedef struct cpuxS {
    Uns32 regs[CPUX_GREGS]; // 32-bit GPRs
} cpux, *cpuxP;

static void vmic_Printf(cpuxP cpux) {

    memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);

    // R0 has the address of the format string in simulated memory
    Addr r0 = cpux->regs[0];
    char *fmt = vmirtGetString(domain, r0);

    // handle case of uninitialized memory
    if(!fmt) fmt = "";

    // R1…R5 are the optional parameters
    vmiPrintf(
        fmt,
        cpux->regs[1],
        cpux->regs[2],
        cpux->regs[3],
        cpux->regs[4],
        cpux->regs[5]
    );
}
```

Notes and Restrictions

1. If the simulated address passed to `vmirtGetString` is un-initialized, the function will return `NULL`.
2. Apart from the un-initialized memory check, there is no automatic test that the addressed memory contains a valid string: it is up to the application and the model to validate this.

QuantumLeap Semantics

Synchronizing

8 Simulated Memory Management

The `vmiRt.h` interface defines functions to allow simulation of MMUs and related aspects of virtual memory management. This section describes these functions.

A key concept in simulated memory management modeling is the *memory domain*. A memory domain describes an address space and related properties such as access permissions.

At initialization, simulated processors are provided with a physical *code domain* (from where instructions should be fetched) and a physical *data domain* (from which data should be read and written). For many processors, code and data domains are identical. The processor constructor may create various further *virtual domains* as required to efficiently simulate an MMU. For example, a processor may create a virtual *kernel domain* and a virtual *user domain*, for simulation of kernel and user mode accesses respectively. Within each virtual domain, regions of memory may be dynamically mapped and unmapped, and also access permissions may be updated. The simulator efficiently validates mappings and access permissions and calls the *memory access exception handlers* defined in the processor attribute structure for invalid accesses.

Memory domains are closely related to code dictionaries, described in the *VMI Morph Time Function Reference*. For a modal processor (for example, one with user and kernel modes) there is typically a memory domain associated with each mode. A processor mode switch will require both the current data and code domain and the current code dictionary to be switched. An outline template showing how to describe a processor with memory and code domains is shown below.

Processor Attributes

The processor attribute structure should provide names for the dictionaries in use by the processor. This name list is a `NULL`-terminated list of strings. For example:

```
// this model has two modes: kernel (mode 0) and user (mode 1).
static const char *dictNames[] = {"KERNEL", "USER", 0};

//
// Configuration block for instruction-accurate modeling
//
const vmiIASAttr cpuxIASAttrs = {

    ////////////////////////////////////////////////////////////////////////
    // VERSION & SIZE ATTRIBUTES
    ////////////////////////////////////////////////////////////////////////

    .versionString = VMI_VERSION,
    .dictNames     = dictNames,
    . . . etc . . .
}
```

Virtual Memory Constructor

The virtual memory constructor is defined with the `VMI_VMINIT_FN` macro and supplied as the `vmInitCB` field of the processor attributes structure. It is passed a pointer to an array of code domains and a pointer to an array of data domains. Each array has the same number of entries as the number of dictionary names in the `dictNames` field provided in the processor attributes structure (in this example, each array will have two entries). By default, all entries in the code array are initialized to the physical code domain derived from the platform and all entries in the data array are initialized to the physical data domain derived from the platform. The constructor may override these defaults with virtual domains if virtual memory management using an MMU is to be simulated.

```
typedef enum cpuxModeE {
    CPM_KERNEL = 0,
    CPM_USER   = 1
} cpuxMode;

#define PA_LOW_ADDR  0x0000000000ULL
#define PA_HIGH_ADDR 0x1FFFFFFFFFULL
#define KM_ADDR      0x1000000000000ULL

// Processor VM initialization routine. Arguments:
//     processor:   the current processor
//     codeDomains: array of 2 code domains
//     dataDomains: array of 2 data domains
VMI_VMINIT_FN(cpuxVMInit)
{
    cpuxP cpux = (cpuxP)processor;

    memDomainP physicalCodeDomain = codeDomains[0];
    memDomainP physicalDataDomain = dataDomains[0];
    memDomainP kernelDomain;
    memDomainP userDomain;

    ASSERT(
        physicalCodeDomain==physicalDataDomain,
        "expected code & data domains to match"
    );

    // save the physical domain in the processor structure for later use
    cpux->physicalDomain = physicalCodeDomain;

    // define new 56-bit virtual address spaces
    kernelDomain = vmirtNewDomain("kernel", 56);
    userDomain   = vmirtNewDomain("user"    56);

    // set up direct mapped image of physical memory range PA_LOW_ADDR:
    // PA_HIGH_ADDR at virtual address KM_ADDR in physical domain
    vmirtAliasMemory(
        physicalCodeDomain,     // physical domain
        kernelDomain,           // virtual domain
        PA_LOW_ADDR,            // low physical address
        PA_HIGH_ADDR,           // high physical address
        KM_ADDR,                // low virtual address
        0                       // no MRU code management
    );

    // specify the domains to use with this processor
    codeDomains[CPM_KERNEL] = kernelDomain;
    dataDomains[CPM_KERNEL] = kernelDomain;
    codeDomains[CPM_USER]   = userDomain;
    dataDomains[CPM_USER]   = userDomain;
}
```

Mode Switch Instructions
When processing an instruction that causes a mode switch, the translated native code should be constructed to contain an embedded call to a function that does the mode switch. The function will do this by calling `vmirtSetMode`, described in a later section. When the mode is switched, the current processor data and code domains will be updated to the domains with matching index in the `codeDomains` and `dataDomains` arrays. For example, a switch to mode 0 will set the current data domain to `dataDomains[0]` and a switch to mode 1 will set the current data domain to `dataDomains[1]`.

```
static void vmic_SwitchMode(cpuxP cpux, cpuxMode newMode) {

    if(cpux->mode!=newMode) {
        // switch to the new operating mode – this will change
        // dictionary and domain.
        vmirtSetMode((vmiProcessorP)cpux, newMode);
        cpux->mode = newMode;
    }
}
```

Non-Paged and Paged Mappings
There are two distinct types of memory mappings that are established between virtual and physical memory domains: *non-paged* and *paged*.

Non-paged mappings represent translations that are not managed by structures such as TLBs: they are simple aliases of an address range in one domain to an address range in another. They have the following characteristics:

1. The aliases can be of any size, even down to a single byte;
2. They can be hierarchical: a non-paged mapping can be made to an address range in a domain that is itself a non-paged mapping to a range in another domain, and so on;
3. They are relatively static at run time: there is typically some effort in the processor virtual memory constructor to establish the non-paged mappings, which then usually persist for the simulation;
4. They have global scope and are not dependent on processor ASID (*address space id*).

Paged mappings represent virtual memory translations managed by structures such as TLBs. They have the following characteristics:

1. They are typically a multiple of some basic page size (1K, 4K etc);
2. They are not hierarchical: memory mapping cannot be made to a page-mapped region;
3. They are very dynamic at run time: mappings come and go rapidly as the operating system runs;
4. They can have global scope or be limited to a particular processor ASID.

These functions should be used when managing non-paged mappings:

```
vmirtAliasMemory
vmirtUnaliasMemory
```

These functions should be used when managing paged mappings:

```
vmirtSetProcessorASID
vmirtGetProcessorASID
vmirtAliasMemoryVM
vmirtUnaliasMemoryVM
vmirtGetDomainMappedASID
vmirtGetMRUStateTable
vmirtGetNthStateIndex
```

8.1 vmirtGetProcessorCodeDomain

Prototype

```
memDomainP vmirtGetProcessorCodeDomain(vmiProcessorP processor);
```

Description

This routine returns the current code domain for a processor. The code domain is the source of instruction fetches.

Example

```
#include “vmi/vmiRt.h”

static void vmic_MapCodeDomain(cpuxP cpux, Uns32 paLow, Uns32 paHigh, Uns32 va) {

    memDomainP domainP = vmirtGetProcessorPhysicalCodeDomain((vmiProcessorP)cpux);
    memDomainP domainV = vmirtGetProcessorCodeDomain((vmiProcessorP)cpux);

    vmirtAliasMemory(
        domainP,                // physical domain
        domainV,                // virtual domain
        paLow,                  // low physical address
        paHigh,                 // high physical address
        va,                     // low virtual address
        0                       // (no MRU management)
    );
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

8.2 vmirtGetProcessorDataDomain

Prototype

```
memDomainP vmirtGetProcessorDataDomain(vmiProcessorP processor);
```

Description

This routine returns the current data domain for a processor. The data domain is used for all load and store instructions.

Example

```
#include “vmi/vmiRt.h”

static void vmic_MapDataDomain(cpuxP cpux, Uns32 paLow, Uns32 paHigh, Uns32 va) {

    memDomainP domainP = vmirtGetProcessorPhysicalDataDomain((vmiProcessorP)cpux);
    memDomainP domainV = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);

    vmirtAliasMemory(
        domainP,                // physical domain
        domainV,                // virtual domain
        paLow,                  // low physical address
        paHigh,                 // high physical address
        va,                     // low virtual address
        0                       // (no MRU management)
    );
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

8.3 vmirtGetProcessorPhysicalCodeDomain

Prototype

```
memDomainP vmirtGetProcessorPhysicalCodeDomain(vmiProcessorP processor);
```

Description

This routine returns the *physical* code domain for a processor. The physical code domain is the domain associated with the externally-connected instruction bus. Typically, this function will be used when establishing address mappings from model-defined domains (as in the example below) or in intercept libraries that monitor instruction fetches.

Example

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

static void vmic_MapCodeDomain(cpuxP cpux, Uns32 paLow, Uns32 paHigh, Uns32 va) {

    memDomainP domainP = vmirtGetProcessorPhysicalCodeDomain((vmiProcessorP)cpux);
    memDomainP domainV = vmirtGetProcessorCodeDomain((vmiProcessorP)cpux);

    vmirtAliasMemory(
        domainP,                // physical domain
        domainV,                // virtual domain
        paLow,                  // low physical address
        paHigh,                 // high physical address
        va,                     // low virtual address
        0                       // (no MRU management)
    );
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

8.4 vmirtGetProcessorPhysicalDataDomain

Prototype

```
memDomainP vmirtGetProcessorPhysicalDataDomain(vmiProcessorP processor);
```

Description

This routine returns the *physical* data domain for a processor. The physical data domain is the domain associated with the externally-connected data bus. Typically, this function will be used when establishing address mappings from model-defined domains (as in the example below) or in intercept libraries that monitor loads and stores.

Example

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

static void vmic_MapDataDomain(cpuxP cpux, Uns32 paLow, Uns32 paHigh, Uns32 va) {

    memDomainP domainP = vmirtGetProcessorPhysicalDataDomain((vmiProcessorP)cpux);
    memDomainP domainV = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);

    vmirtAliasMemory(
        domainP,                // physical domain
        domainV,                // virtual domain
        paLow,                  // low physical address
        paHigh,                 // high physical address
        va,                     // low virtual address
        0                       // (no MRU management)
    );
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

8.5 vmirtGetProcessorInternalCodeDomain

Prototype

```
memDomainP vmirtGetProcessorInternalCodeDomain(vmiProcessorP processor);
```

Description

This routine returns the *internal* code domain for a processor. The internal code domain is a model-defined memory domain, specified using function `vmirtSetProcessorInternalCodeDomain` in the processor virtual memory constructor.

The purpose of the internal code domain is to allow intercept libraries and platform fetch callbacks to monitor raw physical fetch accesses *before* such accesses are modified by structures such as tightly-coupled memories, memory-mapped system register blocks and caches that sit between the processor core and the externally-connected instruction bus.

Example

This example shows how an intercept library can be written that monitors read, write and fetch accesses on the processor internal data and code domains.

```
#include “vmi/vmiRt.h”

//
// Report a processor access
//
static void report(
    const char   *desc,
    vmiProcessorP processor,
    Uns32         bytes,
    const Uns8   *value
) {
    // report only non-artifact accesses
    if(processor) {

        Uns32 i;

        vmiPrintf("%s: %s [", vmirtProcessorName(processor), desc);

        for(i=0; i<bytes; i++) {
            vmiPrintf("%02x", value[i]);
        }

        vmiPrintf("]\n");
    }
}

//
// Memory callback
//
static VMI_MEM_WATCH_FN(watchCB) {
    report(userData, processor, bytes, value);
}

//
// Constructor
//
VMIOS_CONSTRUCTOR_FN(constructor) {

    // get internal code and data domains
    memDomainP ICDomain = vmirtGetProcessorInternalCodeDomain(processor);
    memDomainP IDDomain = vmirtGetProcessorInternalDataDomain(processor);
```

```
    // add internal callbacks
    vmirtAddFetchCallback(ICDomain, processor, 0, -1, watchCB, "IFETCH");
    vmirtAddWriteCallback(IDDomain, processor, 0, -1, watchCB, "IWRITE");
    vmirtAddReadCallback (IDDomain, processor, 0, -1, watchCB, "IREAD ");
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

8.6 vmirtGetProcessorInternalDataDomain

Prototype

```
memDomainP vmirtGetProcessorInternalDataDomain(vmiProcessorP processor);
```

Description

This routine returns the *internal* data domain for a processor. The internal data domain is a model-defined memory domain, specified using function `vmirtSetProcessorInternalDataDomain` in the processor virtual memory constructor.

The purpose of the internal data domain is to allow intercept libraries and platform fetch callbacks to monitor raw physical fetch accesses *before* such accesses are modified by structures such as tightly-coupled memories, memory-mapped system register blocks and caches that sit between the processor core and the externally-connected instruction bus.

Example

This example shows how an intercept library can be written that monitors read, write and fetch accesses on the processor internal data and code domains.

```
#include “vmi/vmiRt.h”

//
// Report a processor access
//
static void report(
    const char   *desc,
    vmiProcessorP processor,
    Uns32         bytes,
    const Uns8   *value
) {
    // report only non-artifact accesses
    if(processor) {

        Uns32 i;

        vmiPrintf("%s: %s [", vmirtProcessorName(processor), desc);

        for(i=0; i<bytes; i++) {
            vmiPrintf("%02x", value[i]);
        }

        vmiPrintf("]\n");
    }
}

//
// Memory callback
//
static VMI_MEM_WATCH_FN(watchCB) {
    report(userData, processor, bytes, value);
}

//
// Constructor
//
VMIOS_CONSTRUCTOR_FN(constructor) {

    // get internal code and data domains
    memDomainP ICDomain = vmirtGetProcessorInternalCodeDomain(processor);
    memDomainP IDDomain = vmirtGetProcessorInternalDataDomain(processor);
```

```
    // add internal callbacks
    vmirtAddFetchCallback(ICDomain, processor, 0, -1, watchCB, "IFETCH");
    vmirtAddWriteCallback(IDDomain, processor, 0, -1, watchCB, "IWRITE");
    vmirtAddReadCallback (IDDomain, processor, 0, -1, watchCB, "IREAD ");
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

8.7 vmirtSetProcessorInternalCodeDomain

Prototype

```
vmirtSetProcessorInternalCodeDomain(
    vmiProcessorP processor,
    memDomainP    domain
);
```

Description

This routine defines the *internal* code domain for a processor. The internal code domain is a model-defined memory domain that records raw physical instruction fetch accesses before such accesses are modified by structures such as tightly-coupled memories, memory-mapped system register blocks and caches that sit between the processor core and the externally-connected instruction bus. The purpose of this function is to allow a processor model to register a domain that can be used by intercept libraries and platform memory watch callbacks to monitor processor activity. See function `vmirtGetProcessorInternalCodeDomain` for an example use of internal domains in an intercept library.

If `vmirtSetProcessorInternalCodeDomain` is not called in the processor virtual memory constructor, then the domain associated with the externally-connected instruction bus is used as the internal domain also.

Example

This example is extracted from the OVP ARM processor model. The model creates memory domains to model TCM and FCSE structures between the externally connected domains and the internal domains.

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

VMI_VMINIT_FN(armVMInit) {

    armP       arm           = (armP)processor;
    memDomainP extCodeDomain = codeDomains[0];
    memDomainP extDataDomain = dataDomains[0];

    . . . lines deleted . . .

    // save physical memDomains on processor structure
    arm->ids.external = extCodeDomain;
    arm->ids.postTCM  = postTCMCodeDomain;
    arm->ids.postFCSE = postFCSECodeDomain;
    arm->dds.external = extDataDomain;
    arm->dds.postTCM  = postTCMDataDomain;
    arm->dds.postFCSE = postFCSEDataDomain;

    // set internal code and data domains
    vmirtSetProcessorInternalCodeDomain(processor, postFCSECodeDomain);
    vmirtSetProcessorInternalDataDomain(processor, postFCSEDataDomain);

    . . . lines deleted . . .
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

8.8 vmirtSetProcessorInternalDataDomain

Prototype

```
vmirtSetProcessorInternalDataDomain(
    vmiProcessorP processor,
    memDomainP    domain
);
```

Description

This routine defines the *internal* data domain for a processor. The internal data domain is a model-defined memory domain that records raw physical load and store accesses before such accesses are modified by structures such as tightly-coupled memories, memory-mapped system register blocks and caches that sit between the processor core and the externally-connected data bus. The purpose of this function is to allow a processor model to register a domain that can be used by intercept libraries and platform memory watch callbacks to monitor processor activity. See function `vmirtGetProcessorInternalDataDomain` for an example use of internal domains in an intercept library.

If `vmirtSetProcessorInternalDataDomain` is not called in the processor virtual memory constructor, then the domain associated with the externally-connected data bus is used as the internal domain also.

Example

This example is extracted from the OVP ARM processor model. The model creates memory domains to model TCM and FCSE structures between the externally connected domains and the internal domains.

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

VMI_VMINIT_FN(armVMInit) {

    armP       arm           = (armP)processor;
    memDomainP extCodeDomain = codeDomains[0];
    memDomainP extDataDomain = dataDomains[0];

    . . . lines deleted . . .

    // save physical memDomains on processor structure
    arm->ids.external = extCodeDomain;
    arm->ids.postTCM  = postTCMCodeDomain;
    arm->ids.postFCSE = postFCSECodeDomain;
    arm->dds.external = extDataDomain;
    arm->dds.postTCM  = postTCMDataDomain;
    arm->dds.postFCSE = postFCSEDataDomain;

    // set internal code and data domains
    vmirtSetProcessorInternalCodeDomain(processor, postFCSECodeDomain);
    vmirtSetProcessorInternalDataDomain(processor, postFCSEDataDomain);

    . . . lines deleted . . .
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

8.9 vmirtSetProcessorCodeDomain

Prototype

```
void vmirtSetProcessorCodeDomain(
    vmiProcessorP processor,
    memDomainP    domain
);
```

Description

This routine allows the current processor code domain to be set to the passed domain. The new code domain will be used for all instruction fetches until `vmirtSetProcessorCodeDomain` is called again with a different domain or `vmirtSetMode` is called with a different mode (in which case the code domain will be set to the one associated with that mode by the virtual memory constructor or by `vmirtSetProcessorCodeDomains`).

Example

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

static void vmic_OverrideCodeDomain(cpuxP cpux, memDomainP override) {
    vmirtSetProcessorCodeDomain((vmiProcessorP)cpux, override);
}
```

Notes and Restrictions

1. If the processor implements virtual memory mappings (created with `vmirtAliasMemoryVM`) then the physical domain associated with old and new domains must be the same.

QuantumLeap Semantics

Non-self-synchronizing

8.10 vmirtSetProcessorDataDomain

Prototype

```
void vmirtSetProcessorDataDomain(
    vmiProcessorP processor,
    memDomainP    domain
);
```

Description

This routine allows the current processor data domain to be set to the passed domain. The new data domain will be used for all loads and stores until `vmirtSetProcessorDataDomain` is called again with a different domain or `vmirtSetMode` is called with a different mode (in which case the data domain will be set to the one associated with that mode by the virtual memory constructor or by `vmirtSetProcessorDataDomains`).

Example

The ARM processor implements some instructions that allow loads and stores to be executed with *user* privileges when the processor is in a *privileged* mode (`LDRT`, `STRT` etc). These are implemented in the reference model by temporarily switching the data domain to the user virtual domain for the duration of the instruction, by the following code in `armMorph.c`:

```
// If this is an LDRT/STRT instruction and we are currently in privileged mode
// with TLB enabled, emit code to switch to the user mode data memDomain
static void emitTranslateOn(armMorphStateP state) {
    if(doTranslate(state)) {
        armEmitArgProcessor(state);
        armEmitCall(state, (vmiCallFn)armVMSetUserPrivilegedModeDataDomain);
    }
}

// If this is after a LDRT/STRT instruction and we are currently in privileged
// mode with TLB enabled, emit code to switch back to the privileged mode data
// memDomain. Note that this code is not executed if the prior access causes an
// exception; in this case, mode is restored in armDataAbort.
static void emitTranslateOff(armMorphStateP state) {
    if(doTranslate(state)) {
        armEmitArgProcessor(state);
        armEmitCall(state, (vmiCallFn)armVMSetNormalPrivilegedModeDataDomain);
    }
}

// Emit code for STR variant
static LOAD_STORE_FN(emitSTR) {

    Uns32  memBits = state->info.sz*8;
    vmiReg rs      = getRSBits(state, state->info.r1, memBits);

    // emit store, possibly with translation (LDRT, STRT)
    emitTranslateOn(state);
    armEmitStoreRRO(state, memBits, offset, base, rs);
    emitTranslateOff(state);
}
```

Functions `armVMSetUserPrivilegedModeDataDomain` and `armVMSetNormalPrivilegedModeDataDomain` are implemented in file `armVM.c` as follows:

```
// Set the current data memDomain
inline static void setVirtualDataDomain(armP arm, memDomainP domain) {
    vmirtSetProcessorDataDomain((vmiProcessorP)arm, domain);
}

 // Set the privileged mode data domain to the user domain (for LDRT, STRT)
void armVMSetUserPrivilegedModeDataDomain(armP arm) {
    setVirtualDataDomain(arm, getMemSubSysD(arm)->vmUser);
}

// Restore the privileged mode data domain to its normal state
void armVMSetNormalPrivilegedModeDataDomain(armP arm) {
    setVirtualDataDomain(arm, getMemSubSysD(arm)->vmPriv);
}
```

The effect of these instructions is to temporarily switch data domain to the user-mode virtual domain for the duration of the instruction and then switch it back to the default privileged mode domain afterwards. The actual JIT-translated code generated is:

1. An embedded call to switch to the user-mode data domain;
2. A simulated read or write action;
3. A second embedded call to switch back to the privileged-mode data domain.

If the simulated read or write causes a simulated exception, then *the second embedded call is not made*. The effect of this could be that the data domain is incorrectly left as the user domain when the instruction completes whereas it ought to be the privileged domain. This case is handled by restoring the privileged domain in the data abort exception function that is called for all such exceptions:

```
// Do data abort exception
void armDataAbort(armP arm, Uns32 faultStatusD, Uns32 faultAddress) {

    // extract fault status and domain from faultStatusD
    armFaultStatus faultStatus = faultStatusD &  ARM_FAULT_STAUS_MASK;
    Uns8           domain      = faultStatusD >> ARM_DOMAIN_SHIFT;

    // update fault status and address registers
    CP15_FIELD(arm, FaultStatus, Status)   = faultStatus;
    CP15_FIELD(arm, FaultAddress, Address) = faultAddress;

    // update domain unless invalid
    if(domain != INVALID_DOMAIN) {
        CP15_FIELD(arm, FaultStatus, Domain) = domain;
    }

    // if the abort is generated in privileged mode when the MMU is enabled,
    // set the data memDomain to the privileged mode memDomain (in case this is
    // a result of LDRT or STRT)
    if(IN_PRIV_TLB_MODE(arm)) {
        armVMSetNormalPrivilegedModeDataDomain(arm);
    }

    // take the exception
    doException(arm, ARM_CPSR_ABORT, ARM_DABT_VECTOR, getPC(arm)+8);
}
```

Notes and Restrictions

1. Be very careful to ensure that domains are set correctly after simulated exceptions: see the example above.
2. If the processor implements virtual memory mappings (created with `vmirtAliasMemoryVM`) then the physical domain associated with old and new domains must be the same.

QuantumLeap Semantics

Non-self-synchronizing

8.11 vmirtSetProcessorCodeDomains

Prototype

```
void vmirtSetProcessorCodeDomains(
    vmiProcessorP processor,
    memDomainPP   domains
);
```

Description

This routine allows the code domains associated with every processor dictionary to be replaced with the passed domains. The `domains` pointer must have one entry for every processor mode. Unlike `vmirtSetProcessorCodeDomain`, the effect of this function is permanent: the new domains will be used with their respective processor modes henceforth, even if the processor switches mode.

Example

This example is taken from the OVP MIPS model. This model supports microthreaded processors that can migrate from one VPE context to another. On each migration, the domains that the TC microthread uses for code and data accesses are modified to reflect the new VPE context.

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

void mipsWriteTCBind(mipsP cxt, mipsP tc, Uns32 newValue) {

    mipsP oldVPE   = VPE_FOR_TC(tc);
    mipsP cpu      = CPU_FOR_VPE(oldVPE);
    Uns32 oldValue = COP0_REG(tc, TCBind);
    mipsP newVPE;

    . . . .

    if(newCurVPE==oldCurVPE) {

        // no action if field CurVPE is unchanged

    } else if(!COP0_FIELD(cpu, MVPControl, VPC)) {

        // TCBind/CurVPE not writable unless MVPControl/VPC is clear
        COP0_FIELD(tc, TCBind, CurVPE) = oldCurVPE;

    } else if((newVPE=findVPE(cpu, newCurVPE))) {

        // reassign to the new parent
        vmirtSetSMPParent((vmiProcessorP)tc, (vmiProcessorP)newVPE);
        tc->context = newVPE;

        . . . .

        // update memory domains if this processor has virtual memory
        if(VIRTUAL_MEMORY_SUPPORT(tc)) {
            vmirtSetProcessorCodeDomains((vmiProcessorP)tc, newVPE->domains);
            vmirtSetProcessorDataDomains((vmiProcessorP)tc, newVPE->domains);
        }
    }
}
```

Notes and Restrictions

1. If the processor implements virtual memory mappings (created with `vmirtAliasMemoryVM`) then the physical domain associated with old and new domains must be the same.

QuantumLeap Semantics

Non-self-synchronizing

8.12 vmirtSetProcessorDataDomains

Prototype

```
void vmirtSetProcessorDataDomains(
    vmiProcessorP processor,
    memDomainPP   domains
);
```

Description

This routine allows the data domains associated with every processor dictionary to be replaced with the passed domains. The `domains` pointer must have one entry for every processor mode. Unlike `vmirtSetProcessorDataDomain`, the effect of this function is permanent: the new domains will be used with their respective processor modes henceforth, even if the processor switches mode.

Example

This example is taken from the OVP MIPS model. This model supports microthreaded processors that can migrate from one VPE context to another. On each migration, the domains that the TC microthread uses for code and data accesses are modified to reflect the new VPE context.

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

void mipsWriteTCBind(mipsP cxt, mipsP tc, Uns32 newValue) {

    mipsP oldVPE   = VPE_FOR_TC(tc);
    mipsP cpu      = CPU_FOR_VPE(oldVPE);
    Uns32 oldValue = COP0_REG(tc, TCBind);
    mipsP newVPE;

    . . . .

    if(newCurVPE==oldCurVPE) {

        // no action if field CurVPE is unchanged

    } else if(!COP0_FIELD(cpu, MVPControl, VPC)) {

        // TCBind/CurVPE not writable unless MVPControl/VPC is clear
        COP0_FIELD(tc, TCBind, CurVPE) = oldCurVPE;

    } else if((newVPE=findVPE(cpu, newCurVPE))) {

        // reassign to the new parent
        vmirtSetSMPParent((vmiProcessorP)tc, (vmiProcessorP)newVPE);
        tc->context = newVPE;

        . . . .

        // update memory domains if this processor has virtual memory
        if(VIRTUAL_MEMORY_SUPPORT(tc)) {
            vmirtSetProcessorCodeDomains((vmiProcessorP)tc, newVPE->domains);
            vmirtSetProcessorDataDomains((vmiProcessorP)tc, newVPE->domains);
        }
    }
}
```

Notes and Restrictions

1. If the processor implements virtual memory mappings (created with `vmirtAliasMemoryVM`) then the physical domain associated with old and new domains must be the same.

QuantumLeap Semantics

Non-self-synchronizing

8.13 vmirtGetProcessorCodeEndian

Prototype

```
memEndian vmirtGetProcessorCodeEndian(vmiProcessorP processor);
```

Description

This routine returns the endianness of memory accesses that the processor makes when *fetching instructions*, either `MEM_ENDIAN_BIG` or `MEM_ENDIAN_LITTLE`. This is typically used in intercept libraries

Notes and Restrictions

1. Note that in some processors, endianness can be switched at run time.

QuantumLeap Semantics

Non-self-synchronizing

8.14 vmirtGetProcessorDataEndian

Prototype

```
memEndian vmirtGetProcessorDataEndian(vmiProcessorP processor);
```

Description

This routine returns the endianness of memory accesses that the processor makes when *reading or writing values*, either `MEM_ENDIAN_BIG` or `MEM_ENDIAN_LITTLE`. This is typically used in intercept libraries.

Example

```
#include “vmi/vmiRt.h”

// Read 4 byte word written by processor at the passed address
Uns32 read4Byte(processorP processor, Addr simAddress) {

    memEndian  endian = vmirtGetProcessorDataEndian(processor);
    memDomainP domain = vmirtGetProcessorDataDomain(processor);

    return vmirtRead4ByteDomain(domain, simAddress, endian, MEM_AA_FALSE);
}
```

Notes and Restrictions

1. Note that in some processors, endianness can be switched at run time.
2. In VMI versions prior to 2.0.17, this function was called `vmirtGetProcessorEndian`.

QuantumLeap Semantics

Non-self-synchronizing

8.15 vmirtNewDomain

Prototype

```
memDomainP vmirtNewDomain(const char *name, Uns8 addressBits);
```

Description

This routine creates a new memory domain object with the passed name. The domain has addresses constrained to `addressBits` wide.

This routine will typically be used during processor virtual memory initialization.

Example

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

typedef enum cpuxModeE {
    CPM_KERNEL = 0,
    CPM_USER   = 1
} cpuxMode;

#define PA_LOW_ADDR  0x0000000000ULL
#define PA_HIGH_ADDR 0x1FFFFFFFFFULL
#define KM_ADDR      0x1000000000000ULL

// Processor VM initialization routine. Arguments:
//     processor:   the current processor
//     codeDomains: array of 2 code domains
//     dataDomains: array of 2 data domains
//
VMI_VMINIT_FN(cpuxVMInit)
{
    cpuxP cpux = (cpuxP)processor;

    memDomainP physicalCodeDomain = codeDomains[0];
    memDomainP physicalDataDomain = dataDomains[0];
    memDomainP kernelDomain;
    memDomainP userDomain;

    ASSERT(
        physicalCodeDomain==physicalDataDomain,
        "expected code & data domains to match"
    );

    // save the physical domain in the processor structure for later use
    cpux->physicalDomain = physicalCodeDomain;

    // define 56-bit virtual address spaces
    kernelDomain = vmirtNewDomain("kernel", 56);
    userDomain   = vmirtNewDomain("user",   56);

    // set up direct mapped image of physical memory range PA_LOW_ADDR:
    // PA_HIGH_ADDR at virtual address KM_ADDR in physical domain
    vmirtAliasMemory(
        physicalCodeDomain,     // physical domain
        kernelDomain,           // virtual domain
        PA_LOW_ADDR,            // low physical address
        PA_HIGH_ADDR,           // high physical address
        KM_ADDR,                // low virtual address
        0                       // no MRU code management
    );

    // specify the domains to use with this processor
    codeDomains[CPM_KERNEL] = kernelDomain;
```

```
    dataDomains[CPM_KERNEL] = kernelDomain;
    codeDomains[CPM_USER]   = userDomain;
    dataDomains[CPM_USER]   = userDomain;
}
```

Notes and Restrictions

1. Memory domains are restricted to a maximum of 64 bits wide.

QuantumLeap Semantics

Synchronizing

8.16 vmirtAliasMemory

Prototype

```
Bool vmirtAliasMemory(
    memDomainP physicalDomain,
    memDomainP virtualDomain,
    Addr       physicalLowAddr,
    Addr       physicalHighAddr,
    Addr       virtualLowAddr,
    memMRUSetP mruSet
);
```

Description

This routine creates an alias of a region in a *physical domain* so that that physical memory is visible within a *virtual domain*. The region to be aliased has the address range `physicalLowAddr:physicalHighAddr` inclusive in the physical domain. Within the virtual domain, the same memory appears starting at address `virtualLowAddr`. The function returns a boolean indicating whether the mapping succeeded.

The same physical memory may be mapped at one or more virtual addresses in one or more virtual domains.

The `mruSet` argument is deprecated in this function and should always be 0.

Example

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

// processor structure definition
typedef struct cpuxS {
    memDomainP physicalDomain;
} cpux, *cpuxP;

static void vmic_MapDataDomain(cpuxP cpux, Uns32 paLow, Uns32 paHigh, Uns32 va) {

    memDomainP domainP = vmirtGetProcessorPhysicalDataDomain((vmiProcessorP)cpux);
    memDomainP domainV = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);

    vmirtAliasMemory(
        domainP,               // physical domain
        domainV,               // virtual domain
        paLow,                 // low physical address
        paHigh,                // high physical address
        va,                    // low virtual address
        0                      // (deprecated – always 0)
    );
}
```

Notes and Restrictions

1. Access permissions in the new virtual memory region are inherited from the physical memory region.
2. An alias specified with `vmirtAliasMemory` will replace any previously-existing alias for the same virtual address range: there is no need to remove a previous mapping (using `vmirtUnaliasMemory`) first.

3. This function should be used only to establish non-paged mappings: use `vmirtAliasMemoryVM` to establish a paged mapping.

QuantumLeap Semantics

Synchronizing

8.17 vmirtUnaliasMemory

Prototype

```
Bool vmirtUnaliasMemory(
    memDomainP virtualDomain,
    Addr       virtualLowAddr,
    Addr       virtualHighAddr
);
```

Description

This routine removes the address range `virtualLowAddr:virtualHighAddr` inclusive from the virtual domain, and returns a boolean indicating whether the unmapping succeeded. The addresses must previously have been mapped by a call to `vmirtAliasMemory` or `vmirtAliasMemoryVM`. Once the address range has been removed from the domain, any attempt to read, write or fetch using addresses in the range will cause the corresponding *memory access exception handler* for the processor model to be called.

Example

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

static void vmic_UnmapDataDomain(cpuxP cpux, Uns32 vaLow, Uns32 vaHigh) {

    memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);

    vmirtUnaliasMemory(
        domain,                 // virtual domain
        vaLow,                  // low virtual address
        vaHigh                  // high virtual address
    );
}
```

Notes and Restrictions

1. There is no requirement for address ranges specified with `vmirtUnaliasMemory` to exactly match ranges previously specified using `vmirtAliasMemory`. For example, it is legal to use `vmirtAliasMemory` to specify a large range and then `vmirtUnaliasMemory` to unmap a "hole" within the large range.
2. If this function is used to remove a mapping created by `vmirtAliasMemoryVM`, the mapping will be removed *irrespective of the ASID which was specified when the mapping was created*. Use function `vmirtUnaliasMemoryVM` to remove a mapping only for a specific ASID.

QuantumLeap Semantics

Synchronizing

8.18 vmirtProtectMemory

Prototype

```
Bool vmirtProtectMemory(
    memDomainP    domain,
    Addr          lowAddr,
    Addr          highAddr,
    memPriv       privilege,
    memPrivAction action
);
```

Description

This routine updates access permissions on the address range `lowAddr:highAddr` inclusive in the memory domain. Privilege values are defined by the type `memPriv` in `vmiTypes.h`:

```
//
// Memory access privilege (bitmask)
//
typedef enum memPrivE {

    // access permission options
    MEM_PRIV_NONE   = 0x00, // no access permitted
    MEM_PRIV_R      = 0x01, // read permitted
    MEM_PRIV_W      = 0x02, // write permitted
    MEM_PRIV_RW     = 0x03, // read & write permitted
    MEM_PRIV_X      = 0x04, // execute permitted
    MEM_PRIV_RX     = 0x05, // read & execute permitted
    MEM_PRIV_WX     = 0x06, // write & execute permitted
    MEM_PRIV_RWX    = 0x07, // read, write & execute permitted

    // other access constraints
    MEM_PRIV_ALIGN  = 0x08, // alignment checks always enabled
    MEM_PRIV_DEVICE = 0x10, // device region with restricted access

} memPriv;
```

Values `0x0-0x7` specify various combinations of read, write and execute privilege. Value `MEM_PRIV_ALIGN` specifies that all accesses to the address range must be *aligned to the access size*; this is required by some processors (e.g. ARM processors with a TLB) in which misaligned accesses are not allowed to regions of memory with certain properties. Value `MEM_PRIV_DEVICE` specifies that this region of memory is a device with restricted access permissions: see the *Imperas Morph Time Function Reference* for more information about how device region access is constrained.

The `action` argument (also from `vmiTypes.h`) defines how the supplied privilege should be used:

```
//
// How privilege mask is to be applied
//
typedef enum memPrivActionE {
    MEM_PRIV_SET,                   // newPriv = privilege
    MEM_PRIV_ADD,                   // newPriv = oldPriv | privilege
    MEM_PRIV_SUB                    // newPriv = oldPriv & ~privilege
} memPrivAction;
```

Any attempt by a simulated program to access memory in a way which is not permitted will cause one of the memory exception handlers in the `vmiAttrs` attribute structure to be called:

```
vmiRdWrSnapFn  rdSnapCB;            // read alignment snap function
vmiRdWrSnapFn  wrSnapCB;            // write alignment snap function
. . .
vmiRdPrivExceptFn  rdPrivExceptCB;  // read privilege exception
vmiWrPrivExceptFn  wrPrivExceptCB;  // write privilege exception
vmiRdAlignExceptFn rdAlignExceptCB; // read alignment exception
vmiWrAlignExceptFn wrAlignExceptCB; // write alignment exception
. . .
vmiIFetchFn        ifetchExceptCB;  // execute privilege exception
```

See the *OVP Processor Modeling Guide* for a detailed description of address snap and exception handlers.

Example

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

static void vmic_AddExecutePermission(cpuxP cpux, Uns32 vaLow, Uns32 vaHigh) {
    memDomainP domain = vmirtGetProcessorCodeDomain((vmiProcessorP)cpux);
    vmirtProtectMemory(domain, vaLow, vaHigh, MEM_PRIV_X, MEM_PRIV_ADD);
}
```

Notes and Restrictions

1. If a no memory at all is defined at a particular physical address range, then any attempt to change access permissions for that physical address range will have no effect – such *void* regions always have no read, write or execute permission.
2. If an address range in a virtual memory domain is unmapped (no physical equivalent has been specified by `vmirtAliasMemory` or `vmirtAliasMemoryVM`) then any attempt to change access permissions for that virtual address range will have no effect – such *void* regions always have no read, write or execute permission.

QuantumLeap Semantics

Synchronizing

8.19 vmirtGetDomainAddressBits

Prototype

```
Uns8 vmirtGetDomainAddressBits(memDomainP domain);
```

Description

This routine returns the width of the domain (the number of significant bits with which that domain can be read or written). The returned value has a maximum value of 64.

This function is typically useful only in intercept library plugins.

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

8.20 vmirtGetDomainPrivileges

Prototype

```
memPriv vmirtGetDomainPrivileges(memDomainP domain, Addr simAddr);
```

Description

Given an address in a memory domain, this function returns the access privileges currently applicable to that address. The returned value is defined by an enum in `vmiTypes.h`:

```
//
// Memory access privilege (bitmask)
//
typedef enum memPrivE {

    // access permission options
    MEM_PRIV_NONE   = 0x00, // no access permitted
    MEM_PRIV_R      = 0x01, // read permitted
    MEM_PRIV_W      = 0x02, // write permitted
    MEM_PRIV_RW     = 0x03, // read & write permitted
    MEM_PRIV_X      = 0x04, // execute permitted
    MEM_PRIV_RX     = 0x05, // read & execute permitted
    MEM_PRIV_WX     = 0x06, // write & execute permitted
    MEM_PRIV_RWX    = 0x07, // read, write & execute permitted

    // other access constraints
    MEM_PRIV_ALIGN  = 0x08, // alignment checks always enabled
    MEM_PRIV_DEVICE = 0x10, // device region with restricted access

} memPriv;
```

Values `0x0-0x7` specify various combinations of read, write and execute privilege. Value `MEM_PRIV_ALIGN` specifies that all accesses to the address range must be *aligned to the access size*; this is required by some processors (e.g. ARM processors with a TLB) in which misaligned accesses are not allowed to regions of memory with certain properties. Value `MEM_PRIV_DEVICE` specifies that this region of memory is a device with restricted access permissions: see the *Imperas Morph Time Function Reference* for more information about how device region access is constrained.

Example

The ARM processor model example uses this function in the virtual memory support module (`armVM.c`) to validate DMA region accessibility:

```
static Bool validateAddressDMA(
    armP        arm,
    memDomainP  virtualDomain,
    armDMAUnitP unit,
    Uns32       address,
    memPriv     priv
) {
    Bool isExternal = unit->isExternal;
    Bool inTCM      = addressInTCM(arm, virtualDomain, unit, address, priv);

    if(!(vmirtGetDomainPrivileges(virtualDomain, address) & priv)) {

        // no access at the required address (access permission failures are
        // higher priority than TCM address failures)
```

```
        return True;

    } else if(inTCM==isExternal) {

        . . .
    }
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

8.21 vmirtIsExecutable

Prototype

```
Bool vmirtIsExecutable(vmiProcessorP processor, Addr simPC);
```

Description

Given an address, this function returns a boolean indicating if that address is executable in the current processor code domain. It is typically used in processor fetch exception handlers.

Example

The OR1K example model uses `vmirtIsExecutable` to determine whether execute privilege exceptions should be taken:

```
VMI_IFETCH_FN(or1kIFetchExceptionCB) {

    or1kP or1k = (or1kP)processor;

    if(or1k->reset) {

        // force a reset
        or1k->reset = False;
        or1kTakeException(or1k, RST_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else if(takeIEE(or1k)) {

        // external interrupt must be taken
        or1kTakeException(or1k, EXI_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else if(takeTEE(or1k)) {

        // tick timer interrupt must be taken
        or1kTakeException(or1k, TTI_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else if(address & 3) {

        // handle misaligned fetch exception
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, BUS_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else if(!vmirtIsExecutable(processor, address)) {

        // handle execute privilege exception
        or1k->EEAR = (Uns32)address;
        or1kTakeException(or1k, IPF_ADDRESS);
        return VMI_FETCH_EXCEPTION_COMPLETE;

    } else {

        // no fetch exception
        return VMI_FETCH_NONE;
    }
}
```

Notes and Restrictions

1. See also `vmirtGetDomainPrivileges` that allows domain-based privilege checks.

QuantumLeap Semantics

Synchronizing

8.22 vmirtGetDomainMapped

Prototype

```
memMapped vmirtGetDomainMapped(
    memDomainP domain,
    Addr       lowAddr,
    Addr       highAddr
);
```

Description

Given an address range `lowAddr:highAddr`, this function returns a member of the enumeration `memMapped` that indicates whether the address range is mapped in the domain:

```
typedef enum memMappedE {
    MEM_MAP_NONE = 0,               // no addresses in the range mapped
    MEM_MAP_PART = 1,               // some addresses in the range mapped
    MEM_MAP_FULL = 2,               // all addresses in the range mapped
} memMapped;
```

If no address in the range is accessible[2], the function returns `MEM_MAP_NONE`; if all addresses in the range are accessible, the function returns `MEM_MAP_FULL`; otherwise, the function returns `MEM_MAP_PART`.

Example

The MIPS example model uses `vmirtGetDomainMapped` to determine whether a memory page is mapped by the TLB in the user address space:

```
// Is the passed address range mapped in the user domain?
static Bool isMappedUser(mipsP mips, Uns32 lowVA, Uns32 highVA) {
    memDomainP userDomain = mips->domains[MIPS_DMODE_USER];
    return vmirtGetDomainMapped(userDomain, lowVA, highVA);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

[2] *Accessible* is defined as having one or more of read, write or execute permissions.

8.23 vmirtMapVAToPA

Prototype

```
Addr vmirtMapVAToPA(
    memDomainP  virtualDomain,
    Addr        virtualAddr,
    memDomaonPP physicalDomain,
    memRegionPP cachedRegion
);
```

Description

Given a virtual domain and a virtual address, this function returns the *physical address* and *physical domain* to which that virtual address corresponds (in other words, the physical address and domain derived from any mapping set up for this virtual address using a previous call to `vmirtAliasMemory` or `vmirtAliasMemoryVM`).

The `cachedRegion` argument, if non-null, should be a pointer to a region cache that can be used by the function to accelerate the lookup process (it saves the result of any previous translation at this location and tries this mapping result first on any subsequent call).

This function is typically used in cache and MMU models.

Example

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

typedef struct cpuxS {
    memRegionP cachedRegion;
} cpux, *cpuxP;

static Uns32 vmic_mapVAToPA(cpuxP cpux, Uns32 va) {
    memDomainP virtD = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);
    memDomainP physD;
    return vmirtMapVAToPA(virtD, va, &physD, &cpux->cachedRegion);
}
```

Notes and Restrictions

1. If there is no valid mapping for the virtual address, then the original address is returned and the `physicalDomain` by-ref argument set to `NULL`.
2. In the case that a virtual mapping has been established to an address that is itself virtually mapped, the address and domain returned will be that of the ultimate, non-virtually-mapped address.

QuantumLeap Semantics

Synchronizing

8.24 vmirtMapMemory

Prototype

```
Bool vmirtMapMemory(
    memDomainP domain,
    Addr       lowAddr,
    Addr       highAddr,
    memType    type
);
```

Description

Function `vmirtMapMemory` specifies that a the address range `lowAddr:highAddr` in the memory domain should be mapped either as ROM (if `type` is `MEM_ROM`) or as RAM (if `type` is `MEM_RAM`). The backing store used for the address range is managed entirely by the simulator.

This function is most often used to populate memory domains created by `vmirtNewDomain` which are intended to be used to model (for example) register banks or local processor memories such as boot ROMs.

Example

```
#include “vmi/vmiRt.h”

#define PROM_SIZE 0x10000
#define PROM_BASE 0x80000000

static void vmic_AllocPROM(cpuxP cpux) {

    // allocate PROM domain
    cpux->promDomain = vmirtNewDomain(“PROM”, 32);

    // specify PROM region in the domain
    vmirtMapMemory(cpux->promDomain, 0, PROM_SIZE-1, MEM_ROM);

    // insert PROM into processor virtual address space
    vmirtAliasMemory(
        cpux->promDomain.
        Cpux->virtualDomain,
        0, PROM_SIZE-1,
        PROM_BASE
    );
}
```

Notes and Restrictions

1. Only memory that is not currently mapped may be mapped using `vmirtMapMemory`.
2. ROM memories are not writable by processors but can be written by `vmirtWriteNByteDomain` etc if the `trueAccess` parameter is `False`. This allows a ROM to be initialized.

QuantumLeap Semantics

Synchronizing

8.25 vmirtMapCallbacks

Prototype

```
Bool vmirtMapCallbacks(
    memDomainP    domain,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemReadFn  readCB,
    vmiMemWriteFn writeCB,
    void         *userData
);
```

Description

Function `vmirtMapCallbacks` specifies that a the address range `lowAddr:highAddr` in the memory domain should be mapped using the passed `readCB` and `writeCB` callback functions.

This function is most often used to populate memory domains created by `vmirtNewDomain` which are intended to be used to model callback-activated devices.

The type `vmiMemReadFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_READ_FN(_NAME)      \
    void _NAME(                     \
        void          *userData,    \
        vmiProcessorP  processor,   \
        Addr           address,     \
        Uns32          bytes,       \
        void          *value        \
        Addr           VA,          \
        Bool           isFetch,     \
        memAccessAttrs attrs        \
    )
typedef VMI_MEM_READ_FN((*vmiMemReadFn));
```

When called, the `readCB` is passed the following arguments:

1. `userData`: model-specific data, supplied as the last argument to `vmirtMapCallbacks`.
2. `processor`: the processor performing the read, or `NULL` if there is no processor context.
3. `address`: the *physical* address of the first byte being read.
4. `bytes`: a count of the bytes being read (typically 1, 2, 4 or 8).
5. `value`: a pointer to a buffer that should be filled by the callback with the result of the read.
6. `VA`: the *virtual* address of the first byte being read.
7. `isFetch`: whether this read is the result of an instruction fetch.
8. `attrs`: the attributes of the read, which is a member of the `memAccessAttrs` enumeration, described below (*new from VMI version 4.2.0*).

The type `vmiMemWriteFn` is also defined in `vmiTypes.h`:

```
#define VMI_MEM_WRITE_FN(_NAME)     \
    void _NAME(                     \
        void          *userData,    \
        vmiProcessorP  processor,   \
        Addr           address,     \
        Uns32          bytes,       \
        const void    *value,       \
        Addr           VA,          \
        memAccessAttrs attrs        \
    )
typedef VMI_MEM_WRITE_FN((*vmiMemWriteFn));
```

When called, the `writeCB` is passed the following arguments:

1. `userData`: model-specific data, supplied as the last argument to `vmirtMapCallbacks`.
2. `processor`: the processor performing the read, or `NULL` if there is no processor context.
3. `address`: the *physical* address of the first byte being written.
4. `bytes`: a count of the bytes being written (typically 1, 2, 4 or 8).
5. `value`: a pointer to a buffer containing the bytes being written.
6. `VA`: the *virtual* address of the first byte being written.
7. `attrs`: the attributes of the read, which is a member of the `memAccessAttrs` enumeration, described below (*new from VMI version 4.2.0*).

The type `vmiMemAccessAttrs` is defined in `vmiTypes.h`:

```
typedef enum memAccessAttrsE {
    MEM_AA_FALSE = 0x0, // this is an artifact access
    MEM_AA_TRUE  = 0x1, // this is a true processor access
    MEM_AA_FETCH = 0x2, // this access is a fetch
} memAccessAttrs;
```

The read and write callbacks can be invoked in one of three state combinations, defined by the processor and attrs arguments as follows:

1. `processor=NULL`, `attrs=MEM_AA_FALSE`: this combination indicates an artifact read/write (e.g. from a debugger) with no processor context.
2. `processor=non-NULL`, `attrs=MEM_AA_FALSE`: this combination indicates an artifact read/write (e.g. from a debugger) with a processor context. This will typically be an access using `icmDebugReadProcessorMemory` or `icmDebugWriteProcessorMemory` in the platform.
3. `processor=non-NULL`, `attrs==MEM_AA_TRUE`: this combination indicates a true processor read/write access.
4. `attrs=MEM_AA_FALSE|MEM_AA_FETCH`: this combination indicates an artifact fetch (usually, by the JIT code generator) with processor context.
5. `attrs=MEM_AA_TRUE|MEM_AA_FETCH`: this combination indicates an true processor fetch.

Example

This example is taken from the OVP MIPS processor model. It creates the memory mapped Global Configuration Registers (GCRs) for a 1004K processor.

```
static mipsGCRGlobalP allocGCRGlobals(mipsP tc, memDomainP physicalDomain) {
```

```
    mipsP          vpe        = VPE_FOR_TC(tc);
    mipsGCRGlobalP gcrGlobals = vpe->gcrGlobals;

    if(!gcrGlobals) {

        mipsP cpu = CPU_FOR_VPE(vpe);

        gcrGlobals = cpu->gcrGlobals;

        if(!gcrGlobals) {

            mipsP cmp = CMP_FOR_CPU(cpu);

            gcrGlobals = cmp->gcrGlobals;

            if(!gcrGlobals) {

                gcrGlobals = TYPE_CALLOC(mipsGCRGlobal);

                // link gcrGlobals to CMP
                . . . code omitted for clarity . . .

                // create GCR and CMP domains
                memDomainP gcrDomain = vmirtNewDomain("GCR", 32);
                memDomainP cmpDomain = vmirtNewDomain("CMP", VM_BITS);

                // implement the GCR region using callbacks
                vmirtMapCallbacks(
                    gcrDomain, 0, GCB_SIZE-1, readGCR, writeGCR, gcrGlobals
                );

                . . . code omitted for clarity . . .
}
```

Notes and Restrictions

1. Only memory that is not currently mapped may be mapped natively using `vmirtMapCallbacks`.

QuantumLeap Semantics

Synchronizing

8.26 vmirtMapNativeMemory

Prototype

```
Bool vmirtMapNativeMemory(
    memDomainP domain,
    Addr       lowAddr,
    Addr       highAddr,
    void      *memory
);
```

Description

The simulator usually maintains the backing store used for simulated memory contents itself – memory pages are allocated on demand when required by loads and stores from previously-unmapped regions, and might be relocated if merging of adjacent memory regions occurs.

Sometimes, it may be required that specific address ranges are not managed by the simulator in this way but should instead be mapped to specific native host memory buffers. As an example, it might be that it is more convenient to maintain the contents of a frame buffer at a fixed native address so that it can easily be processed by a renderer implemented using a PSE[3] or by C code in an ICM platform.

Function `vmirtMapNativeMemory` specifies that a native memory buffer specified as the `memory` argument should be used to hold the contents of the address range `lowAddr:highAddr` in the memory domain.

Example

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

#define FRAME_BUF_SIZE 0x100000
static Uns8 frameBuffer[FRAME_BUF_SIZE];

static void vmic_MapFrameBuffer(cpuxP cpux, Uns32 fbLow) {

    memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);
    Uns32      fbHigh = fbLow+FRAME_BUF_SIZE-1;

    // use native buffer 'frameBuffer' as backing store for the read buffer
    // of the passed processor
    vmirtMapNativeMemory(domain, fbLow, fbHigh, frameBuffer);
}
```

Notes and Restrictions

1. Be careful to ensure the native memory is large enough to cover the required address range – the simulator does not check this.
2. Only memory that is not currently mapped may be mapped natively using `vmirtMapNativeMemory`.

[3] A PSE is a *peripheral simulation engine*, used to implement peripheral models in the Imperas environment.

QuantumLeap Semantics
Synchronizing

8.27 vmirtAddReadCallback

Prototype

```
void vmirtAddReadCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine installs a model-specific *read watch point function*, `watchCB`, on the address range `lowAddr:highAddr` in `domain`. The callback will be invoked whenever any address in the range is read during simulation.

If `scope` is `NULL`, the callback will be activated on any access to the memory range by *any processor* using any address alias. If scope is non-`NULL` and not the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the *given processor* accesses the memory range using any address alias. If `scope` is the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the domain is accessed by a *master processor* of the domain. A processor is a master processor of a domain if it directly accesses the domain, or accesses a domain that has been aliased to the domain using functions `vmirtAliasMemory` or `vmirtAliasMemoryVM` at any level of indirection.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME)     \
    void _NAME(                     \
        void         *userData,     \
        vmiProcessorP processor,    \
        Addr          address,      \
        Uns32         bytes,        \
        const void   *value,        \
        Addr          VA            \
    )
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

When called, the `watchCB` is passed the following arguments:

1. `userData`: model-specific data, supplied as the first argument to `vmirtAddReadCallback`.
2. `processor`: the processor performing the read, or `NULL` if this is an artifact read (for example, resulting from a call to `vmirtReadNByteDomain` with the `attrs==MEM_AA_FALSE`, or an access by the simulator core).
3. `address`: the *physical* address of the first byte being read or written.
4. `bytes`: a count of the bytes being read (typically 1, 2, 4 or 8).
5. `value`: a pointer to a buffer containing the bytes being read.

6. VA: the *virtual* address of the first byte being read[4].

Example

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

static VMI_MEM_WATCH_FN(trapRead) {
    vmiPrintf(
        "%s: read %u bytes at address 0xllu\n",
        userData, bytes, address
    );
}

static void vmic_InstallTrapRead(cpuxP cpux, Addr low, Addr high) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback(domain, processor, low, high, trapRead, "TRAP READ");
}
```

Notes and Restrictions

1. It is legal to install more than one callback on the same address range using this function. In this case, callbacks will be called in the order that they were installed.
2. If a callback is installed on a virtual memory domain, the callback will be invoked whenever the *physical memory* to which that address range is mapped is read, irrespective of the source of the read.
3. Callbacks may be installed on virtual memory ranges that currently have no mapping. If those ranges are subsequently mapped, the callbacks will be invoked whenever the physical memory to which that address range is mapped is read, irrespective of the source of the read, as described above.
4. Callbacks remain active on a domain through all mapping and unmapping operations that may follow, until removed by `vmirtRemoveReadCallback`.
5. Argument scope is present only for VMI version 5.8.0 onwards.

QuantumLeap Semantics

Synchronizing

[4] To be precise, this is the *address with which the read was issued*, which may or may not be a valid virtual address in the domain in which the callback is installed.

8.28 vmirtAddWriteCallback

Prototype

```
void vmirtAddWriteCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine installs a model-specific *write watch point function*, `watchCB`, on the address range `lowAddr:highAddr` in `domain`. The callback will be invoked whenever any address in the range is written during simulation.

If `scope` is `NULL`, the callback will be activated on any access to the memory range by *any processor* using any address alias. If scope is non-`NULL` and not the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the *given processor* accesses the memory range using any address alias. If `scope` is the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the domain is accessed by a *master processor* of the domain. A processor is a master processor of a domain if it directly accesses the domain, or accesses a domain that has been aliased to the domain using functions `vmirtAliasMemory` or `vmirtAliasMemoryVM` at any level of indirection.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME)      \
    void _NAME(                      \
        void         *userData,      \
        vmiProcessorP processor,     \
        Addr          address,       \
        Uns32         bytes,         \
        const void   *value,         \
        Addr          VA             \
    )
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

When called, the `watchCB` is passed the following arguments:

1. `userData`: model-specific data, supplied as the first argument to `vmirtAddWriteCallback`.
2. `processor`: the processor performing the write, or `NULL` if this is an artifact write (for example, resulting from a call to `vmirtWriteNByteDomain` with `attrs==MM_AA_FALSE`).
3. `address`: the address of the first byte being written.
4. `bytes`: a count of the bytes being written (typically 1, 2, 4 or 8).
5. `value`: a pointer to a buffer containing the bytes being written.

6. VA: the *virtual* address of the first byte being written[5].

Example

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

static VMI_MEM_WATCH_FN(trapWrite) {
    vmiPrintf(
        "%s: wrote %u bytes at address 0xllu\n",
        userData, bytes, address
    );
}

static void vmic_InstallTrapWrite(cpuxP cpux, Addr low, Addr high) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddWriteCallback(domain, processor, low, high, trapRead, "TRAP WRITE");
}
```

Notes and Restrictions

1. It is legal to install more than one callback on the same address range using this function. In this case, callbacks will be called in the order that they were installed.
2. If a callback is installed on a virtual memory domain, the callback will be invoked whenever the *physical memory* to which that address range is mapped is written, irrespective of the source of the write.
3. Callbacks may be installed on virtual memory ranges that currently have no mapping. If those ranges are subsequently mapped, the callbacks will be invoked whenever the physical memory to which that address range is mapped is written, irrespective of the source of the write, as described above.
4. Callbacks remain active on a domain through all mapping and unmapping operations that may follow, until removed by `vmirtRemoveWriteCallback`.
5. Argument scope is present only for VMI version 5.8.0 onwards.

QuantumLeap Semantics

Synchronizing

[5] To be precise, this is the *address with which the write was issued*, which may or may not be a valid virtual address in the domain in which the callback is installed.

8.29 vmirtAddFetchCallback

Prototype

```
void vmirtAddFetchCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine installs a model-specific *fetch watch point function*, `watchCB`, on the address range `lowAddr:highAddr` in `domain`. The callback will be invoked whenever any instruction in the address in the range is fetched during simulation.

If `scope` is `NULL`, the callback will be activated on any access to the memory range by *any processor* using any address alias. If scope is non-`NULL` and not the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the *given processor* accesses the memory range using any address alias. If `scope` is the special value `VMI_MASTER_SCOPE`, the callback will be activated only when the domain is accessed by a *master processor* of the domain. A processor is a master processor of a domain if it directly accesses the domain, or accesses a domain that has been aliased to the domain using functions `vmirtAliasMemory` or `vmirtAliasMemoryVM` at any level of indirection.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME)      \
    void _NAME(                      \
        void         *userData,      \
        vmiProcessorP processor,     \
        Addr          address,       \
        Uns32         bytes,         \
        const void   *value          \
        Addr          VA             \
    )
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

When called, the `watchCB` is passed the following arguments:

1. `userData`: model-specific data, supplied as the first argument to `vmirtAddFetchCallback`.
2. `processor`: the processor performing the fetch, or `NULL` if this is an artifact fetch.
3. `address`: the address of the first byte being fetched.
4. `bytes`: a count of the bytes being fetched (typically 1, 2, 4 or 8).
5. `value`: a pointer to a buffer containing the bytes being fetched.
6. `VA`: the *virtual* address of the first byte being fetched.
7. Argument scope is present only for VMI version 5.8.0 onwards.

Example

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

static VMI_MEM_WATCH_FN(trapFetch) {
    vmiPrintf(
        "%s: fetched %u bytes at address 0xllu\n",
        userData, bytes, address
    );
}

static void vmic_InstallWatchFetch(cpuxP cpux, Addr low, Addr high) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddFetchCallback(domain, processor, low, high, trapRead, "TRAP FETCH");
}
```

Notes and Restrictions

1. It is legal to install more than one callback on the same address range using this function. In this case, callbacks will be called in the order that they were installed.
2. Callbacks may be installed on virtual memory ranges that currently have no mapping. If those ranges are subsequently mapped, the callbacks will be invoked whenever a fetch is made to an address in that range.
3. Callbacks remain active on a domain through all mapping and unmapping operations that may follow, until removed by `vmirtRemoveFetchCallback`.

QuantumLeap Semantics

Synchronizing

8.30 vmirtRemoveReadCallback

Prototype

```
void vmirtRemoveReadCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine uninstalls any previously-installed read watch point function `watchCB` on the address range `lowAddr:highAddr`. Only calls that were installed with exactly matching `userData` and `scope` will be uninstalled.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME)      \
    void _NAME(                      \
        void         *userData,      \
        vmiProcessorP processor,     \
        Addr          address,       \
        Uns32         bytes,         \
        const void   *value          \
    )
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

Example

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

static VMI_MEM_WATCH_FN(trapRead) {
    vmiPrintf(
        “%s: read %u bytes at address 0xllu\n”,
        userData, bytes, address
    );
}

static void vmic_InstallTrapRead(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddReadCallback(domain, processor, low, high, trapRead, id);
}
static void vmic_RemoveTrapRead(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtRemoveReadCallback(domain, processor, low, high, trapRead, id);
}

static char *id1 = “id1”;
static char *id2 = “id2”;

static void testReadTraps(cpuxP cpux, Addr low, Addr high) {
    // install two read watch point callbacks with userData “id1” and “id2”
    vmic_InstallTrapRead(cpux, low, high, id1);
    vmic_InstallTrapRead(cpux, low, high, id2);
    // remove read watch point callback with userData “id1”
    vmic_RemoveTrapRead(cpux, low, high, id1);
}
```

Notes and Restrictions

1. Address ranges used when installing and uninstalling callbacks do not need to match. For example, it is legal to install a callback on a wide range and uninstall it only from a narrower range. In this case, the callback will remain active on the part or parts of the install range that are excluded from the uninstall range.
2. Argument scope is present only for VMI version 5.8.0 onwards.

QuantumLeap Semantics

Synchronizing

8.31 vmirtRemoveWriteCallback

Prototype

```
void vmirtRemoveWriteCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine uninstalls any previously-installed write watch point function `watchCB` on the address range `lowAddr:highAddr`. Only calls that were installed with exactly matching `userData` and `scope` will be uninstalled.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME)      \
    void _NAME(                      \
        void         *userData,      \
        vmiProcessorP processor,     \
        Addr          address,       \
        Uns32         bytes,         \
        const void   *value          \
    )
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

Example

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

static VMI_MEM_WATCH_FN(trapWrite) {
    vmiPrintf(
        “%s: write %u bytes at address 0xllu\n”,
        userData, bytes, address
    );
}

static void vmic_InstallTrapWrite(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddWriteCallback(domain, processor, low, high, trapRead, id);
}
static void vmic_RemoveTrapWrite(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtRemoveWriteCallback(domain, Processor, low, high, trapWrite, id);
}

static char *id1 = “id1”;
static char *id2 = “id2”;

static void testWriteTraps(cpuxP cpux, Addr low, Addr high) {
    // install two write watch point callbacks with userData “id1” and “id2”
    vmic_InstallTrapWrite(cpux, low, high, id1);
    vmic_InstallTrapWrite(cpux, low, high, id2);
    // remove write watch point callback with userData “id1”
    vmic_RemoveTrapWrite(cpux, low, high, id1);
}
```

Notes and Restrictions

1. Address ranges used when installing and uninstalling callbacks do not need to match. For example, it is legal to install a callback on a wide range and uninstall it only from a narrower range. In this case, the callback will remain active on the part or parts of the install range that are excluded from the uninstall range.
2. Argument scope is present only for VMI version 5.8.0 onwards.

QuantumLeap Semantics

Synchronizing

8.32 vmirtRemoveFetchCallback

Prototype

```
void vmirtRemoveFetchCallback(
    memDomainP    domain,
    vmiProcessorP scope,
    Addr          lowAddr,
    Addr          highAddr,
    vmiMemWatchFn watchCB,
    void         *userData
);
```

Description

This routine uninstalls any previously-installed fetch watch point function `watchCB` on the address range `lowAddr:highAddr`. Only calls that were installed with exactly matching `userData` and `scope` will be uninstalled.

The type `vmiMemWatchFn` is defined in `vmiTypes.h`:

```
#define VMI_MEM_WATCH_FN(_NAME)      \
    void _NAME(                      \
        void         *userData,      \
        vmiProcessorP processor,     \
        Addr          address,       \
        Uns32         bytes,         \
        const void   *value          \
    )
typedef VMI_MEM_WATCH_FN((*vmiMemWatchFn));
```

Example

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

static VMI_MEM_WATCH_FN(trapFetch) {
    vmiPrintf(
        “%s: fetch %u bytes at address 0xllu\n”,
        userData, bytes, address
    );
}

static void vmic_InstallTrapFetch(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtAddFetchCallback(domain, processor, low, high, trapFetch, id);
}
static void vmic_RemoveTrapFetch(cpuxP cpux, Addr low, Addr high, char *id) {
    vmiProcessorP processor = (vmiProcessorP)cpux;
    memDomainP    domain    = vmirtGetProcessorDataDomain(processor);
    vmirtRemoveFetchCallback(domain, processor, low, high, trapFetch, id);
}

static char *id1 = “id1”;
static char *id2 = “id2”;

static void testFetchTraps(cpuxP cpux, Addr low, Addr high) {
    // install two fetch watch point callbacks with userData “id1” and “id2”
    vmic_InstallTrapFetch(cpux, low, high, id1);
    vmic_InstallTrapFetch(cpux, low, high, id2);
    // remove fetch watch point callback with userData “id1”
    vmic_RemoveTrapFetch(cpux, low, high, id1);
}
```

Notes and Restrictions

1. Address ranges used when installing and uninstalling callbacks do not need to match. For example, it is legal to install a callback on a wide range and uninstall it only from a narrower range. In this case, the callback will remain active on the part or parts of the install range that are excluded from the uninstall range.
2. Argument scope is present only for VMI version 5.8.0 onwards.

QuantumLeap Semantics

Synchronizing

8.33 vmirtAliasMemoryVM

Prototype

```
Bool vmirtAliasMemoryVM(
    memDomainP physicalDomain,
    memDomainP virtualDomain,
    Addr       physicalLowAddr,
    Addr       physicalHighAddr,
    Addr       virtualLowAddr,
    memMRUSetP mruSet,
    memPriv    privilege,
    Uns32      ASIDMaskOrG,
    Uns32      ASID
);
```

Description

This routine creates a virtual memory page alias of a region in a *physical domain* so that that physical memory is visible within a *virtual domain*. The region to be aliased has the address range `physicalLowAddr:physicalHighAddr` inclusive in the physical domain. Within the virtual domain, the same memory appears starting at address `virtualLowAddr`. The new mapping is created with access privileges specified by `privilege`:

```
//
// Memory access privilege (bitmask)
//
typedef enum memPrivE {

    // access permission options
    MEM_PRIV_NONE   = 0x00, // no access permitted
    MEM_PRIV_R      = 0x01, // read permitted
    MEM_PRIV_W      = 0x02, // write permitted
    MEM_PRIV_RW     = 0x03, // read & write permitted
    MEM_PRIV_X      = 0x04, // execute permitted
    MEM_PRIV_RX     = 0x05, // read & execute permitted
    MEM_PRIV_WX     = 0x06, // write & execute permitted
    MEM_PRIV_RWX    = 0x07, // read, write & execute permitted

    // other access constraints
    MEM_PRIV_ALIGN  = 0x08, // alignment checks always enabled
    MEM_PRIV_DEVICE = 0x10, // device region with restricted access

} memPriv;
```

Values `0x0-0x7` specify various combinations of read, write and execute privilege. Value `MEM_PRIV_ALIGN` specifies that all accesses to the address range must be *aligned to the access size*; this is required by some processors (e.g. ARM processors with a TLB) in which misaligned accesses are not allowed to regions of memory with certain properties. Value `MEM_PRIV_DEVICE` specifies that this region of memory is a device with restricted access permissions: see the *Imperas Morph Time Function Reference* for more information about how device region access is constrained.

The type of mapping to create is specified using the `ASIDMaskOrG` argument. Prior to VMI version 4.5.0, this argument was a simple boolean, `isGlobal`. A value of `True` indicated that the mapping was a *global* mapping (always valid); a value of `False`

indicated that the mapping was only valid when the processor ASID matched the passed `ASID` (see function `vmirtSetProcessorASID`). From VMI version 4.5.0, the parameter has been changed to an `Uns32`, `ASIDMaskOrG`. This is interpreted as follows:

1. **`ASIDMaskOrG=1 (True)`**: A value of 1 indicates that the mapping is global and the `ASID` is ignored, as previously.
2. **`ASIDMaskOrG=0 (False)`**: A value of 0 indicates that the mapping is ASID-mapped and that the processor ASID must exactly match the passed `ASID` for the mapping to be valid, as previously.
3. **`ASIDMaskOrG=<another value>`**: If any other value is specified for the `ASIDMaskOrG` argument, then this value is used as a *mask* to apply to the processor ASID to determine if the region matches. Specifically, the region is deemed to match only if:
   `(REGION_ASID&ASIDMaskOrG)==(PROCESSOR_ASID&ASIDMaskOrG)`

This modified mapping behavior simplifies implementation of processor models that support *hardware hypervisors*. Such a processor usually allows memory regions to be created in different classes:

1. Global regions;
2. ASID-mapped regions;
2 Regions which must match a *virtual machine ID* (VMID);
3 Regions that must match a VMID and an ASID.

Typically, such processors also have two levels of TLB in some operating modes: a *guest* TLB that maps addresses to intermediate physical addresses (IPAs) and a *second stage* TLB that maps the IPA to a true physical address. Using an ASID mask enables both TLB stages to be implemented as a single `vmirtAliasMemoryVM` call with the ASID mask used to select the parts of a simulated ASID that are relevant for this region. The simulated ASID passed to `vmirtAliasMemoryVM` will usually contain:

1. The region ASID as a bitfield;
2. The processor VMID as a bitfield;
3. Further bits indicated the enabled status of the guest and second stage TLBs.

The same physical memory may be mapped at one or more virtual addresses in one or more virtual domains.

The `mruSet` argument is used by the simulator's native MRU/LRU management, typically to implement a TLB LRU replacement policy – See the *Imperas Processor Modeling Guide* for more information and examples.

The function returns a boolean indicating whether the mapping succeeded.

Example

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

// processor structure definition
typedef struct cpuxS {
    memDomainP physicalDomain;
    Uns32      ASID;
```

```
} cpux, *cpuxP;

static void vmic_MapDataDomain(
    cpuxP cpux,
    Uns32 paLow,
    Uns32 paHigh,
    Uns32 va,
    Bool  isGlobal
) {
    memDomainP domainP = vmirtGetProcessorPhysicalDataDomain((vmiProcessorP)cpux);
    memDomainP domainV = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);

    vmirtAliasMemoryVM(
        domainP,                // physical domain
        domainV,                // virtual domain
        paLow,                  // low physical address
        paHigh,                 // high physical address
        va,                     // low virtual address
        0,                      // not MRU-managed
        MEM_PRIV_RW,            // access permissions (read and write)
        isGlobal,               // is the VM page ASID-managed?
        cpux->ASID              // ASID to use (if not global)
    );
}
```

Notes and Restrictions

1. Access permissions in the new virtual memory region are those of the physical memory region, restricted further by the passed `privilege`. If the passed `privilege` specifies an access permission that is not present in the physical region, that access permission is not granted.
2. Paged alias regions must have a size that is an exact power of two and must be aligned to a boundary that is a multiple of the same power of two.
3. Paged alias regions are limited to a maximum size of 4Gb (2^32 bytes).
4. `vmirtAliasMemoryVM` may not be used to create an alias to a physical domain that itself has paged aliases.
5. If a virtual domain contains any paged alias regions, then all such regions must refer to the same physical domain (it is illegal to create paged aliases to two or more physical domains from one virtual domain).

QuantumLeap Semantics

Synchronizing

8.34 vmirtUnaliasMemoryVM

Prototype

```
Bool vmirtUnaliasMemoryVM(
    memDomainP virtualDomain,
    Addr       virtualLowAddr,
    Addr       virtualHighAddr,
    Uns32      ASIDMaskOrG,
    Uns32      ASID
);
```

Description

This routine removes the page-mapped address range `virtualLowAddr:virtualHighAddr` inclusive from the virtual domain, and returns a boolean indicating whether the unmapping succeeded. For the mapping to be removed, the addresses must previously have been mapped by a call to `vmirtAliasMemoryVM`, using the same values for `ASIDMaskOrG` and `ASID`. Once the address range has been removed from the domain, any attempt to read, write or fetch using addresses in the range will cause the corresponding *memory access exception handler* for the processor model to be called. When implementing an MMU model, these exception handlers will typically remap memory using `vmirtAliasMemoryVM` or perform other actions required to model the MMU.

Refer to the documentation of `vmirtAliasMemoryVM` for a detailed description of the `ASID` and `ASIDMaskOrG` arguments.

Example

```
#include “vmi/vmiRt.h”

// processor structure definition
typedef struct cpuxS {
    memDomainP physicalDomain;
    Uns32      ASID;
} cpux, *cpuxP;

static void vmic_UnmapDataDomain(
    cpuxP cpux,
    Uns32 vaLow,
    Uns32 vaHigh,
    Bool  isGlobal,
    Uns32 ASID
) {
    memDomainP domain = vmirtGetProcessorDataDomain((vmiProcessorP)cpux);

    vmirtUnaliasMemory(
        domain,                 // virtual domain
        vaLow,                  // low virtual address
        vaHigh                  // high virtual address
        isGlobal,               // removing a global entry?
        ASID                    // ASID (if not a global entry)
    );
}
```

Notes and Restrictions

1. Address ranges specified with `vmirtUnaliasMemoryVM` must exactly match the ranges previously specified using `vmirtAliasMemoryVM` when the mapping was established.
2. Using `vmirtUnaliasMemoryVM` to remove an alias where none was established has no effect.
3. Using `vmirtUnaliasMemoryVM` to remove an alias which was established with different `isGlobal` or `ASID` values has no effect. Use `vmirtUnaliasMemory` to remove an alias irrespective of `isGlobal` and `ASID`.

QuantumLeap Semantics

Synchronizing

8.35 vmirtGetDomainMappedASID

Prototype

```
memMappedASID vmirtGetDomainMappedASID(
    memDomainP domain,
    Addr       lowAddr,
    Addr       highAddr,
    Uns32      ASID
);
```

Description

This routine returns an enumeration value indicating whether any address in the range `lowAddr:highAddr` is ASID-mapped in the `domain` with the passed `ASID`. Type `memMappedASID` is defined in `vmiTypes.h` as follows:

```
typedef enum memMappedASIDE {
    ASID_MAP_NONE,                  // address not ASID-mapped
    ASID_MAP_INACTIVE,              // address ASID-mapped but not active
    ASID_MAP_ACTIVE                 // address ASID-mapped and active
} memMappedASID;
```

A return value of `ASID_MAP_NONE` means that no address in the range is mapped with that ASID. A return value of `ASID_MAP_INACTIVE` means that there is some address in that range mapped using the ASID, but the mapped memory has not been referenced since the processor last switched to the ASID (in other words, the memory has not been recently addressed and is stale). A return value of `ASID_MAP_ACTIVE` means that there is some address in that range mapped using the ASID and that the referenced memory has been referenced since the processor last switched to the ASID (in other words, the memory is not stale).

This function is typically used for simulation efficiency reasons to determine whether or not a TLB mapping is valid, or should be preserved.

Example

This example is from the MIPS model:

```
static memMappedASID getPTEntryStatus(
    mipsP     tc,
    tlbEntryP entry,
    mipsMode  mode,
    Uns32     index
) {
    Uns32         mask   = MODE_MASK(mode);
    tlbPhysP      pt     = &entry->pt[index];
    memDomainP    domain = tc->domains[mode];
    Uns32         lowVA  = getPTLowVA(entry, index);
    Uns32         highVA = lowVA + entry->simPageSize - 1;
    Uns32         ASID   = entry->ASID;
    memMappedASID status = vmirtGetDomainMappedASID(domain, lowVA, highVA, ASID);

    // if this entry was originally mapped with this ASID but has since been
    // remapped with another ASID, clear the simulator mapped flag (faster
    // processing next time)
    if(status==ASID_MAP_NONE) {
        pt->simMapped &= ~mask;
```

```
    }

    return status;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

8.36 vmirtSetProcessorASID

Prototype

```
void vmirtSetProcessorASID(vmiProcessorP processor, Uns32 ASID);
```

Description

This routine sets the processor ASID (*address space ID*) to the passed value. If a non-global page-mapped alias is created with `vmirtAliasMemoryVM`, then that mapping will be valid only if the ASID matches the current processor ASID.

Example

This example is from the MIPS model:

```
void mipsSetASID(mipsP tc, Uns8 newASID) {

    Uns8 oldASID = getASID(tc);

    COP0_FIELD(tc, EntryHi, ASID) = newASID;

    if(!tc->tlb) {
        // no action unless TLB active
    } else if(oldASID==newASID) {
       // no action unless ASID changed
    } else {

        // remove non-user mappings for all hidden TLB entries
        setPrivilegesASIDChangeNonUser(tc, newASID);

        // update the core model ASID
        vmirtSetProcessorASID((vmiProcessorP)tc, newASID);
    }
}
```

Notes and Restrictions

1. The initial processor ASID is 0.

QuantumLeap Semantics

Non-self-synchronizing

8.37 vmirtGetProcessorASID

Prototype

```
Uns32 vmirtGetProcessorASID(vmiProcessorP processor);
```

Description

This routine returns the processor ASID (*address space ID*) previously set by `vmirtSetProcessorASID`.

Notes and Restrictions

1. The initial processor ASID is 0.

QuantumLeap Semantics

Non-self-synchronizing

8.38 vmirtGetMRUStateTable

Prototype

```
const Uns32 *vmirtGetMRUStateTable(Uns32 numEntries, Uns32 entryIndex);
```

Description

This function is used when modeling constructs such as *translation lookaside buffers* (TLBs) with *least-recently-used* (LRU) replacement policies. It returns a transition table to be used for entry `entryIndex` of a LRU-managed set with `numEntries` entries.

It is possible to associate a memory region mapping with a transition table / state variable pair using the `mruSet` argument of `vmirtAliasMemory`. When this association has been made, the simulator will automatically maintain the state variable so that the least-recently-used region of a set of regions can be determined using `vmirtGetNthStateIndex`.

Example

See the *Imperas Processor Modeling Guide* for detailed examples using this function.

Notes and Restrictions

1. Currently, a value of `numEntries` greater than `8` is not supported.
2. `entryIndex` must lie in the range `0..numEntries-1`.

QuantumLeap Semantics

Synchronizing

8.39 vmirtGetNthStateIndex

Prototype

```
Uns8 vmirtGetNthStateIndex(Uns32 numEntries, Uns32 state, Uns32 position);
```

Description

This function is used when modeling constructs such as *translation lookaside buffers* (TLBs) with *least-recently-used* (LRU) replacement policies. Given a state in a table with `numEntries` previously allocated by `vmirtGetMRUStateTable`, it returns the entry index with the passed position implied by `state`. The most-recently-used entry has position `0`; the least-recently-used entry has position `numEntries-1`.

It is possible to associate a memory region mapping with a transition table / state variable pair using the `mruSet` argument of `vmirtAliasMemory`. When this association has been made, the simulator will automatically maintain the state variable so that the least-recently-used region of a set of regions can be determined using this function.

Example

See the *Imperas Processor Modeling Guide* for detailed examples using this function.

Notes and Restrictions

1. `position` must lie in the range `0..numEntries-1`.
2. This function returns a correct state index only for state transition tables allocated with `vmirtGetMRUStateTable`.

QuantumLeap Semantics

Synchronizing

8.40 vmirtAddPCCallback

Prototype

```
Bool vmirtAddPCCallback(
    vmiProcessorP processor,
    Addr          simPC,
    vmiPCWatchFn  watchCB,
    void         *userData
);
```

Description

This routine registers a callback function `watchCB` that is called whenever the processor executes as instruction at virtual address `simPC`. Type `vmiPCWatchFn` is defined in `vmiTypes.h` as follows:

```
#define VMI_PC_WATCH_FN(_NAME) void _NAME( \
    void         *userData,  \
    vmiProcessorP processor, \
    Addr          thisPC     \
)
typedef VMI_PC_WATCH_FN((*vmiPCWatchFn));
```

The function return address indicates whether the callback was successfully added. The callback can be removed using `vmirtRemovePCCallback`.

This function is most commonly used in combination with the file/line query API to implement coverage tools: see the example.

Example

This example shows the core of a *line coverage tool*, implemented as a binary intercept library.

```
static VMI_PC_WATCH_FN(incrementCount) {
    coverLineP line = (coverLineP)userData;
    line->count++;
}

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiFileLineCP fl;
    coverLineP    line = &object->lines[0];

    // iterate over the file/line pairs recorded
    // in the DWARF information for the processor executable
    // and register a callback when the indicated address is executed
    for(
        fl=vmirtNextFLByAddr(processor, 0);
        fl!=NULL && line<=&object->lines[MAX_LINES];
        fl=vmirtNextFLByAddr(processor, fl), line++, object->numLines++
    ) {

        Addr address = vmirtGetFLAddr(fl);

        // Set the file/line pointer for this entry
        line->fl = fl;

        // Add callback on the address for this file/line
        // with the appropriate coverLineP to increment
```

```
        vmirtAddPCCallback(processor, address, incrementCount, line);
    }
}
```

Notes and Restrictions

1. This function is available only with the Imperas Professional Tools.
2. It is possible to add many callbacks with the same `processor` and `simPC` arguments. Each will be called in turn when the processor executes at the given simulated address.

QuantumLeap Semantics

Synchronizing

8.41 vmirtRemovePCCallback

Prototype

```
Bool vmirtRemovePCCallback(
    vmiProcessorP processor,
    Addr          simPC,
    vmiPCWatchFn  watchCB,
    void         *userData
);
```

Description

This routine uninstalls a callback function `watchCB` that was previously installed using the passed `processor`, `simPC`, `watchCB` and `userData` arguments. Type `vmiPCWatchFn` is defined in `vmiTypes.h` as follows:

```
#define VMI_PC_WATCH_FN(_NAME) void _NAME( \
    void         *userData,  \
    vmiProcessorP processor, \
    Addr          thisPC     \
)
typedef VMI_PC_WATCH_FN((*vmiPCWatchFn));
```

The function return address indicates whether the callback was successfully removed: al value of `False` indicates that no callback was previously installed with matching arguments.

This function is most commonly used to implement tools that require simulated PC tracking to change dynamically (for example, a tool that performs function call/return tracing).

Example

This example shows part of a function call/return tracing tool, implemented as a binary intercept library. A callback is installed on every function return address; when called, this function takes appropriate action to track the return and then uninstalls itself.

```
static VMI_PC_WATCH_FN(exitFunc) {

    callContextP context = (callContextP)userData;
    Uns32        stackPointer;

    vmiosRegRead(processor, context->object->sp, &stackPointer);

    // If stack pointer matches then we have found
    // the matching return for this context
    if(stackPointer==context->sp) {

        // get the function result
        Int32 result;
        vmiosRegRead(processor, context->object->result, &result);

        // Report the function return (including elapsed instructions)
        vmiPrintf(
            "*** breturn-plugin: cpu %s: Function %s returned with result %d "
            "(" FMT_64u " instructions)\n",
            vmirtProcessorName(processor),
            TRACE_FUNC,
```

```
            result,
            vmirtGetICount(processor)-context->entryICount
        );

        // Remove this callback
        vmirtRemovePCCallback(processor, thisPC, exitFunc, context);

        // Free the memory allocated for the context
        free(context);
    }
}
```

Notes and Restrictions

1. This function is available only with the Imperas Professional Tools.
2. If there are several callbacks installed with the same `processor`, `simPC`, `watchCB` and `userData` arguments, only one instance will be uninstalled by each call `vmirtRemovePCCallback`.

QuantumLeap Semantics

Synchronizing

9 Accessing and Setting the Program Counter

The simulator does not directly model the processor program counter (to do so would make simulation much slower). Instead, it provides a run time interface to get and set the program counter (that is, to perform a jump) if required. This section describes functions related to program counter management.

9.1 vmirtGetPC

Prototype

```
Addr vmirtGetPC(vmiProcessorP processor);
```

Description

This routine returns the processor program counter.

Example

```
#include “vmi/vmiRt.h”

// routine called to take an exception, jumping to ‘address’
// the current program counter is saved in register CPUX_REG_EPCR
static void vmic_TakeException(cpuxP cpux, Uns32 address) {
    Uns32 simPC = vmirtGetPC((vmiProcessorP)cpux);
    cpuxEnterKernelMode(cpux);
    cpux->regs[CPUX_REG_EPCR] = simPC;
    vmirtSetPC((vmiProcessorP)cpux, address);
}

// handler for read out of bounds
VMI_READ_EXCEPT_FN(cpuxRdPrivExceptionCB) {
    vmic_TakeException((cpuxP)processor, DPF_ADDRESS);
}

// handler for write out of bounds
VMI_WRITE_EXCEPT_FN(cpuxWrPrivExceptionCB) {
    vmic_TakeException((cpuxP)processor, DPF_ADDRESS);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

9.2 vmirtGetPCDS

Prototype

```
Addr vmirtGetPCDS(vmiProcessorP processor, Uns8 *delaySlotOffset);
```

Description

This routine returns the current processor program counter and delay slot offset (using a byref argument, which must be non-`NULL`). It is required in order to correctly model interrupt behavior on processors with delay slots. There are two distinct modes:

1. If the current instruction is *not* a delay slot instruction, the function returns the current program counter and sets the byref argument `delaySlotOffset` to 0.
2. If the current instruction is a delay slot instruction, the function returns the address of the *branch instruction for which this is a delay slot* and sets the byref argument `delaySlotOffset` to *the offset of the delay slot instruction from the branch instruction*. The address of the delay slot instruction can therefore be calculated if required by adding `delaySlotOffset` to the function result.

As an example, on a machine with a 4-byte instruction format and a single delay slot instruction per branch, when called from a delay slot instruction `vmirtGetPCDS` will return the address of the branch instruction and set `delaySlotOffset` to 4.

Example

```
#include “vmi/vmiRt.h”

// routine called to take an exception, jumping to ‘address’
// the current program counter is saved in register CPUX_REG_EPCR
static void vmic_TakeException(cpuxP cpux, Uns32 address) {

    Uns8  simD;
    Uns32 simPC = vmirtGetPCDS((vmiProcessorP)cpux, &simD);

    cpuxEnterKernelMode(cpux);
    cpux->regs[CPUX_REG_EPCR] = simPC;
    vmirtSetPC((vmiProcessorP)cpux, address);

    // set SR[DSX] for exception in a delay slot
    if(simD) {
        cpux->regs[CPUX_REG_SR] |= SPR_SR_DSX;
    }
}

// handler for read out of bounds
VMI_READ_EXCEPT_FN(cpuxRdPrivExceptionCB) {
    vmic_TakeException(cpuxP)processor, DPF_ADDRESS);
}

// handler for write out of bounds
VMI_WRITE_EXCEPT_FN(cpuxWrPrivExceptionCB) {
    vmic_TakeException((cpuxP)processor, DPF_ADDRESS);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics
Non-self-synchronizing

9.3 vmirtSetPC

Prototype

```
void vmirtSetPC(vmiProcessorP processor, Addr newPC);
```

Description

This routine sets the current program counter. This effectively performs a *branch* operation from within a run time call. When called from an embedded function call from translated native code, the branch takes effect just after the return from the embedded function call. Any code after the embedded function call in the translated native code is not executed.

This function differs from `vmirtSetPCException` in that exception watchpoints installed for the processor will *not* be triggered (see the *Imperas OVPsim and CpuManager User Guide* for more information about watchpoints).

Example

This example is from the ARM processor model.

```
#include “vmi/vmiRt.h”

static void doCheckHandler(armP arm, Uns32 handler) {

    arm->itStateRT        = 0;
    arm->regs[ARM_REG_LR] = (getPC(arm)+ARM_PC_DELTA(arm)) | IN_THUMB_MODE(arm);

    vmirtSetPC((vmiProcessorP)arm, CP_REG_UNS32(arm, TEEHBR)+handler);
}
```

Notes and Restrictions

1. Branch instructions implemented by `vmirtSetPC` are much less efficient than morph time branches emitted with `vmimtUncondJump` and similar functions (see the *VMI Morph Time Function Reference*). Therefore, use `vmirtSetPC` only in circumstances where the jump behavior cannot be described by morph time calls (for example, in exception-like handlers as above).

QuantumLeap Semantics

Non-self-synchronizing

9.4 vmirtSetPCDS

Prototype

```
void vmirtSetPCDS(
    vmiProcessorP processor,
    Addr          newPC1,
    Addr          newPC2,
    vmiPostSlotFn slotCB
);
```

Description

This routine sets the current program counter to `newPC1`. This effectively performs a *branch* operation from within a run time call. Once the instruction at `newPC1` has executed, control is immediately transferred to `newPC2`. When called from an embedded function call from translated native code, the branch to `newPC1` takes effect just after the return from the embedded function call. Any code after the embedded function call in the translated native code is not executed.

This function is typically used in processor models that support return from an exception handler directly to a delay slot instruction.

Example

This example is from the ARC processor model, which supports return from an exception handler directly to a delay slot instruction. In such cases, a status flag `status32.DE` is non-zero and the post-delay-slot address is in the `bta` register:

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

inline static void setPC(arcP arc, Uns32 pc) {
    if(AUX_FIELD(arc, status32, DE)) {
        vmirtSetPCDS((vmiProcessorP)arc, setPCState(arc, pc), AUX_REG(arc, bta), 0);
    } else {
        vmirtSetPC((vmiProcessorP)arc, setPCState(arc, pc));
    }
}
```

Notes and Restrictions

1. Branch instructions implemented by `vmirtSetPCDS` are much less efficient than morph time branches emitted with `vmimtUncondJump` and similar functions (see the *VMI Morph Time Function Reference*). Therefore, use `vmirtSetPCDS` only in circumstances where the jump behavior cannot be described by morph time calls (for example, in exception-like handlers as above).

QuantumLeap Semantics

Non-self-synchronizing

9.5 vmirtSetPCException

Prototype

```
void vmirtSetPCException(vmiProcessorP processor, Addr newPC);
```

Description

This routine sets the current program counter in order to take an exception. This effectively performs a *branch* operation from within a run time call. When called from an embedded function call from translated native code, the branch takes effect just after the return from the embedded function call. Any code after the embedded function call in the translated native code is not executed.

This function differs from `vmirtSetPC` in that any exception watchpoints installed for the processor will be triggered before the next simulated instruction is executed (see the *Imperas OVPsim and CpuManager User Guide* for more information about watchpoints).

Example

This example is from the ARM processor model.

```
#include “vmi/vmiRt.h”

static void jumpToExceptionVector(armP arm, armException exception) {

    Uns32 vectorBase = CP_FIELD(arm, SCTLR, V) ? ARM_VECTOR_HIGH : 0x0;
    Uns32 vector     = exceptions[exception].code;

    arm->exception = exception;

    vmirtSetPCException((vmiProcessorP)arm, vector|vectorBase);
}
```

Notes and Restrictions

1. Branch instructions implemented by `vmirtSetPCException` are much less efficient than morph time branches emitted with `vmimtUncondJump` and similar functions (see the *VMI Morph Time Function Reference*).

QuantumLeap Semantics

Non-self-synchronizing

9.6 vmirtSetPCFlushTarget

Prototype

```
void vmirtSetPCFlushTarget(vmiProcessorP processor, Addr newPC);
```

Description

This routine is identical to `vmirtSetPC` except that it discards the target address in the current code dictionary before performing the jump. In other words, it forces code for the target address to be translated afresh using the morph time interface defined in the *VMI Morph Time Function Reference*.

This function is required in order to implement some classes of exception correctly. As an example, suppose that processor behavior is such that any pending exception is taken after the *next* instruction fetch and furthermore that an instruction generating such an exception is located in the middle of a sequence of instructions comprising a code block (see the *VMI Morph Time Function Reference* for a definition of a code block and how the simulator determines the instructions that are represented in each one). Suppose that, when executing a code block, an embedded model callback changes the system state so that there is now an exception that should be handled after the next instruction fetch. It would obviously be incorrect to do nothing in this case, as the simulator would then execute remaining instructions in the current code block instead of taking the exception. It would also be wrong to use `vmirtSetPC` to jump directly to the exception handler because that would not model the next instruction fetch that should be performed before the exception is taken. The solution is to use `vmirtSetPCFlushTarget` to jump to the *next* instruction. This will have the following effect:

1. The target code block (which is also the currently executing code block in this case) will be discarded. This means that no previously-translated instructions after the current instruction in this code block will be executed.
2. The processor program counter will be set to the next instruction address.
3. The processor execution exception handler (`ifetchExceptionCB` in the processor attribute structure) will be called for the next instruction address. This should call `vmirtSetPC` to reset the processor program counter to the appropriate simulated exception handler address (and perform any other required actions, such as entering kernel mode).

In this way it is possible to correctly model exceptions taken on the next instruction address. The example below outlines an embedded function call that might be inserted to perform the first of the three steps described above.

Example

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

static void vmic_TakeInterruptNext(cpuxP cpux) {

    if(exceptionPending(cpux)) {

        vmiProcessorP processor = (vmiProcessorP)cpux;
        Addr          thisPC    = vmirtGetPC(processor);
        Addr          nextPC    = getNextPC(thisPC);

        vmirtSetPCFlushTarget(processor, nextPC);
    }
}
```

Notes and Restrictions

1. Branch instructions implemented by `vmirtSetPCFlushTarget` are much less efficient than morph time branches emitted with `vmimtUncondJump` and similar functions (see the *VMI Morph Time Function Reference*). Therefore, use `vmirtSetPCFlushTarget` only in circumstances where the jump behavior cannot be described by morph time calls (for example, in exception handlers as described above).
2. `vmirtSetPCFlushTarget` is also less efficient than `vmirtSetPC` since it forces cached native code blocks to be discarded and retranslated.
   Only use `vmirtSetPCFlushTarget` when it is *mandatory* that the target is regenerated to get correct behavior

QuantumLeap Semantics

Synchronizing

9.7 vmirtSetPCFlushDict

Prototype

```
void vmirtSetPCFlushDict(vmiProcessorP processor, Addr newPC);
```

Description

This routine sets the current program counter and flushes all code blocks from the current dictionary. It is required when the processor has undergone some state change that would make any previously-generated code blocks in the current dictionary potentially invalid.

It is usually much more efficient to implement modal behavior of this kind by having multiple code dictionaries to represent different modes (see the section *Simulated Memory Access* elsewhere in this document for a detailed explanation of code dictionaries and their interaction with processor modes). However, it occasionally makes sense to implement mode changes by flushing a dictionary instead. An example might be simulation of a processor with a hardware mode that enables a trace buffer to record a sequence of recent branch addresses. In almost all normal operation, the trace buffer is disabled. In the very rare case that the trace buffer is enabled and must be simulated, every branch must be preceded in translated native code by an embedded call to simulate the buffer. In these circumstances, it is often more convenient to simply flush the current dictionary when the trace buffer is enabled or disabled by a processor state change instead of incurring the overhead of another dictionary to model the trace buffer.

Example

```
#include “vmi/vmiRt.h”

// embedded call made on status register (CPUX_SR) change
static void vmic_TestTraceModeChange(cpuxP cpux) {

    Bool simTraceEnabled = cpux->regs[CPUX_SR] & HW_TRACE_MODE;

    if(simTraceEnabled!=cpux->simTraceEnabled) {

        vmiProcessorP processor = (vmiProcessorP)cpux;
        Addr          thisPC    = vmirtGetPC(processor);
        Addr          nextPC    = getNextPC(thisPC);

        vmirtSetPCFlushDict((vmiProcessorP)cpux, nextPC);
        cpux->simTraceEnabled = simTraceEnabled;
    }
}
```

Notes and Restrictions

1. `vmirtSetPCFlushDict` is very slow because all code blocks in a dictionary must be discarded and retranslated. Use `vmirtSetPCFlushDict` only in very rare circumstances as described above.
2. See also the related routine `vmirtFlushDict`, which allows the dictionary to be flushed without changing the program counter.

QuantumLeap Semantics

Synchronizing

10 Processor Modes

Processor models can be modal – for example, a processor may support both *user* and *kernel* modes, in which some instructions behave differently, or in which memory access permissions have different restrictions. Modal models can be written in three different ways:

1. All behavior that is mode-dependent can be modeled explicitly in the morphed code emitted for each instruction. For example, code for a privileged instruction might explicitly test a processor mode bit, and, depending on the value of that bit, either implement the privileged behavior or implement a jump to an exception vector. This approach is not recommended because the generated code is usually much more verbose than necessary and contains many more branches than necessary, both of which badly degrade performance.
2. The model can be designed to support multiple *code dictionaries*, one for each mode, with calls to `vmirtSetMode` (documented in this section) to switch modes on processor state changes. This is the recommended method for implementing modal code in most cases. See the *OVP Processor Modeling Guide* for extensive examples of the use of processor modes.
3. In cases where there are only minor differences in behavior between what would otherwise be different modes, and where it is very likely that translated code will always be executed with the processor in the *same* mode, block masks can be used to implement modal instructions without the cost of additional dictionaries and mode switches.

10.1 vmirtSetMode

Prototype

```
Bool vmirtSetMode(vmiProcessorP processor, Uns32 dictionaryIndex);
```

Description

Processors are often modal – for example, many have *user* and *kernel* modes in which instruction implementations are different. Register writes may succeed in kernel mode but cause an instruction privilege trap in user mode. In addition, virtual-to-physical memory maps or access permissions often differ between modes.

The simulator allows such modal processors to be simulated efficiently by supporting multiple *dictionary modes*. Each dictionary mode has an associated *dictionary* of JIT-translated code blocks and *memory domain* defining virtual-to-physical address translations. The currently-active dictionary mode can be changed by calling `vmirtSetMode` to indicate the new code dictionary and associated memory domain to be used. This is explained in detail in the section *Simulated Memory Management* elsewhere in this document.

The argument `dictionaryIndex` is an index number of the new dictionary mode. Valid index numbers for a processor are defined by the processor attribute structure: see *Simulated Memory Management* elsewhere in this document for more information about this.

The new code dictionary will be used when translating the next instruction executed after the call to `vmirtSetMode`.

Example

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

typedef enum cpuxModeE {
    CPM_KERNEL = 0,
    CPM_USER   = 1
} cpuxMode;

// embedded call made on mode change
static void vmic_SwitchMode(cpuxP cpux, cpuxMode newMode) {

    if(vmirtGetMode((vmiProcessorP)cpux)!=newMode) {
        // switch to the new operating mode – this will change
        // dictionary and domain.
        vmirtSetMode((vmiProcessorP)cpux, newMode);
    }
}
```

Notes and Restrictions

1. The new dictionary will be used for the next simulated instruction after the current instruction. Deferring the mode change by more than one instruction is not supported.

2. If the mode switch succeeded, the function returns `True`. If the processor does not support the indexed mode, the function returns `False` and the processor state is not updated.
3. The mapping between *processor modes* as defined in the processor specification and *dictionary modes* used by the simulator is not necessarily one-to-one. For example, it could be the case that the memory map used in a single *processor-defined* mode may change greatly when configuration registers change: in the ARM processor, enabling or disabling the TLB is a specific example. In such a case, a single processor-defined mode is best simulated using *two* dictionary modes (a TLB-enabled mode and a TLB-disabled mode).
4. To access processor-defined modes, see functions `vmirtGetCurrentMode` and `vmirtGetNextMode`.

QuantumLeap Semantics

Non-self-synchronizing

10.2 vmirtGetMode

Prototype

```
Uns32 vmirtGetMode(vmiProcessorP processor);
```

Description

Processors are often modal – for example, many have *user* and *kernel* modes in which instruction implementations are different. Register writes may succeed in kernel mode but cause an instruction privilege trap in user mode. In addition, virtual-to-physical memory maps or access permissions often differ between modes.

The simulator allows such modal processors to be simulated efficiently by supporting multiple *dictionary modes*. Each dictionary mode has an associated *dictionary* of JIT-translated code blocks and *memory domain* defining virtual-to-physical address translations.

This function returns the current processor *dictionary index*. The dictionary index can be changed using related function `vmirtSetMode`.

Example

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

typedef enum cpuxModeE {
    CPM_KERNEL = 0,
    CPM_USER   = 1
} cpuxMode;

// embedded call made on mode change
static void vmic_SwitchMode(cpuxP cpux, cpuxMode newMode) {

    if(vmirtGetMode((vmiProcessorP)cpux)!=newMode) {
        // switch to the new operating mode – this will change
        // dictionary and domain.
        vmirtSetMode((vmiProcessorP)cpux, newMode);
    }
}
```

Notes and Restrictions

1. The mapping between *processor modes* as defined in the processor specification and *dictionary modes* used by the simulator is not necessarily one-to-one (see notes associated with function `vmirtSetMode` for more information).
2. To access processor-defined modes, see functions `vmirtGetCurrentMode` and `vmirtGetNextMode`.

QuantumLeap Semantics

Non-self-synchronizing

10.3 vmirtSetBlockMask

Prototype

```
void vmirtSetBlockMask(vmiProcessorP processor, Uns32 blockMask);
```

Description

In cases where there are only minor differences in behavior between what would otherwise be different modes, and where it is very likely that translated code will always be executed with the processor in the *same* mode, block masks can be used to implement modal instructions without the cost of additional dictionaries and mode switches.

A processor has a notion of a *current block mask*, which is an `Uns32` value. The current block mask is set by `vmirtSetBlockMask`.

When creating JIT-translated code using the Imperas Morph Time Function interface, code can be emitted to validate the current block mask using function `vmimtValidateBlockMask`, which has the following prototype:

```
void vmirtValidateBlockMask(Uns32 blockMask);
```

This causes code to be inserted in the JIT compiled block with the following effect:

1. The `blockMask` specified by `vmimtValidateBlockMask` is compared with the current block mask using a bitwise AND operation.
2. If the result is *the same as* the `blockMask` specified by `vmimtValidateBlockMask`, the code block is executed and there is no further special action.
3. If the result *is different to* the `blockMask` specified by `vmimtValidateBlockMask`, the code block is deleted and remorphed.

In cases where code blocks are almost always executed with the block mask that was in effect when they were created, this results in very efficient simulation without the cost of an additional code dictionary that might otherwise be required.

Example

The standard ARM processor model uses block masks to control code generated for `LDM` and `STM` instructions that access user-mode registers from other modes. The model is designed so that, in each processor mode, registers r0 to r15 for that mode are always held in the current register bank, and registers not normally visible in that mode are held in subsidiary model registers. The model is written so that it accesses either the normal or subsidiary registers as applicable for the current processor mode. It uses a `blockMask` to ensure that the assumption made about the current mode when code was generated is still valid when the code is executed.

In `armMorph.c`, function `getBankedReg` returns a `vmiReg` structure for a register, using `vmimtValidateBlockMask` to check that the processor is in the correct mode (function `armEmitValidateBlockMask` is a wrapper round `vmimtValidateBlockMask`):

```
// Macro returning base register (and validating block mode)
#define GET_BASE_REG(_BM) \
    armEmitValidateBlockMask(_BM);  \
    return VMI_NOREG

// Macro returning banked register (and validating block mode)
#define GET_BANKED_REG(_BM, _N, _SET) \
    armEmitValidateBlockMask(_BM);  \
    return ARM_BANK_REG(_N, _SET)

// If 'userRegs' is True and the register index specifies a banked register,
// return a vmiReg structure for the banked register; otherwise, return
// VMI_NOREG. Also, validate the current block mode for registers r8 to r14.
static vmiReg getBankedReg(armMorphStateP state, Uns32 r, Bool userRegs) {

    if(!userRegs) {

        return VMI_NOREG;

    } else switch(state->arm->CPSR.fields.mode) {

        case ARM_CPSR_FIQ:
            switch(r) {
                case 8:  GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       8,  fiq);
                case 9:  GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       9,  fiq);
                case 10: GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       10, fiq);
                case 11: GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       11, fiq);
                case 12: GET_BANKED_REG(ARM_BM_R8_R12_FIQ,       12, fiq);
                case 13: GET_BANKED_REG(ARM_BM_R13_R14_SPSR_FIQ, 13, fiq);
                case 14: GET_BANKED_REG(ARM_BM_R13_R14_SPSR_FIQ, 14, fiq);
                default: return VMI_NOREG;
            }
            break;

        case ARM_CPSR_IRQ:
            switch(r) {
                case 8:  GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 9:  GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 10: GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 11: GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 12: GET_BASE_REG(ARM_BM_R8_R12_BASE);
                case 13: GET_BANKED_REG(ARM_BM_R13_R14_SPSR_IRQ, 13, irq);
                case 14: GET_BANKED_REG(ARM_BM_R13_R14_SPSR_IRQ, 14, irq);
                default: return VMI_NOREG;
            }
            break;

        … code for remaining modes follows …
    }
}
```

Function `armSetBlockMask` in file `armUtils.c` is called to set the current block mask on a mode switch:

```
// Return current processor block mask
static armBlockMask getBlockMask(armP arm) {
    switch(arm->CPSR.fields.mode) {
        case ARM_CPSR_USER:       return ARM_BM_USER;
        case ARM_CPSR_FIQ:        return ARM_BM_FIQ;
        case ARM_CPSR_IRQ:        return ARM_BM_IRQ;
        case ARM_CPSR_SUPERVISOR: return ARM_BM_SUPERVISOR;
        case ARM_CPSR_ABORT:      return ARM_BM_ABORT;
        case ARM_CPSR_UNDEFINED:  return ARM_BM_UNDEFINED;
        case ARM_CPSR_SYSTEM:     return ARM_BM_SYSTEM;
        default:                  return ARM_BM_USER;
    }
}
```

```
// Update processor block mask
void armSetBlockMask(armP arm) {
    vmirtSetBlockMask((vmiProcessorP)arm, getBlockMask(arm));
}
```

The `blockMask` values are specified by a `typedef` in file `armMode.h`:

```
typedef enum armBlockMaskE {

    ARM_BM_R8_R12_BASE      = 0x01,
    ARM_BM_R8_R12_FIQ       = 0x02,
    ARM_BM_R13_R14_BASE     = 0x04,
    ARM_BM_R13_R14_SPSR_FIQ = 0x08,
    ARM_BM_R13_R14_SPSR_IRQ = 0x10,
    ARM_BM_R13_R14_SPSR_SVC = 0x20,
    ARM_BM_R13_R14_SPSR_ABT = 0x40,
    ARM_BM_R13_R14_SPSR_UND = 0x80,

    ARM_BM_USER       = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_BASE,
    ARM_BM_SYSTEM     = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_BASE,
    ARM_BM_FIQ        = ARM_BM_R8_R12_FIQ  | ARM_BM_R13_R14_SPSR_FIQ,
    ARM_BM_IRQ        = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_SPSR_IRQ,
    ARM_BM_SUPERVISOR = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_SPSR_SVC,
    ARM_BM_ABORT      = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_SPSR_ABT,
    ARM_BM_UNDEFINED  = ARM_BM_R8_R12_BASE | ARM_BM_R13_R14_SPSR_UND

} armBlockMask;
```

Notes and Restrictions

1. Since a block mask match failure causes a code block to be deleted and retranslated, block masks can make simulation much slower if this often happens. They should therefore only be used in cases where it can be inferred that mismatches are very unlikely to occur.

QuantumLeap Semantics

Non-self-synchronizing

10.4 vmirtGetBlockMask

Prototype

```
Uns32 vmirtGetBlockMask(vmiProcessorP processor);
```

Description

This function returns the current processor block mask. The block mask can be changed using related function `vmirtSetBlockMask`.

Example

```
#include “vmi/vmiRt.h”

// embedded call made to include extra bits in block mask
static void vmic_AddToBlockMask(cpuxP cpux, Uns32 addBits) {
    Uns32 blockMask = vmirtGetBlockMask((vmiProcessorP)cpux);
    vmirtSetBlockMask((vmiProcessorP)cpux, blockMask|addBits);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

11 Floating Point Operations

VMI floating point operation support has been extensively revised with version 5.13.0. A detailed description of this enhanced support can be found in the *VMI Morph Time Function Reference*.

This section describes the functions that are used to configure a model floating point unit. Refer to the *VMI Morph Time Function Reference* for all other details of floating point instruction simulation.

11.1 vmirtConfigureFPU

Prototype

```
void vmirtConfigureFPU(
    vmiProcessorP processor,
    vmiFPConfigCP config
);
```

Description

This function is used to specify a default configuration for floating point operations for which no instruction-specific configuration has been specified. Configuration information is given in a static constant structure of type `vmiFPConfig`, defined in `vmiTypes.h` as follows:

```
typedef struct vmiFPConfigS {
    Uns32               QNaN32;
    Uns64               QNaN64;
    Uns16               indeterminateUns16;
    Uns32               indeterminateUns32;
    Uns64               indeterminateUns64;
    vmiFPQNaN32ResultFn QNaN32ResultCB;
    vmiFPQNaN64ResultFn QNaN64ResultCB;
    vmiFPInd16ResultFn  indeterminate16ResultCB;
    vmiFPInd32ResultFn  indeterminate32ResultCB;
    vmiFPInd64ResultFn  indeterminate64ResultCB;
    vmiFPTinyResultFn   tinyResultCB;
    vmiFPTinyArgumentFn tinyArgumentCB;
    vmiFP32ResultFn     fp32ResultCB;
    vmiFP64ResultFn     fp64ResultCB;
    vmiFPArithExceptFn  fpArithExceptCB;
    vmiFPFlags          suppressFlags;
    Bool                stickyFlags;
    Bool                fzClearsPF;
    Bool                tininessAfterRounding;
} vmiFPConfig;
```

The native floating point operations described in the *VMI Morph Time Function Reference* normally follow exact IEEE Standard 754 or Intel x87 semantics, depending on the floating point operand type. This function allows the FPU for a processor model to be configured to be non-compliant in some respects, so that the native floating point operation code may be used even when a processor model is not fully IEEE compliant.

The structure fields are as follows:

1. `QNaN32` specifies the bit pattern produced when a floating point operation generates a 32-bit `QNaN` result. Normally this should be `0x7fc00000`, but older versions of IEEE Standard 754 permit the most significant bit of the significand to be reversed for `QNaN` and `SNaN`. On a processor such as the MIPS where `QNaN` and `SNaN` values are indeed reversed, a different value should be specified (for example, `0x7fbfffff` for MIPS).
2. `QNaN64` specifies the bit pattern produced when a floating point operation generates a 64-bit `QNaN` result. Normally this should be `0x7ff8000000000000ULL`, but a processor such as the MIPS where `QNaN` and `SNaN` values are reversed, a

different value should be specified (for example, `0x7ff7fffffffffffULL` for MIPS).

3. `QNaN32ResultCB` is a callback function which, if given, is called whenever a 32-bit `QNaN` result is generated to give the processor model the opportunity to modify the resulting `QNaN` value. See the *VMI Morph Time Function Reference* for more information about `QNaN` result handlers.
4. `QNaN64ResultCB` is a callback function which, if given, is called whenever a 64-bit `QNaN` result is generated to give the processor model the opportunity to modify the resulting `QNaN` value. See the *VMI Morph Time Function Reference* for more information about `QNaN` result handlers.
5. `indeterminateUns16` specifies the bit pattern produced when a floating point operation generates a 16-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x8000`.
6. `indeterminateUns32` specifies the bit pattern produced when a floating point operation generates a 32-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x80000000`.
7. `indeterminateUns64` specifies the bit pattern produced when a floating point operation generates a 64-bit indeterminate integer result. For processors compliant with IEEE Standard 754, this should be `0x8000000000000000ULL`.
8. `indeterminate16ResultCB` is a callback function which, if given, is called whenever a 16-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.
9. `indeterminate32ResultCB` is a callback function which, if given, is called whenever a 32-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.
10. `indeterminate64ResultCB` is a callback function which, if given, is called whenever a 64-bit indeterminate result is generated to allow the processor model to provide the required indeterminate value. See the *VMI Morph Time Function Reference* for more information about indeterminate result handlers.
11. `tinyResultCB` is a callback function which, if given, is called whenever a tiny (denormalized) result is generated to give the processor model the opportunity to modify the resulting tiny value (or take any other action). See the *VMI Morph Time Function Reference* for more information about tiny result handlers.
12. `tinyArgumentCB` is a callback function which, if given, is called whenever a denormalized argument is detected to give the processor model the opportunity to modify the argument value (or take any other action). See the *VMI Morph Time Function Reference* for more information about tiny argument handlers.
13. `fp32ResultCB` is a callback function which, if given, is called whenever a 32-bit floating point result is generated to allow the processor model to modify the result value or flags. Result callbacks are typically specified only for instruction-specific configurations, so this field should usually be `NULL` for a configuration used with `vmirtConfigureFPU`. See the *VMI Morph Time Function Reference* for more information about result handlers.

14. `fp64ResultCB` is a callback function which, if given, is called whenever a 64-bit floating point result is generated to allow the processor model to modify the result value or flags. Result callbacks are typically specified only for instruction-specific configurations, so this field should usually be `NULL` for a configuration used with `vmirtConfigureFPU`. See the *VMI Morph Time Function Reference* for more information about result handlers.
15. `fpArithExceptCB` is an exception handler callback function which is called whenever a floating point operation generates unmasked exceptions. The exception handler callback will typically update processor state and cause a jump to a vector address. See the *VMI Morph Time Function Reference* for more information about the exception handler callback.
16. `suppressFlags` is a field if type `vmiFPFlags` which enables flags generated by a floating point operation to be suppressed: any flag set to 1 in the bitmask will be masked out of the operation result flags.
17. `stickyFlags` is a boolean field which specifies whether the operation result flags should replace any current value of the output flags (if `False`) or whether operation flags should be combined with existing flags using bitwise-or (if `True`).
18. `fzClearsPF` is a boolean field that should be `True` if the processor implements *flush-to-zero* mode and when denormal results are flushed to zero *the precision flag in the floating point status word is not set*. If the processor does *not* implement flush-to-zero mode, or if the precision flag should be *set* when results are flushed to zero, then the argument should be `False`. Most floating point implementations set the precision flag when a denormal result is flushed to zero (e.g. x86, MIPS) but some do not (e.g. ARM).
19. `tininessAfterRounding` is a boolean field that indicates whether tininess should be detected before rounding a result or afterwards. This affects behavior for intermediate results that round to a minimum normal value of greater absolute magnitude. The boolean affects all floating point operations using IEEE types.

Example

This example is extracted from the standard PowerPC model. Note that fields in the static `config` structure are initialized by name.

```
#include “vmi/vmiRt.h”
#include “vmi/vmiTypes.h”

#define PPC32_SFP_QNAN_MASK 0x7FFC00000
#define PPC32_DFP_QNAN_MASK 0x7FF8000000000000ULL

void ppc32ConfigureFPU(ppc32P ppc32) {

    // FPU configuration
    const static vmiFPConfig config = {
        .QNaN32                  = PPC32_SFP_QNAN_MASK,
        .QNaN64                  = PPC32_DFP_QNAN_MASK,
        .QNaN32ResultCB          = QNaN32ResultCB,
        .QNaN64ResultCB          = QNaN64ResultCB,
        .indeterminate32ResultCB = handleIndeterminate32,
        .indeterminate64ResultCB = handleIndeterminate64,
        .tinyResultCB            = handleTinyResult,
        .tinyArgumentCB          = handleTinyArgument,
        .fpArithExceptCB         = ppc32FPArithExcept
    };
```

```
    // set up QNaN values and handlers
    vmirtConfigureFPU((vmiProcessorP)ppc32, &config);

    // initialize floating point control word
    ppc32UpdateFPControlWord(ppc32);
}
```

Notes and Restrictions

1. This function is intended to be called once only in the processor constructor. It flushes all processor code dictionaries to reflect the new FPU configuration. *Calling the function frequently during simulation will severely degrade performance*.
2. The default configuration can be overridden on an instruction-specific basis. See the *VMI Morph Time Function Reference* for more information.

QuantumLeap Semantics

Non-self-synchronizing

11.2 vmirtGetFPControlWord

Prototype

```
vmiFPControlWord vmirtGetFPControlWord(vmiProcessorP processor);
```

Description

This function returns the currently-active floating point control word for the processor. This is specified by a structure in `vmiTypes.h`:

```
typedef struct vmiFPControlWordS {

                        // INTERRUPT MASKS
    Uns32   IM  : 1;    // invalid operation mask
    Uns32   DM  : 1;    // denormal mask
    Uns32   ZM  : 1;    // divide-by-zero mask
    Uns32   OM  : 1;    // overflow mask
    Uns32   UM  : 1;    // underflow mask
    Uns32   PM  : 1;    // precision mask

                        // ROUNDING AND PRECISION
    Uns32   RC  : 2;    // rounding control
    Bool    FZ  : 1;    // flush to zero
    Uns32   DAZ : 1;    // denormals are zeros flag

} vmiFPControlWord;
```

The first six fields are *interrupt masks* that specify whether a floating-point arithmetic exception of the indicated type should by masked. If the exception is masked (the bit is 1), the exception will be ignored. If the exception is unmasked (the bit is 0), any exception of the indicated type will be signaled by calling the processor arithmetic exception handler (defined with the `VMI_ARITH_EXCEPT_FN` macro in `vmiAttrs.h` and passed as the `arithExceptCB` field of the processor `vmiIASAttr` structure).

The `RC` field specifies the *rounding control* to use when arithmetic results cannot be exactly represented. The field value should be one of the first four members of the `vmiFPRC` enumeration:

```
typedef enum {

    // these values are valid in both conversion functions and in the rounding
    // control field of vmiFPControlWord, below
    vmi_FPR_NEAREST = 0,    // round towards nearest (even)
    vmi_FPR_NEG_INF = 1,    // round towards negative infinity
    vmi_FPR_POS_INF = 2,    // round towards positive infinity
    vmi_FPR_ZERO    = 3,    // round towards zero

    // this value is valid in conversion functions only
    vmi_FPR_CURRENT = 4     // use currently-active rounding control

} vmiFPRC;
```

The `FZ` field specifies that denormal results should be flushed to zero. The `DAZ` field specifies that denormal arguments should be treated as zero. Neither of these modes are IEEE 754 compliant, but most processors implement variations of these options.

Notes and Restrictions
None.

QuantumLeap Semantics
Non-self-synchronizing

11.3 vmirtSetFPControlWord

Prototype

```
void vmirtSetFPControlWord(
    vmiProcessorP    processor,
    vmiFPControlWord fpcw
);
```

Description

This function sets the currently-active floating point control word for the processor. This is specified by a structure in `vmiTypes.h`; see `vmirtGetFPControlWord` for details.

Example

This example is extracted from the standard PowerPC model.

```
#include “vmi/vmiRt.h”
#include “vmi/vmiTypes.h”

void ppc32UpdateFPControlWord(ppc32P ppc32) {
    //
    // Set the new control word bits
    //
    vmiFPControlWord cw = {0};

    //
    // Interrupt Masks (register bits are enables, invert sense)
    //
    Uns8 FE = getMSR_FE(ppc32);

    if (FE == 0) {
        cw.IM = 1;  // Invalid Operation mask
        cw.ZM = 1;  // Divide by Zero mask
        cw.OM = 1;  // Overflow mask
        cw.UM = 1;  // Underflow mask
        cw.PM = 1;  // Precision mask
    } else {
        cw.IM = ~(ppc32->FPSCR.bits.VE);  // Invalid Operation mask
        cw.ZM = ~(ppc32->FPSCR.bits.ZE);  // Divide by Zero mask
        cw.OM = ~(ppc32->FPSCR.bits.OE);  // Overflow mask
        cw.UM = ~(ppc32->FPSCR.bits.UE);  // Underflow mask
        cw.PM = ~(ppc32->FPSCR.bits.XE);  // Precision mask
    }
    cw.DM = 1; // Denormal mask

    //
    // Rounding and precision
    //
    switch (ppc32->FPSCR.bits.RN) {
    case 0:
        cw.RC = vmi_FPR_NEAREST;
        break;
    case 1:
        cw.RC = vmi_FPR_ZERO;
        break;
    case 2:
        cw.RC = vmi_FPR_POS_INF;
        break;
    case 3:
        cw.RC = vmi_FPR_NEG_INF;
        break;
    }

    //
```

```
    //  TODO: Flush to zero
    //
    cw.FZ = 0;

    //
    //  TODO: Denormals are zeros
    //
    cw.DAZ = 0;

    //
    // Assign Control Word
    //
    vmirtSetFPControlWord((vmiProcessorP)ppc32, cw);
}
```

Notes and Restrictions
None

QuantumLeap Semantics
Non-self-synchronizing

11.4 vmirtRestoreFPState

Prototype

```
void vmirtRestoreFPState(vmiProcessorP processor);
```

Description

The simulator core uses native floating point instructions to implement most floating point operations (see the *VMI Morph Time Function Reference* guide for more details on these operations). As these instructions execute, the native floating point state is modified to reflect exceptions and rounding modes required to implement that operation. As a consequence, the native floating point state seen in an embedded call is by default dependent on any floating point operations that the processor has executed prior to the embedded call.

This function can be used in an embedded call to restore the native floating point state to the state when the simulator was invoked. Use this function prior to any code in an embedded call that uses floating point instructions to avoid unexpected behavior.

Example

This example is extracted from the standard PowerPC model. This model requires conversion of a double-precision floating point operand to a single-precision result in an embedded call. The function definition is as follows:

```
#include “vmi/vmiRt.h”
#include “vmi/vmiTypes.h”

static Uns32 vmic_ConvDoubleToSingle(ppc32P ppc32, Uns64 value) {

    doubleT FRS = { .r64 = value };
    singleT WORD;

    vmirtRestoreFPState((vmiProcessorP)ppc32);

    WORD.f32 = FRS.f64;

    return WORD.r32;
}
```

Notes and Restrictions

None

QuantumLeap Semantics

Non-self-synchronizing

12 Connection Operations

Processor models may have *connections* associated with them. Connections are used to implement direct communication channels between processors. These communication channels allow the processors to communicate without sharing memory. Currently, the only form of connection object supported is a FIFO queue.

This section describes routines that are used to establish connections between processors at initialization time, and to send and receive data using connection objects using run time calls.

The *VMI Morph Time Function Reference* describes functions that are used to emit code to send and receive data, which may be more convenient to use in some circumstances. Those functions are mentioned below for comparative purposes.

12.1 vmirtConnGetInput

Prototype

```
vmiConnInputP vmirtConnGetInput(
    vmiProcessorP processor,
    const char   *connPortName,
    Uns32         width
);
```

Description

Deprecated. See section 4.4 for an example of how to create and initialize connection ports.

QuantumLeap Semantics

Non-self-synchronizing

12.2 vmirtConnGetOutput

Prototype

```
vmiConnOutputP vmirtConnGetOutput(
    vmiProcessorP processor,
    const char   *connPortName,
    Uns32         width
);
```

Description

Deprecated. See section 4.4 for an example of how to create and initialize connection ports.

QuantumLeap Semantics

Non-self-synchronizing

12.3 vmirtConnGetInputInfo

Prototype

```
void vmirtConnGetInputInfo(vmiConnInputP conn, vmiConnInfoP info);
```

Description

This function returns information about the current state of the input connection container object `conn` by filling a structure of type `vmiConnInfo`. The structure is defined as follows:

```
typedef struct vmiConnInfoS {
    Uns32 words;        // maximum number of entries the connection can hold
    Uns32 numFilled;    // number of filled entries
    Uns32 numEmpty;     // number of empty entries
    Uns32 bits;         // the width of each entry in bits
} vmiConnInfo, *vmiConnInfoP;
```

Example

```
#include "vmi/vmiMessage.h"
#include "vmi/vmiRt.h"
#include "vmi/vmiTypes.h"

// processor structure definition
typedef struct cpuxS {
    vmiConnInputP inputConn;        // input connection (see section 2.3)
} cpux, *cpuxP;

// Print debug information about the processor input connection
static void vmic_InputConnDebug(cpuxP cpux) {

    // fill a local structure with connection information
    vmiConnInfo info;
    vmirtConnGetInputInfo(cpux->inputConn, vmiConnInfoP info);

    vmiPrintf("processor %s:\n", vmirtProcessorName((vmiProcessorP)cpux));
    vmiPrintf("  words     = %u\n", info.words);
    vmiPrintf("  numFilled = %u\n", info.numFilled);
    vmiPrintf("  numEmpty  = %u\n", info.numEmpty);
    vmiPrintf("  bits      = %u\n", info.bits);
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

12.4 vmirtConnGetOutputInfo

Prototype

```
void vmirtConnGetOutputInfo(vmiConnOutputP conn, vmiConnInfoP info);
```

Description

This function returns information about the current state of the output connection container object `conn` by filling a structure of type `vmiConnInfo`. The structure is defined as follows:

```
typedef struct vmiConnInfoS {
    Uns32 words;        // maximum number of entries the connection can hold
    Uns32 numFilled;    // number of filled entries
    Uns32 numEmpty;     // number of empty entries
    Uns32 bits;         // the width of each entry in bits
} vmiConnInfo, *vmiConnInfoP;
```

Example

```
#include "vmi/vmiMessage.h"
#include "vmi/vmiRt.h"
#include "vmi/vmiTypes.h"

// processor structure definition
typedef struct cpuxS {
    vmiConnOutputP outputConn;      // output connection  (see section 2.3)
} cpux, *cpuxP;

// Print debug information about the processor input connection
static void vmic_OutputConnDebug(cpuxP cpux) {

    // fill a local structure with connection information
    vmiConnInfo info;
    vmirtConnGetOutputInfo(cpux->outputConn, vmiConnInfoP info);

    vmiPrintf("processor %s:\n", vmirtProcessorName((vmiProcessorP)cpux));
    vmiPrintf("  words     = %u\n", info.words);
    vmiPrintf("  numFilled = %u\n", info.numFilled);
    vmiPrintf("  numEmpty  = %u\n", info.numEmpty);
    vmiPrintf("  bits      = %u\n", info.bits);
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

12.5 vmirtConnGet

Prototype

```
Bool vmirtConnGet(vmiConnInputP conn, void *value, Bool peek);
```

Description

This function performs a read from a connection container object specified by `conn`. The value read is assigned to the address specified by `value`; the size of `value` (in bits) must have been specified previously when calling `vmirtConnGetInput`. If `peek` is `True`, the value will be removed from the container; otherwise, it will be copied from the container.

In the case that the input connection container is empty prior to the attempted read, the function returns `False` and `value` is unaffected; otherwise, if the read was successful, the function returns `True`.

This function may be used in combination with `vmirtConnNotifyPut` and `vmirtHalt` to implement a blocking read, as shown in the example below. Blocking reads may also be implemented more directly using the morph-time function `vmimtConnGetRB`; see the *VMI Morph Time Function Reference* for more details.

Example

```
#include "vmi/vmiRt.h"
#include "vmi/vmiTypes.h"

#define CPUX_GREGS 32               // number of GPRs
#define CPUX_GBITS 32               // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32         regs[CPUX_GREGS]; // 32-bit GPRs
    vmiConnInputP inputConn;        // input connection
} cpux, *cpuxP;

// See section 2.3 for example of constructing connections and initializing
// cpux->inputConn

// Callback activated when the current connection is updated
static VMI_CONN_UPDATE_FN(restartAfterBlock) {
    vmirtRestartNow(processor);
}

// Get from connection, block if data not available
static void vmic_ConnGetOrBlock(cpuxP cpux, Uns32 thisPC, Uns32 rd, Bool peek) {

    if(!vmirtConnGet(cpux->inputConn, &cpux->regs[rd], peek)) {

        // on restart, reexecute this instruction
        vmirtSetPC((vmiProcessorP)cpux, thisPC);

        // halt this processor after this instruction
        vmirtHalt((vmiProcessorP)cpux);

        // install notifier to be called on connection event
        vmirtConnNotifyPut(cpux->inputConn, restartAfterBlock);
    }
```

```
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

12.6 vmirtConnPut

Prototype

```
Bool vmirtConnPut(vmiConnOutputP conn, const void *value);
```

Description

This function performs a write to a connection container object specified by `conn`. The value to write is specified by `value`; the size of `value` (in bits) must have been specified previously when calling `vmirtConnGetOutput`.

In the case that the output connection container is full prior to the attempted write, the function returns `False` and the write is not performed; otherwise, if the write was successful, the function returns `True`.

This function may be used in combination with `vmirtConnNotifyGet` and `vmirtHalt` to implement a blocking write, as shown in the example below. Blocking writes may also be implemented more directly using the morph-time function `vmimtConnPutRB`; see the *VMI Morph Time Function Reference* for more details.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32               // number of GPRs
#define CPUX_GBITS 32               // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32          regs[CPUX_GREGS];// 32-bit GPRs
    vmiConnOutputP outputConn;      // output connection
} cpux, *cpuxP;

// See section 2.3 for example of constructing connections and initializing
// cpux->outputConn

// Callback activated when the current connection is updated
static VMI_CONN_UPDATE_FN(restartAfterBlock) {
    vmirtRestartNow(processor);
}

// Put to connection, block if full
static void vmic_ConnPutOrBlock(cpuxP cpux, Uns32 thisPC, Uns32 rb) {

    if(!vmirtConnPut(cpux->outputConn, &cpux->regs[rb])) {

        // on restart, reexecute this instruction
        vmirtSetPC((vmiProcessorP)cpux, thisPC);

        // halt this processor after this instruction
        vmirtHalt((vmiProcessorP)cpux);

        // install notifier to be called on connection event
        vmirtConnNotifyGet(cpux->outputConn, restartAfterBlock);
    }
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

12.7 vmirtConnNotifyGet

Prototype

```
Bool vmirtConnNotifyGet(vmiConnOutputP conn, vmiConnUpdateFn updateCB);
```

Description

This function registers an update function, `updateCB`, which is called the next time a value is removed from the output connection container (for example, by a call to `vmirtConnGet`). The callback is one-shot: that is, it is deregistered when it is called.

This function is required to support blocking writes with `vmirtConnPut`, as shown in the following example. The registered callback will typically call either `vmirtRestartNow` or `vmirtRestartNext` to restart a halted processor.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32               // number of GPRs
#define CPUX_GBITS 32               // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32          regs[CPUX_GREGS];// 32-bit GPRs
    vmiConnOutputP outputConn;      // output connection
} cpux, *cpuxP;

// See section 2.3 for example of constructing connections and initializing
// cpux->outputConn

// Callback activated when the current connection is updated
static VMI_CONN_UPDATE_FN(restartAfterBlock) {
    vmirtRestartNow(processor);
}

// Put to connection, block if full
static void vmic_ConnPutOrBlock(cpuxP cpux, Uns32 thisPC, Uns32 rb) {

    if(!vmirtConnPut(cpux->outputConn, &cpux->regs[rb])) {

        // on restart, reexecute this instruction
        vmirtSetPC((vmiProcessorP)cpux, thisPC);

        // halt this processor after this instruction
        vmirtHalt((vmiProcessorP)cpux);

        // install notifier to be called on connection event
        vmirtConnNotifyGet(cpux->outputConn, restartAfterBlock);
    }
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

12.8 vmirtConnNotifyPut

Prototype

```
Bool vmirtConnNotifyPut(vmiConnInputP conn, vmiConnUpdateFn updateCB);
```

Description

This function registers an update function, `updateCB`, which is called the next time a value is written to the input connection container (for example, by a call to `vmirtConnPut`). The callback is one-shot: that is, it is deregistered when it is called.

This function is required to support blocking reads with `vmirtConnGet`, as shown in the following example. The registered callback will typically call either `vmirtRestartNow` or `vmirtRestartNext` to restart a halted processor.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiTypes.h”

#define CPUX_GREGS 32               // number of GPRs
#define CPUX_GBITS 32               // size of GPR (bits)

// processor structure definition
typedef struct cpuxS {
    Uns32         regs[CPUX_GREGS]; // 32-bit GPRs
    vmiConnInputP inputConn;        // input connection
} cpux, *cpuxP;

// See section 2.3 for example of constructing connections and initializing
// cpux->inputConn

// Callback activated when the current connection is updated
static VMI_CONN_UPDATE_FN(restartAfterBlock) {
    vmirtRestartNow(processor);
}

// Get from connection, block if data not available
static void vmic_ConnGetOrBlock(cpuxP cpux, Uns32 thisPC, Uns32 rd, Bool peek) {

    if(!vmirtConnGet(cpux->inputConn, &cpux->regs[rd], peek)) {

        // on restart, reexecute this instruction
        vmirtSetPC((vmiProcessorP)cpux, thisPC);

        // halt this processor after this instruction
        vmirtHalt((vmiProcessorP)cpux);

        // install notifier to be called on connection event
        vmirtConnNotifyPut(cpux->inputConn, restartAfterBlock);
    }
}
```

Notes and Restrictions

See section 4.4.

QuantumLeap Semantics

Synchronizing

13 Simulation Environment Access

This section describes support functions that allow access to the simulation environment. For example, it is possible to obtain the values of command line flags, or Imperas Simulator (`imperas.exe`) platform attributes, or CpuManager / OVPsim user attributes.

13.1 vmirtGetCurrentProcessor

Prototype

```
vmiProcessorP vmirtGetCurrentProcessor(void);
```

Description

This routine returns the currently-executing processor. If it is called in a context where there is no currently-executing processor (for example, in a processor constructor or destructor) it returns `NULL`.

Example

```
#include “vmi/vmiRt.h”

static void vmic_TakeInterruptNext(void) {

    vmiProcessorP processor = vmirtGetCurrentProcessor();
    cpuxP         cpux      = (cpuxP)processor;

    if(exceptionPending(cpux)) {

        Addr thisPC = vmirtGetPC(processor);
        Addr nextPC = getNextPC(thisPC);

        vmirtSetPCFlushTarget(processor, nextPC);
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Thread safe

13.2 vmirtCPUId

Prototype

```
Uns32 vmirtCPUId(vmiProcessorP processor);
```

Description

This routine returns the *CPU id* of the passed processor. (For simulations using a CpuManager / OVPsim platform, the CPU id is specified using the `cpuId` parameter to `icmNewProcessor`; for Imperas Simulator (`imperas.exe`) simulations, the CPU id is specified in the platform file.)

This function is useful when simulating multiprocessor platforms where behavior differs between processors. For example, processors might jump to different handler addresses on interrupt.

Example

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

// routine called to take an exception, jumping to 'baseAddress + cpuId*256'
// the current program counter is saved in register CPUX_REG_EPCR
static void vmic_TakeException(cpuxP cpux, Uns32 baseAddress) {
    Uns32 simPC     = vmirtGetPC((vmiProcessorP)cpux);
    Uns32 cpuId     = vmirtCPUId((vmiProcessorP)cpux);
    Uns32 intVector = baseAddress + cpuId*256;
    cpuxEnterKernelMode(cpux);
    cpux->regs[CPUX_REG_EPCR] = simPC;
    vmirtSetPC((vmiProcessorP)cpux, intVector);
}

// handler for read out of bounds
VMI_READ_EXCEPT_FN(cpuxRdPrivExceptionCB) {
    takeException((cpuxP)processor, DPF_ADDRESS);
}

// handler for write out of bounds
VMI_WRITE_EXCEPT_FN(cpuxWrPrivExceptionCB) {
    takeException((cpuxP)processor, DPF_ADDRESS);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

13.3 vmirtGetProcessorForCPUId

Prototype

```
vmiProcessorP vmirtGetProcessorForCPUId(Uns32 cpuId);
```

Description

Given a *CPU id*, this function returns any processor in the platform with a matching id. If there is no matching processor, it returns `NULL`. (For simulations using a CpuManager / OVPsim platform, the CPU id is specified using the `cpuId` parameter to `icmNewProcessor`; for Imperas Simulator (`imperas.exe`) simulations, the CPU id is specified in the platform file.)

This function enables direct inter-processor introspection. An example use is a situation where registers of one processor are directly readable by another (perhaps by memory mapping). This is often the case in platforms where processors have been specifically designed to operate in groups.

Example

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

// return the value of register 'regId' from processor 'cpuId'
static Uns32 vmic_ReadRegister(Uns32 cpuId, Uns32 regId) {

    vmiProcessorP processor = vmirtGetProcessorForCPUId(cpuId);

    if(processor) {
        return ((cpuxP)processor)->regs[regId];
    } else {
        return 0;
    }
}
```

Notes and Restrictions

1. Introspection is only useful when the processor model is being used in a known platform context. For example, when querying the value of another processor's register, there is an implicit assumption that the processor returned by `vmirtGetCPUForId` is of a particular type (`cpux` in the above example).

QuantumLeap Semantics

Synchronizing

13.4 vmirtProcessorFlags

Prototype

```
Uns32 vmirtProcessorFlags(vmiProcessorP processor);
```

Description

Every processor in a simulation can be supplied with a model-specific set of flags as a 32-bit value. These flags can be freely used by the model, perhaps to enable special operating modes or enable debug output.

In a CpuManager / OVPsim simulation, these flags are specified by the `cpuFlags` argument to `icmNewProcessor`.

In the Imperas Simulator (`imperas.exe`), these flags can be supplied in two ways:

1. Using the per-processor `cpuflags` entry in the platform file;
2. Using the command line `--cpuflags` option (for platforms with only one processor type).

Example

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

// processor-specific configuration flags
#define ENABLE_VERBOSE(_P)   (vmirtProcessorFlags(_P) & 0x01)
#define ENABLE_TEST_MODE(_P) (vmirtProcessorFlags(_P) & 0x02)
#define ENABLE_BARE_BOOT(_P) (vmirtProcessorFlags(_P) & 0x04)
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

13.5 vmirtProcessorName

Prototype

```
const char *vmirtProcessorName(vmiProcessorP processor);
```

Description

This function returns the instance name of the passed processor, typically used for output message context. (For simulations using a CpuManager / OVPsim platform, the instance name is specified using the `name` parameter to `icmNewProcessor`; for Imperas Simulator (`imperas.exe`) simulations, the instance name is specified in the platform file.)

Example

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

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

    // Check for valid CPU license
    vmiMessage("I", "CPUX_INFO",
        "Constructor called for processor %s of type %s",
        vmirtProcessorName(processor),
        vmirtProcessorType(processor)
    }

    . . . etc . . .
}
```

Notes and Restrictions

1. When called for a processor that is a member of an SMP cluster that has not been explicitly named using `vmirtSetProcessorName`, the returned name is a constructed string using the format:
   `<baseName>_<index1>_<index2>_..._<indexN>`
   where `<baseName>` is the given name of the SMP root processor and `<index1>`, `<index2>`, … etc are the SMP indices of each processor level down the hierarchy.
2. If the name is constructed using SMP indices, it is valid only until the next call to `vmirtSetProcessorName`.
3. If the name is constructed using SMP indices, it will change if the processor is relocated in the SMP hierarchy by a call the `vmirtSetSMPParent`.

QuantumLeap Semantics

Non-self-synchronizing

13.6 vmirtSetProcessorName

Prototype

```
void vmirtSetProcessorName(vmiProcessorP processor, const char *name);
```

Description

This function sets the name of the processor. The new name will be shown in all trace output and verbose listings, and will be returned by `vmirtProcessorName`. This function should usually be called in the processor constructor.

Example

This example is taken from the OVP MIPS32 model. `vmirtSetProcessorName` is used in this model to specify the names of all CPU, VPE and TC objects in the SMP cluster. The generated names incorporate the parent name at leach level to ensure uniqueness in the platform:

```
static void setName(vmiProcessorP processor, const char *type) {

    vmiProcessorP parent = vmirtGetSMPParent(processor);

    if(parent) {

        // TCs appear to be children of CPU rather than VPE
        if(((mipsP)parent)->objectType == MOT_VPE) {
            parent = vmirtGetSMPParent(parent);
        }

        const char *baseName = vmirtProcessorName(parent);
        char        name[strlen(baseName)+10];
        char        index[10];

        sprintf(index, "%u", vmirtGetSMPIndex(processor));
        strcpy(name, baseName);
        strcat(name, "_");
        strcat(name, type);
        strcat(name, index);

        vmirtSetProcessorName(processor, name);
    }
}
```

Notes and Restrictions

1. For Imperas tool compatibility, choose simple alphanumeric processor names. Names with special characters may prevent the use of Imperas tools.

QuantumLeap Semantics

Non-self-synchronizing

13.7 vmirtProcessorType

Prototype

```
const char *vmirtProcessorType(vmiProcessorP processor);
```

Description

This function returns the instance type of the passed processor. For simulations using a CpuManager / OVPsim platform, the instance type is specified using the `type` parameter to `icmNewProcessor`. For Imperas Simulator (`imperas.exe`) simulations, the instance type is specified in the processor definition file.

Example

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

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

    // Check for valid CPU license
    vmiMessage("I", "CPUX_INFO",
        "Constructor called for processor %s of type %s",
        vmirtProcessorName(processor),
        vmirtProcessorType(processor)
    }

    . . . etc . . .
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

13.8 vmirtProcessorStringAttribute

Prototype

```
const char *vmirtProcessorStringAttribute(
    vmiProcessorP processor,
    const char   *name
);
```

Description

This routine enables any string-valued platform attribute to be accessed within an intercept library. If the named attribute exists and is string-valued, the function returns the value; otherwise, it returns `NULL`.

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to check for a user specified attribute first on the extension library, then on the processor and use a default value if neither are found:

Example

```
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiRt.h"

//
// Constructor
//
static VMIOS_CONSTRUCTOR_FN(constructor) {

    // get parameters for intercept library (passed to constructor)
    paramValuesP params = parameterValues;
    const char  *regName;

    if((regName=params->resultReg)){

        // Use value specified on extension library
        strcpy(object->regName, regName);

    } else if((regName=vmirtProcessorStringAttribute(processor,"resultReg"))){

        // Use value specified on processor
        strcpy(object->regName, regName);

    } else {

        // Use default value
        strcpy(object->regName, regNameDefault);
    }
}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3
2. In VMI versions prior to 6.8.0, this function was called `vmirtPlatformStringAttribute`. This name is deprecated.

QuantumLeap Semantics
Non-self-synchronizing

13.9 vmirtProcessorBoolAttribute

Prototype

```
Bool vmirtProcessorBoolAttribute(
    vmiProcessorP processor,
    const char   *name,
    Bool         *found
);
```

Description

This routine enables any `Bool`-valued processor attribute to be accessed. If the named attribute exists and is of `Bool` type, the attribute value is returned; otherwise, the function generates a parameter error and returns `0`. By-ref parameter `found` is set to indicate whether the parameter was explicitly specified (`True`) or whether the returned value is the default (`False`).

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to get an enable flag for the processor to which the library is attached.

Example

```
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiRt.h"

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiosObjectP kobject = object;

    //
    // Read Attributes
    //
    Bool found;
    Bool val;

    //
    // Are we invoking strace ?
    //
    object->strace = False;
    val = vmirtProcessorBoolAttribute(processor, "strace", &found);
    if (found && val) {
        object->strace = True;
    }

    ...

}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3.

QuantumLeap Semantics

Non-self-synchronizing

13.10 vmirtProcessorUns32Attribute

Prototype

```
Uns32 vmirtProcessorUns32Attribute(
    vmiProcessorP processor,
    const char   *name,
    Bool         *found
);
```

Description

This routine enables any `Uns32`-valued processor attribute to be accessed. If the named attribute exists and is of `Uns32` type, the attribute value is returned; otherwise, the function generates a parameter error and returns `0`. By-ref parameter `found` is set to indicate whether the parameter was explicitly specified (`True`) or whether the returned value is the default (`False`).

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to get CPU number property defined for the processor to which the library is attached.

Example

```
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiRt.h"

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiosObjectP kobject = object;

    //
    // Read Attributes
    //
    Bool found;
    Uns32 val;

    object->numcpu = NUMCPU;
    val = vmirtProcessorUns32Attribute(processor, "numcpu", &found);
    if (found) {
        object->numcpu = val;
    }

    ...

}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3.

QuantumLeap Semantics

Non-self-synchronizing

13.11 vmirtProcessorUns64Attribute

Prototype

```
Uns64 vmirtProcessorUns64Attribute(
    vmiProcessorP processor,
    const char   *name,
    Bool         *found
);
```

Description

This routine enables any `Uns64`-valued processor attribute to be accessed. If the named attribute exists and is of `Uns64` type, the attribute value is returned; otherwise, the function generates a parameter error and returns `0`. By-ref parameter `found` is set to indicate whether the parameter was explicitly specified (`True`) or whether the returned value is the default (`False`).

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to get a page size defined for the processor to which the library is attached.

Example

```
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiRt.h"

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiosObjectP kobject = object;

    //
    // Read Attributes
    //
    Bool found;
    Uns64 val;

    //
    // Default location for user program top of stack at initialization
    //
    object->pagesize = PAGE_SIZE;
    val = vmirtProcessorUns64Attribute(processor, "pagesize", &found);
    if (found) {
        object->pagesize = val;
    }

    ...

}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3
2. In VMI versions prior to 6.8.0, this function was called `vmirtPlatformUns64Attribute`. This name is deprecated.

QuantumLeap Semantics
Non-self-synchronizing

13.12 vmirtProcessorFlt64Attribute

Prototype

```
Flt64 vmirtProcessorFlt64Attribute(
    vmiProcessorP processor,
    const char   *name,
    Bool         *found
);
```

Description

This routine enables any `Flt64`-valued processor attribute to be accessed. If the named attribute exists and is of `Flt64` type, the attribute value is returned; otherwise, the function generates a parameter error and returns `0.0`. By-ref parameter `found` is set to indicate whether the parameter was explicitly specified (`True`) or whether the returned value is the default (`False`).

This function should normally be used only in intercept libraries. In processor models, configuration parameters should be used instead, as described in section 3.

The example below shows how this might be used in an intercept library constructor to get the nominal processor MIPS rate defined for the processor to which the library is attached.

Example

```
#include "vmi/vmiOSAttrs.h"
#include "vmi/vmiRt.h"

//
// Library Constructor Function
//
static VMIOS_CONSTRUCTOR_FN(libraryConstructor)) {

...
    // get processor mips rate
    Bool found;
    Flt64 mipsInMhz =
        vmirtProcessorFlt64Attribute(processor, "mips", &found);
    if (!found) {
        mipsInMhz = DEFAULT_MIPS;
    }
...

}
```

Notes and Restrictions

1. Use of this function in processor models is deprecated in favor of the model configuration parameters described in section 3

QuantumLeap Semantics

Non-self-synchronizing

14 SMP Processor Functions

The VMI interfaces allow *Symmetric MultiProcessing* (SMP) processors to be modeled. This section describes the concepts involved and the functions available to support SMP processors. SMP function support has revised and improved from version 2.0.20 of the API.

14.1 SMP Processor Hierarchies

SMP processor hierarchies consist of an arbitrary number of levels of *container* processor objects that contain, at the lowest level, *leaf* processor objects. Only the leaf level objects are simulated – container objects are *not* simulated, but may contain registers that are shared by descendant leaf objects that *are* simulated. Leaf level processors may get their parents or higher level ancestors using `vmirtGetSMPParent` to access these shared registers. Leaf level objects may be either *self-contained* or *part of a simulation group*.

Self-contained objects are treated by the simulator exactly as if they had been instantiated as separate processors: each has its own instruction count and will run for a complete quantum when simulated. This level of abstraction is suitable for modeling processors with multiple independent cores.

Simulation group members are treated by the simulator as a time-sliced group: when simulated for a quantum, the quantum time will be divided into a number of sub-quanta, and each of the group members will be run for a sub-quantum before moving on to the next. If any group member is halted, the unused simulation time will be used instead by other running members of the group. This level of abstraction is suitable for modeling processors that use microthreading to share some common resources between several thread contexts.

SMP processor hierarchies are defined using the function `smpContext` argument to the processor model constructor. Other functions are available to traverse the SMP hierarchy at run time and to dynamically modify the SMP hierarchy and simulation group schedule order.

14.2 Specifying SMP Attributes in the Processor Constructor

The processor constructor has the following prototype:

```
#define VMI_CONSTRUCTOR_FN(_NAME) void _NAME( \
    vmiProcessorP  processor,           \
    const char    *type,                \
    Bool           simulateExceptions,  \
    vmiSMPContextP smpContext           \
)
```

The `smpContext` argument is used to construct an SMP processor hierarchy beneath the root level processor object. The type `vmiSMPContext` is defined in `vmiTypes.h` as follows:

```
typedef struct vmiSMPContextS {
    Bool  isContainer;  // whether this processor is a container
    Uns32 numChildren;  // if a container, the initial number of children
    Uns32 numSubSlices; // if a container, the number of subslices
    Uns32 index;        // index number in schedule list
} vmiSMPContext, *vmiSMPContextP;
```

Fields in this structure should be updated by the constructor to specify the details of the SMP hierarchy, as follows:

1. If this processor is to be a container object with children beneath it, set the `isContainer` field to `True`. For leaf-level objects, `isContainer` should be `False` (the default value).
2. The initial number of children of the container should be set using the `numChildren` field. It is valid for a container to have no children initially, in which case `numChildren` should be zero (the default). If this is a container object, then on completion of the constructor `numChildren` processor objects will be allocated and attached as children of this processor, with indices `0`, `1`, … `numChildren -1`. The constructor will then be called again for each child processor, enabling further levels to be added to the hierarchy if required beneath each child.
3. The `numSubSlices` field only has effect for a parent of leaf-level processors. If `numSubSlices` is non-zero, then children of this processor comprise a *simulation group*: each quantum for which the processor is simulated will be split into `numSubSlices` sub-quanta, and the children will share these sub-quanta when simulated, as described in the introduction to this section. If `numSubSlices` is zero, then children of this processor will be considered as independent cores when simulated.
4. The `index` field specifies the initial SMP index number for this processor. This index number is returned by `vmirtGetSMPIndex`, and used to construct the processor name if no name has been explicitly specified using `vmirtSetProcessorName` (see `vmirtProcessorName` for a description of the default name syntax). If the processor is a member of a simulation group, the index determines the order in which members of that group are simulated (smallest indices are simulated first). It is legal for two members of a simulation

group to have the same index; in this case, the order in which they are simulated is indeterminate. By default, index numbers increase from zero for each child of a processor.

The `vmiSMPContext` field options are summarized in the following table:

Level	isContainer	numChildren	numSubSlices	Meaning
leaf	False	(ignored)	(ignored)	leaf level processor in SMP hierarchy (default)
parent of container	True	N	(ignored)	container processor with N children which are themselves containers
parent of leaf	True	N	0	container processor with N independently-scheduled children
parent of leaf	True	N	M {!=0)	container processor with N children in a simulation group with M sub-quanta

Example

This example is taken from the standard OVP MIPS processor model. In general, MIPS processors can have up to four hierarchy levels:

1. The *CMP* level – typically, four processors in a cluster (for example, a 1004K);
2. The *core* level – an individual core in a 1004K, or the root level in non-CMP processors (e.g. 34K);
3. The *VPE* level – a *virtual processing element* within a processor such as the 34K;
4. The *TC* level – a *thread context* within a VPE.

The TCs within a VPE are an SMP group that share timers and other hardware resources, but have separate GPRs.

The constructor for the MIPS processor uses the smpContext in four ways:

1. At the CMP level, the processor is specified to be a container for *cores* in the CMP;
2. At the core level, the processor is specified to be a container for *VPEs* in the core;
3. At the VPE level, the processor is specified to be a container for *TCs* in the VPE. It is also specified that the number of sub-slices should be the number of TCs in the CPU (so that when all TCs are running on one VPE, each will execute for one sub-quantum per simulated quantum).
4. At the TC level, index numbers are adjusted so that they increase uniquely across all TCs on the CPU. For example, VPE0 may initially own TCs 0, 1 and 2, and VPE1 may initially own TCs 4 and 5.

The `mips` processor object has an `objectType` field which is used to identify the type of object for which the constructor is being called.

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

VMI_CONSTRUCTOR_FN(mipsConstructor) {

    mipsP        mips         = (mipsP)processor;
    mipsP        parent       = (mipsP)vmirtGetSMPParent(processor);
    vmiViewObjectP baseObject = vmirtGetProcessorViewObject(processor);
    mipsConfigCP config       = allocConfig(mips, parent);
```

```
    Bool          isMT       = config->Config3.MT;
    Uns32         CPUNum     = config->GCR_CONFIG.PCORES+1;

    . . .

    // determine the object type
    if(parent) {
        mips->objectType = isMT ? parent->objectType-1 : MOT_TC;
    } else if(CPUNum>1) {
        mips->objectType = MOT_CMP;
    } else if(isMT) {
        mips->objectType = MOT_CPU;
    } else {
        mips->objectType = MOT_TC;
    }

    . . .

    switch(mips->objectType) {

        case MOT_CMP: {

            // multi-processor cluster with 'CPUNum' children

            // supply SMP configuration properties
            smpContext->isContainer = True;
            smpContext->numChildren = CPUNum;

            // do CMP level initialization
            initializeCMP(processor);
            setName(processor, "CMP");

            break;
        }

        case MOT_CPU: {

            // CPU-level object with 'VPENum' children, scheduled independently
            Uns32 VPENum = mips->config->MVPConf0.PVPE+1;

            // supply SMP configuration properties
            smpContext->isContainer = True;
            smpContext->numChildren = VPENum;

            // do CPU level initialization
            initializeCore(processor);
            setName(processor, "CPU");

            break;
        }

        case MOT_VPE: {

            // VPE-level object with 'TCNum' children, scheduled together
            Uns32 TCNum     = mips->config->MVPConf0.PTC+1;
            Uns32 VPE0TCNum = mips->config->VPE0MaxTC+1;
            Uns32 VPETCNum;

            // limit VPE0TCNum to the available number of TCs
            if(VPE0TCNum>TCNum) {
                VPE0TCNum = TCNum;
            }

            // number of children depends on this VPE index
            if(smpContext->index==0) {
                VPETCNum = VPE0TCNum;
            } else if(smpContext->index==1) {
                VPETCNum = TCNum-VPE0TCNum;
            } else {
                VPETCNum = 0;
            }
```

```
            // supply SMP configuration properties
            smpContext->isContainer  = True;
            smpContext->numChildren  = VPETCNum;
            smpContext->numSubSlices = TCNum;

            // do VPE level initialization
            initializeVPE(processor);
            setName(processor, "VPE");

            break;
        }

        default: {

            // TC-level object, scheduled as a group member
            if(isMT) {

                // get parent VPE index
                Uns32 VPE0TCNum = mips->config->VPE0MaxTC+1;
                Uns32 vpeIndex  = vmirtGetSMPIndex((vmiProcessorP)parent);

                // correct indices for TCs on VPE 1
                if(vpeIndex==1) {
                    smpContext->index += VPE0TCNum;
                }

            } else {

                // do CPU/VPE register initialization
                initializeCore(processor);
                initializeVPE(processor);
            }

            // do TC level initialization
            initializeTC(processor, smpContext->index);
            setName(processor, "TC");

            break;
        }
}
```

14.3 vmirtGetSMPParent

Prototype

```
vmiProcessorP vmirtGetSMPParent(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the parent processor object in the hierarchy. If the processor is not a member of an SMP hierarchy, or is at the root level, `NULL` is returned.

Example

```
#include “vmi/vmiRt.h”

VMI_CONSTRUCTOR_FN(mipsConstructor) {

    mipsP          mips       = (mipsP)processor;
    mipsP          parent     = (mipsP)vmirtGetSMPParent(processor);
    vmiViewObjectP baseObject = vmirtGetProcessorViewObject(processor);
    mipsConfigCP   config     = allocConfig(mips, parent);
    Bool           isMT       = config->Config3.MT;
    Uns32          CPUNum     = config->GCR_CONFIG.PCORES+1;

    . . .
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

14.4 vmirtSetSMPParent

Prototype

```
void vmirtSetSMPParent(vmiProcessorP processor, vmiProcessorP parent);
```

Description

This function relocates an SMP processor within the hierarchy so that its new parent is `parent`. It may be used to relocate both leaf level and container processors.

Example

This example is taken from the OVP MIPS model.

```
#include “vmi/vmiRt.h”

void mipsWriteTCBind(mipsP cxt, mipsP tc, Uns32 newValue) {

    mipsP oldVPE   = VPE_FOR_TC(tc);
    mipsP cpu      = CPU_FOR_VPE(oldVPE);
    Uns32 oldValue = COP0_REG(tc, TCBind);
    mipsP newVPE;

    // get the old value of the CurVPE field
    Uns8 oldCurVPE = COP0_FIELD(tc, TCBind, CurVPE);

    // update the field, preserving read-only bits
    COP0_REG(tc, TCBind) = (
        (oldValue & ~C0_MASK_TCBind) |
        (newValue &  C0_MASK_TCBind)
    );

    // get the new value of the CurVPE field
    Uns8 newCurVPE = COP0_FIELD(tc, TCBind, CurVPE);

    if(newCurVPE==oldCurVPE) {

        // no action if field CurVPE is unchanged

    } else if(!COP0_FIELD(cpu, MVPControl, VPC)) {

        // TCBind/CurVPE not writable unless MVPControl/VPC is clear
        COP0_FIELD(tc, TCBind, CurVPE) = oldCurVPE;

    } else if((newVPE=findVPE(cpu, newCurVPE))) {

        // reassign to the new parent
        vmirtSetSMPParent((vmiProcessorP)tc, (vmiProcessorP)newVPE);
        tc->context = newVPE;

        . . . .
    }
}
```

Notes and Restrictions

1. The new parent and the original parent must lie beneath the same root level container processor.
2. The new parent and the original parent must lie at the same depth in the tree beneath the root level container (but they do not have to be siblings).

QuantumLeap Semantics
Synchronizing

14.5 vmirtGetSMPChild

Prototype

```
vmiProcessorP vmirtGetSMPChild(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the first child processor object in the hierarchy. If the processor is not a member of an SMP hierarchy, or has no children, `NULL` is returned.

Example

```
#include “vmi/vmiRt.h”

static vmiProcessorP getIndexedChild(vmiProcessorP parent, Uns32 index) {

    vmiProcessorP this;

    for(this=vmirtGetSMPChild(parent); this; this=vmirtGetSMPNextSibling(this)) {
        if(vmirtGetSMPIndex(this)==index) {
            return this;
        }
    }

    return (vmiProcessorP)0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

14.6 vmirtGetSMPPrevSibling

Prototype

```
vmiProcessorP vmirtGetSMPPrevSibling(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the previous sibling. If the processor is not a member of an SMP hierarchy, or has no previous sibling, `NULL` is returned.

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

14.7 vmirtGetSMPNextSibling

Prototype

```
vmiProcessorP vmirtGetSMPNextSibling(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the previous sibling. If the processor is not a member of an SMP hierarchy, or has no previous sibling, `NULL` is returned.

Example

```
#include “vmi/vmiRt.h”

static vmiProcessorP getIndexedChild(vmiProcessorP parent, Uns32 index) {

    vmiProcessorP this;

    for(this=vmirtGetSMPChild(parent); this; this=vmirtGetSMPNextSibling(this)) {
        if(vmirtGetSMPIndex(this)==index) {
            return this;
        }
    }

    return (vmiProcessorP)0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

14.8 vmirtGetSMPActiveSibling

Prototype

```
vmiProcessorP vmirtGetSMPActiveSibling(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the *active* sibling. If the processor is not a member of an SMP hierarchy, the processor itself is returned

In every SMP hierarchy, there is only one leaf level processor in each SMP group that is active at any time. When the simulator switches from simulating one member of the SMP group to the next, the previously-running member becomes inactive and the new member becomes active.

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

14.9 vmirtGetSMPIndex

Prototype

```
Uns32 vmirtGetSMPIndex(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the SMP index number associated with that object. If the processor is not a member of an SMP hierarchy zero is returned.

By default, siblings are in numeric index order, with the first having an index of zero and the last an index of `coreNum-1`. This default behavior can be modified in the processor constructor or by `vmirtSetSMPIndex`.

Example

```
#include “vmi/vmiRt.h”

static vmiProcessorP getIndexedChild(vmiProcessorP parent, Uns32 index) {

    vmiProcessorP this;

    for(this=vmirtGetSMPChild(parent); this; this=vmirtGetSMPNextSibling(this)) {
        if(vmirtGetSMPIndex(this)==index) {
            return this;
        }
    }

    return (vmiProcessorP)0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

14.10 vmirtSetSMPIndex

Prototype

```
void vmirtSetSMPIndex(vmiProcessorP processor, Uns32 index);
```

Description

Given a processor object at any level in an SMP hierarchy, this sets the SMP index number associated with that object.

If processors are members of an SMP group, they are simulated in increasing index order. This function can therefore be used to affect the scheduling order of processors in an SMP group.

Notes and Restrictions

1. If SMP members are not explicitly named by `vmirtSetProcessorName`, the SMP index is used to construct the processor name returned by `vmirtProcesorName`, and used in trace output and reporting. Setting the index will have the effect of renaming the processor in this case.

QuantumLeap Semantics

Non-self-synchronizing

14.11 vmirtGetSMPCpuType

Prototype

```
vmiSMPCpuType vmirtGetSMPCpuType(vmiProcessorP processor);
```

Description

Given a processor object at any level in an SMP hierarchy, this returns the `vmiSMPCpuType` enum describing that object.

The `vmiSMPCpuType` enum is defined as:

```
typedef enum vmiSMPCpuTypeE {
    SMP_TYPE_CONTAINER = 0x0,   // Container Processor
    SMP_TYPE_LEAF      = 0x1,   // Leaf processor
    SMP_TYPE_GROUP     = 0x2,   // Part of an SMP simulation group

    // simple leaf processor (not a member of an SMP simulation group)
    SMP_SIMPLE_LEAF = SMP_TYPE_LEAF,

    // leaf processor that is a member of an SMP simulation group
    SMP_GROUP_LEAF  = SMP_TYPE_LEAF     | SMP_TYPE_GROUP,

    // simple container processor
    SMP_CONTAINER   = SMP_TYPE_CONTAINER,

    // container processor that contains a group of SMP_GROUP_LEAF processors
    SMP_GROUP       = SMP_TYPE_CONTAINER | SMP_TYPE_GROUP,
} vmiSMPCpuType;
```

This function may be used to determine if a processor is a leaf or container, and if it is a member of a simulation group.

Additional macros are defined in `vmiTypes.h` to help check specific characteristics:

```
// Is a processor of vmiSMPCpuType _T a leaf processor
#define SMP_IS_LEAF(_T) ((_T) & SMP_TYPE_LEAF)

// Is a processor of vmiSMPCpuType _T an SMP group processor
#define SMP_IS_GROUP(_T) ((_T) & SMP_TYPE_GROUP)
```

Example

The following example will report the type for each processor in an SMP hierarchy:

```
static vmiProcessorP getRoot(vmiProcessorP processor) {

    vmiProcessorP nextParent;
    while ((nextParent=vmirtGetSMPParent(processor)) != NULL) {
        processor = nextParent;
    }

    return processor;
}

static VMI_SMP_ITER_FN(reportType) {

    vmiSMPCpuType type = vmirtGetSMPCpuType(processor);
    vmiPrintf("SMP Type: \tProcessor %16s: type: %s%s\n",
```

```
            vmirtProcessorName(processor),
            SMP_IS_LEAF(type)  ? "Leaf " : "",
            SMP_IS_GROUP(type) ? "Group" : "");
}

reportSMPTypes(vmiProcessorP processor) {

    vmiProcessorP root = getRoot(processor);

    vmiPrintf("SMP Type: SMP Types for %s (root of %s):\n",
              vmirtProcessorName(root), vmirtProcessorName(processor));

    reportType(root, NULL);

    vmirtIterAllDescendants(root, reportType, NULL);
}
```

Notes and Restrictions

1. If the processor is not a member of an SMP hierarchy then the type SMP_SIMPLE_LEAF is returned.

QuantumLeap Semantics

Non-self-synchronizing

14.12 vmirtIterAllChildren

Prototype

```
void vmirtIterAllChildren(
    vmiProcessorP processor,
    vmiSMPIterFn  iterCB,
    void         *userData
);
```

Description

Given a processor object at any level in an SMP hierarchy, this iterates over the children of the processor, calling the passed `iterCB` for each one. The `iterCB` is passed the child processor pointer and a `userData` pointer. If the processor is not an SMP parent, the function has no effect.

The iterator callback should be defined using the `VMI_SMP_ITER_FN` macro.

Example

This example shows how this function might be used in a processor destructor do delete data structures associated with children of the root object:

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

static VMI_SMP_ITER_FN(destructorCB) {
    cpuxP cpux = (cpuxP)processor;
    deleteLeafStructures(cpux);
}

VMI_DESTRUCTOR_FN(mipsDestructor) {
    vmirtIterAllChildren(processor, destructorCB, 0);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

14.13 vmirtIterAllDescendants

Prototype

```
void vmirtIterAllDescendants(
    vmiProcessorP processor,
    vmiSMPIterFn  iterCB,
    void         *userData
);
```

Description

Given a processor object at any level in an SMP hierarchy, this iterates over all descendants of the processor, calling the passed `iterCB` for each one. The `iterCB` is passed the descendant processor pointer and a `userData` pointer. If the processor is not an SMP parent, the function has no effect.

The iterator callback should be defined using the `VMI_SMP_ITER_FN` macro.

Example

This example shows how this function is used in the standard OVP MIPS model destructor:

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

static VMI_SMP_ITER_FN(destructorCB) {

    mipsP mips = (mipsP)processor;

    // free UDI data structures
    mipsFreeUDI(mips);

    // free TLB structures
    if(mips->tlb) {
        mipsFreeTLB(mips);
    }

    // free debugger interface structures
    if(IS_TC(mips)) {
        mipsFreeRegInfo(mips);
    }

    // free shadow registers
    mipsFreeShadowRegisters(mips);

    // free CMP structures
    mipsCMPFree(mips);

    // free configuration
    mipsFreeModelConfig(mips);
}

VMI_DESTRUCTOR_FN(mipsDestructor) {

    // apply destructor to root processor
    destructorCB(processor, 0);

    // apply destructor to all descendants
    vmirtIterAllDescendants(processor, destructorCB, 0);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Non-self-synchronizing

14.14 vmirtIterAllProcessors

Prototype

```
void vmirtIterAllProcessors(
    vmiProcessorP processor,
    vmiSMPIterFn  iterCB,
    void         *userData
);
```

Description

Given a processor object at any level in an SMP hierarchy, this calls the passed `iterCB` for that processor and then iterates over all descendants of the processor, calling the passed `iterCB` for each one. The `iterCB` is passed each pointer and a `userData` pointer. If the processor is not an SMP parent, `iterCB` is called for the given processor only..

The iterator callback should be defined using the `VMI_SMP_ITER_FN` macro.

Example

This example shows how this function could have been used in the standard OVP MIPS model destructor to avoid an explicit call to the destructor for the root level processor:

```
#include “vmi/vmiRt.h”

static VMI_SMP_ITER_FN(destructorCB) {

    mipsP mips = (mipsP)processor;

    // free UDI data structures
    mipsFreeUDI(mips);

    // free TLB structures
    if(mips->tlb) {
        mipsFreeTLB(mips);
    }

    // free debugger interface structures
    if(IS_TC(mips)) {
        mipsFreeRegInfo(mips);
    }

    // free shadow registers
    mipsFreeShadowRegisters(mips);

    // free CMP structures
    mipsCMPFree(mips);

    // free configuration
    mipsFreeModelConfig(mips);
}

VMI_DESTRUCTOR_FN(mipsDestructor) {

    // apply destructor to all descendants
    vmirtIterAllProcessors(processor, destructorCB, 0);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics
Non-self-synchronizing

15 Communication Between Objects

This section describes functions that enable communication between distinct objects in a simulation. The interface is typically used by *intercept objects*.

Intercept objects get all their services from the simulator. Calling functions in another intercept object, although possible, is undesirable because it introduces a requirement to load one object before the other and prevents the objects from being used independently; they would be equally useful if linked into one object. The *shared data* interface allows objects to share data and to call functions through a registry which maps symbolic names (*keys*) to shared entries.

Keys are C strings; they can contain any valid string.
Keys are global and persist until `icmTerminate` is called or the simulator exits.

Version is a constant string which should be compiled into the two (or more) communicating intercept objects. It is the responsibility of the programmer to change the version if the protocol changes so that the programs are no longer compatible. The simulator will abort if two intercept objects use the same key with incompatible versions.

15.1 vmirtFindAddSharedData

Prototype

```
vmiSharedDataHandleP vmirtFindAddSharedData(
     const char *version,
     const char *key,
     void       *value
);
```

Description

This function finds a shared entry by its key, and returns its handle or, if the entry does not exist, creates it. This function can be used by multiple objects, regardless of the order in which they are constructed.

The value field in the entry can be given an initial value, but it should be noted that another user of this entry could also supply an initial value, in which case the last call will win. A more useful method of sharing data is it use `vmirtSetSharedDataValue()` and `vmirtGetSharedDataValue()` when communication has been established.

This function is typically used in an object's constructor, but can be used at any time during a simulation run.

Example

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

vmiSharedDataHandleP handle = vmirtFindAddSharedData(
    "1.0.0", "myApplicationKey", 0
);
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

15.2 vmirtFindSharedData

Prototype

```
vmiSharedDataHandleP vmirtFindSharedData(
     const char *version,
     const char *key
);
```

Description

This function finds a shared entry by its key, and returns its handle.

Example

```
#include “vmi/vmiRt.h”

vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

15.3 vmirtGetSharedDataValue

Prototype

```
void *vmirtGetSharedDataValue(vmiSharedDataHandleP handle);
```

Description

This function returns the client-specific `data` from a shared entry.

Example

```
#include “vmi/vmiRt.h”

vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");

if(handle) {
    myType p = vmirtGetSharedDataValue(handle);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

15.4 vmirtSetSharedDataValue

Prototype

```
void vmirtSetSharedDataValue(vmiSharedDataHandleP handle, void *value);
```

Description

This function sets the client-specific `data` in a shared entry.

Example

```
#include “vmi/vmiRt.h”

vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");

if(handle) {
    myType p = ....
    vmirtSetSharedDataValue(handle, p);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

15.5 vmirtRemoveSharedData

Prototype

```
void vmirtRemoveSharedData(vmiSharedDataHandleP handle);
```

Description

This function removes a shared entry and all its listeners.

Example

```
#include “vmi/vmiRt.h”
...
    vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");
...
    vmirtRemoveSharedData(handle);
...
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

15.6 vmirtWriteListeners

Prototype

```
Int32 vmirtWriteListeners(
    vmiSharedDataHandleP handle,
    Int32                id,
    void                *userData
);
```

Description

This function writes to all listeners registered to this entry (the listeners will usually be in another intercept object). The integer return status is set to zero then passed by address to each listener (who might modify it) before being returned by this function.

Example

```
#include “vmi/vmiRt.h”
#include “myShared.h"

VMIOS_CONSTRUCTOR(constructor) {
  ...
  vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0",  "myApplicationKey" );
  ...
}

...
  myStruct str = { ..... } ;
  // later on, during simulation
  int r = vmirtWriteListeners(handle, MY_CODE, &str);
...
```

Notes and Restrictions

1. Many listeners can be installed on one handle, allowing multiple objects to be notified by one or more shared objects.
2. It is advisable to share the definitions of `MY_CODE` and `myStruct` with other objects that communicate with this one.
3. Multiple objects can write to one shared data entry.

QuantumLeap Semantics

Synchronizing

15.7 vmirtRegisterListener

Prototype

```
void vmirtRegisterListener(
    vmiSharedDataHandleP      handle,
    vmirtSharedDataListenerFn listenerCB,
    void                     *object
);
```

Description

This function registers a *listener function* to be called when `vmirtWriteListeners` is called (by another object). The caller uses the `object` argument to specify a generic object that is passed when the listener is activated by a call to `vmirtWriteListeners`. When used in an intercept library context, this will typically be the `vmiosObjectP` of the current intercept object.

Type `vmirtSharedDataListenerFn` is defined in `vmiTypes.h` as follows:

```
#define VMI_SHARED_DATA_ISTENER_FN(_NAME) void _NAME( \
    void  *userObject,  \
    Int32 *ret,         \
    Int32  id,          \
    void  *userData     \
)
typedef VMI_SHARED_DATA_LISTENER_FN((*vmirtSharedDataListenerFn));
```

The arguments are as follows:
`userObject`: the pointer passed as the `object` argument to `vmirtRegisterListener`.
`ret`: a pointer to the return status. This integer is set to zero by the simulator. Its address is passed to each listener in turn (who might choose to update it).
`id`: an integer passed to `vmirtWriteListeners`, typically used to specify a required service or action.
`userData`: the pointer passed as the `userData` argument to `vmirtWriteListeners`.

Function `vmirtRegisterListener` can be called multiply times for the same shared data object, usually in the context of different intercept libraries. When the listeners are activated by `vmirtWriteListeners`, each registered listener will be notified in turn.

Example

```
#include “vmi/vmiRt.h”
#include “myShared.h"

VMI_SHARED_DATA_LISTENER(myListener) {
    switch (id) {
        case MY_CODE: {
             myStruct *strP = userData;
             strP->member = ....
        }
    }

    // change the return status.
    (*ret)++;
}
```

```
VMIOS_CONSTRUCTOR(constructor) {
    vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");
    vmirtRegisterListener(handle, myListener, object);
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

15.8 vmirtUnregisterListener

Prototype

```
Bool vmirtUnregisterListener(
    vmiSharedDataHandleP      handle,
    vmirtSharedDataListenerFn listenerCB,
    void                     *object
);
```

Description

This function removes a listener function with matching callback and object from a shared data handle. The function return value indicates whether a matching listener was found and removed.

Example

```
#include “vmi/vmiRt.h”
...
VMIOS_DESTRUCTOR(destructor) {
    vmiSharedDataHandleP handle = vmirtFindSharedData("1.0.0", "myApplicationKey");
    vmirtUnregisterListener(handle, myListener, object);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

16 Address Range Hash Utilities

Modeling TLBs and caches often requires the ability to model structures with address ranges in an efficient manner. For example, a TLB may contain many entries mapping pages of different sizes, and an efficient model requires the ability to map from a virtual address to the corresponding TLB entry very quickly.

To facilitate implementation of models containing TLBs and caches, the VMI Run Time Function API implements a *range hash* object, documented in this section.

16.1 vmirtNewRangeTable

Prototype

```
void vmirtNewRangeTable(vmiRangeTablePP table);
```

Description

Create a new *range table* object, used to hold range hash entries.

Example

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

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

// initialize range hash table implementing TLB
static void newTLB(cpuxP cpux) {
    vmirtNewRangeTable(&cpux->tlb);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

16.2 vmirtFreeRangeTable

Prototype
```
void vmirtFreeRangeTable(vmiRangeTablePP table);
```

Description
Free a previously-allocated *range table* object and all its entries.

Example
```
#include "vmi/vmiRt.h"
#include "vmi/vmiMessage.h"

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

// free range hash table implementing TLB
static void freeTLB(cpuxP cpux) {
    vmirtFreeRangeTable(&cpux->tlb);
}
```

Notes and Restrictions
None.

QuantumLeap Semantics
Synchronizing

16.3 vmirtInsertRangeEntry

Prototype

```
vmiRangeEntryP vmirtInsertRangeEntry(
    vmiRangeTablePP table,
    Addr            low,
    Addr            high,
    Uns64           userData
);
```

Description

Create and return a new range table entry spanning the range `low:high` in the passed range table. The `userData` argument can hold any application-specific data required for the entry: for example, it might typically be a physical address, or a structure containing access permissions.

In VMI versions prior to 6.2.0, ranges were not permitted to overlap. From VMI version 6.2.0, range tables may hold multiple overlapping ranges without restriction.

Example

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

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

typedef struct entryInfoS {
    Uns32   pa;    // physical address
    memPriv ar;    // access permissions
} entryInfo, *entryInfoP;

// insert new TLB entry
static void createTLBEntry(cpuxP cpux, Uns32 low, Uns32 high, entryInfo info) {
    union {entryInfo info; Uns64 userData;} u = {info};
    vmirtInsertRangeEntry(&cpux->tlb, low, high, u.userData);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

16.4 vmirtRemoveRangeEntry

Prototype

```
void vmirtRemoveRangeEntry(vmiRangeTablePP table, vmiRangeEntryP entry);
```

Description

Remove the passed entry for the range table and delete it.

Example

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

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

typedef struct entryInfoS {
    Uns32   pa;    // physical address
    memPriv ar;    // access permissions
} entryInfo, *entryInfoP;

// delete TLB entry
static void deleteTLBEntry(cpuxP cpux, vmiRangeEntryP tlbEntry) {
    vmirtRemoveRangeEntry(&cpux->tlb, tlbEntry);
}
```

Notes and Restrictions

1. The range entry must be present in the range table prior to removal.
2. The range entry must not be referenced once it has been removed and deleted.

QuantumLeap Semantics

Synchronizing

16.5 vmirtGetFirstRangeEntry

Prototype

```
vmiRangeEntryP vmirtGetFirstRangeEntry(
    vmiRangeTablePP table,
    Addr            low,
    Addr            high
);
```

Description

Find and return the *first* range entry in the range table that intersects the passed range `low:high`. The range entry may straddle either or both the lower or upper range bounds, or lie completely within the bounds. This function initiates a delete-safe iterator; use `vmirtGetNextRangeEntry` to find subsequent elements.

Example

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

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

typedef struct entryInfoS {
    Uns32   pa;    // physical address
    memPriv ar;    // access permissions
} entryInfo, *entryInfoP;

// find TLB entry for the passed address
static vmiRangeEntryP lookupVA(cpuxP cpux, Uns32 va) {
    return vmirtGetFirstRangeEntry(&cpux->tlb, va, va);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

16.6 vmirtGetNextRangeEntry

Prototype

```
vmiRangeEntryP vmirtGetLastRangeEntry(
    vmiRangeTablePP table,
    Addr            low,
    Addr            high
);
```

Description

Find and return the *next* range entry in the range table that intersects the passed range `low:high`, once the first entry has been found using `vmirtGetFirstRangeEntry`. The range entry may straddle either or both the lower or upper range bounds, or lie completely within the bounds. This iterator function is *delete-safe*; in other words, it may be used to iterate over entries in a table deleting them.

This function is available from VMI version 6.2.0.

Example

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

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

typedef struct entryInfoS {
    Uns32   pa;    // physical address
    memPriv ar;    // access permissions
} entryInfo, *entryInfoP;

// find next TLB entry for the passed address
static vmiRangeEntryP lookupVANext(cpuxP cpux, Uns32 vaLow, Uns32 vaHigh) {
    return vmirtGetNextRangeEntry(&cpux->tlb, vaLow, vaHigh);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

16.7 vmirtGetRangeEntryLow

Prototype

```
Addr vmirtGetRangeEntryLow(vmiRangeEntryP entry);
```

Description

Return the low address of the passed range entry.

Example

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

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

typedef struct entryInfoS {
    Uns32   pa;    // physical address
    memPriv ar;    // access permissions
} entryInfo, *entryInfoP;

// Emit debug for TLB entry
static void debugTLBEntry(vmiRangeEntryP entry) {
    union {Uns64 userData; entryInfo info;} u = {vmirtGetRangeEntryUserData(entry)};
    vmiPrintf(
        “TLB entry 0x%08x:0x%08x phys=0x%08x priv=%u\n”,
        (Uns32)vmirtGetRangeEntryLow(entry),
        (Uns32)vmirtGetRangeEntryHigh(entry),
        u.info.pa,
        u.info.ar
    );
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

16.8 vmirtGetRangeEntryHigh

Prototype

```
Addr vmirtGetRangeEntryLow(vmiRangeEntryP entry);
```

Description

Return the high address of the passed range entry.

Example

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

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

typedef struct entryInfoS {
    Uns32   pa;    // physical address
    memPriv ar;    // access permissions
} entryInfo, *entryInfoP;

// Emit debug for TLB entry
static void debugTLBEntry(vmiRangeEntryP entry) {
    union {Uns64 userData; entryInfo info;} u = {vmirtGetRangeEntryUserData(entry)};
    vmiPrintf(
        “TLB entry 0x%08x:0x%08x phys=0x%08x priv=%u\n”,
        (Uns32)vmirtGetRangeEntryLow(entry),
        (Uns32)vmirtGetRangeEntryHigh(entry),
        u.info.pa,
        u.info.ar
    );
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

16.9 vmirtGetRangeEntryUserData

Prototype

```
Uns64 vmirtGetRangeEntryUserData(vmiRangeEntryP entry);
```

Description

Return any `userData` associated with the range entry.

Example

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

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

typedef struct entryInfoS {
    Uns32   pa;    // physical address
    memPriv ar;    // access permissions
} entryInfo, *entryInfoP;

// Emit debug for TLB entry
static void debugTLBEntry(vmiRangeEntryP entry) {
    union {Uns64 userData; entryInfo info;} u = {vmirtGetRangeEntryUserData(entry)};
    vmiPrintf(
        “TLB entry 0x%08x:0x%08x phys=0x%08x priv=%u\n”,
        (Uns32)vmirtGetRangeEntryLow(entry),
        (Uns32)vmirtGetRangeEntryHigh(entry),
        u.info.pa,
        u.info.ar
    );
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

16.10 vmirtSetRangeEntryUserData

Prototype

```
void vmirtSetRangeEntryUserData(vmiRangeEntryP entry, Uns64 userData);
```

Description

Update any `userData` associated with the range entry.

Example

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

typedef struct cpuxS {
    vmiRangeTableP tlb;
} cpux, *cpuxP;

typedef struct entryInfoS {
    Uns32   pa;    // physical address
    memPriv ar;    // access permissions
} entryInfo, *entryInfoP;

// Modify access rights for TLB entry
static void setTLBEntryAR(vmiRangeEntryP entry, memPriv ar) {
    union {Uns64 userData; entryInfo info;} u = {vmirtGetRangeEntryUserData(entry)};
    u.info.ar = ar;
    vmirtSetRangeEntryUserData(entry, u.userData);
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

17 Application Program Symbol Table Access

This section describes functions allowing symbol tables of applications program loaded by processor models to be accessed.

These functions can be very useful when writing debug routines. They are also of use when writing processor semihosting plugins. Most of these routines are available only with Imperas Professional Tools.

17.1 vmirtAddressLookup

Prototype

```
memDomainP vmirtAddressLookup(
    vmiProcessorP processor,
    const char   *name,
    Addr         *simAddr
);
```

Description

Given a processor and a symbol name, this function searches for the name in any object file or symbol loaded by the processor. If a symbol of the given name is found, the function sets byref argument `simAddr` to the symbol load address and returns the memory domain object into which the object file was originally loaded. If the name cannot be found, the function returns `NULL`.

Example

This example if from the OR1K Newlib semihost library.

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    Uns32 i;

    // first few argument registers (standard ABI)
    object->args[0] = vmiosGetRegDesc(processor, "R3");
    object->args[1] = vmiosGetRegDesc(processor, "R4");
    object->args[2] = vmiosGetRegDesc(processor, "R5");

    // return register (standard ABI)
    object->result = vmiosGetRegDesc(processor, "R11");

    // stack pointer (standard ABI)
    object->sp = vmiosGetRegDesc(processor, "R1");

    // __impure_ptr address
    object->impurePtrDomain = vmirtAddressLookup(
        processor, ERRNO_REF, &object->impurePtrAddr
    );

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

    // initialize remaining file descriptors
    for(i=3; i<FILE_DES_NUM; i++) {
        object->fileDescriptors[i] = -1;
    }
}
```

Notes and Restrictions

1. See also `vmirtGetSymbolByAddr` in this section, which provides more detailed information about the symbol at a specified address.
2. This routine is the only one in this section that is available *without* an Imperas Professional Tools license.

QuantumLeap Semantics

Synchronizing

17.2 vmirtSymbolLookup

Prototype

```
const char *vmirtSymbolLookup(
    vmiProcessorP processor,
    Addr          simAddr,
    Offset       *offset
);
```

Description

Given a processor and an address, this function searches for the *nearest* symbol to the address in any object file loaded by the processor. If a nearby symbol is found, the function returns the name of the symbol, and byref argument `offset` is updated to give the offset of the address from the symbol address. If the address is not found in any object file, or the processor had no object file specified when it was loaded, the function returns `NULL`.

Example

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

// write debug message giving current execution scope
static void vmic_PrintScope(cpuxP cpux) {

    // get current program counter
    Addr        simPC  = vmirtGetPC((vmiProcessorP)cpux);
    Offset      offset;
    const char *symbol = vmirtSymbolLookup((vmiProcessorP)cpux, simPC, &offset);

    if(symbol) {
        vmiPrintf("Scope of 0x%llx is %s+%llu\n", simPC, symbol, offset);
    } else {
        vmiPrintf("Scope of 0x%llx is unknown\n", simPC);
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.

QuantumLeap Semantics

Synchronizing

17.3 vmirtAddSymbolFile

Prototype

```
Bool vmirtAddSymbolFile(
    vmiProcessorP processor,
    const char   *filename,
    Addr          address,
    const char   *sections
);
```

Description

An application program file is used by the simulator to load the program, to obtain the symbols for symbolic interception and by the product's debugger to allow symbolic debug.
The latter functionality is available only if the program is loaded explicitly by the simulator. If the program is loaded by other means, its symbols must be loaded explicitly.

This function attempts to load the symbol table from the given file into a debugger attached to the given processor. It allows the product's multi-processor debugger to debug a program that was not loaded by the simulator (i.e. the program is unknown to the debugger). It returns true if it was successful. It relies on the gdb command:

```
    add-symbol-file <file> <address> [sections]
```

Where <file> <address> and [sections] are the arguments to `vmirtAddSymbolFile`.
The optional [sections] string should be of the form:

```
    -s <section> <section-address>  -s <section> <section-address> …
```

Example

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

if(!vmirtAddSymbolFile(
    processor, "myProg.elf", 0x8000, "-s text 0x9000 –s init 0x9800"
)) {
    vmiPrintf("Unable to load\n");
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The gdb documentation explains the purpose of the –s argument to `add-symbol-file`.

QuantumLeap Semantics

Synchronizing

17.4 vmirtNextSymbolFile

Prototype

```
vmiSymbolFileCP vmirtNextSymbolFile(
    vmiProcessorP   processor,
    vmiSymbolFileCP prev
);
```

Description

This function is an iterator that, given a symbol file associated with a processor, returns the *next* symbol file associated with that processor. If `NULL` is passed as the `prev` parameter, the function returns the first symbol file associated with the processor.

Symbol files added directly to the passed processor are returned first. After this, symbol files of any parent are returned, and so on up the processor hierarchy.

The `vmiSymbolFileCP` object may be used by functions `vmirtGetSymbolFileName`, `vmirtNextSymbolByNameFile`, `vmirtNextSymbolByAddrFile`, `vmirtPrevSymbolByAddrFile`, `vmirtNextFLByAddrFile` and `vmirtPrevFLByAddrFile`.

Example

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

// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf(“new symbol file %s\n”, vmirtGetSymbolFileName(file));

        while((symbol=vmirtNextSymbolByAddrFile(file, symbol))) {
            vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
            vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
            vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
            vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
            vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolFileCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics
Synchronizing

17.5 vmirtGetSymbolFileName

Prototype

```
const char *vmirtGetSymbolFileName(vmiSymbolFileCP symbolFile);
```

Description

This function returns the name of a symbol file found by `vmirtNextSymbolFile`.

Example

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

// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf(“new symbol file %s\n”, vmirtGetSymbolFileName(file));

        while((symbol=vmirtNextSymbolByAddrFile(file, symbol))) {
            vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
            vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
            vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
            vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
            vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

17.6 vmirtGetSymbolByName

Prototype

```
vmiSymbolCP vmirtGetSymbolByName(
    vmiProcessorP processor,
    const char   *name
);
```

Description

Given a processor and a symbol name, this function searches for the name in all object files loaded by the processor. If the symbol is found in an object file, the function returns an object of type `vmiSymbolCP` describing the symbol at that address. If the symbol is not found in the object file, or the processor had no object file specified when it was loaded, the function returns `NULL`.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

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

// write a debug message giving address and details of given symbol
static void vmic_PrintSymbol(cpuxP cpux, const char *name) {

    vmiSymbolCP symbol = vmirtGetSymbolByName(((vmiProcessorP)cpux, name);

    if(symbol) {
        vmiPrintf(“Symbol ‘%s’:\n”, name);
        vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
        vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
        vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
        vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
    } else {
        vmiPrintf(“Symbol ‘%s’ is unknown\n”, name);
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

17.7 vmirtGetSymbolByAddr

Prototype

```
vmiSymbolCP vmirtGetSymbolByAddr(vmiProcessorP processor, Addr simAddr);
```

Description

Given a processor and an address, this function searches for the address in any object file loaded by the processor. If the address is defined within the object file, the function returns an object of type `vmiSymbolCP` describing the object at that address. If the address is not found in the object file, or the processor had no object file specified when it was loaded, the function returns `NULL`.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

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

// write debug message giving current execution scope
static void vmic_PrintScope(cpuxP cpux) {

    // get current program counter
    Addr        simPC  = vmirtGetPC((vmiProcessorP)cpux);
    vmiSymbolCP symbol = vmirtGetSymbolByAddr(((vmiProcessorP)cpux, simPC);

    if(symbol) {
        vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
        vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
        vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
        vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
        vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
    } else {
        vmiPrintf(“Scope of 0x%llx is unknown\n”, simPC);
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

17.8 vmirtNextSymbolByName

Prototype

```
vmiSymbolCP vmirtNextSymbolByName(
    vmiProcessorP processor,
    vmiSymbolCP   prev
);
```

Description

This function is an iterator that, given a symbol in an object file associated with a processor, returns the *next* symbol in that object file in alphabetic order. If `NULL` is passed as the `prev` parameter, the function returns the alphabetically first symbol found in the first object file associated with the processor.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

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

// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByName(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByName((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.
4. To get symbols for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtNextSymbolByNameFile` functions instead.

QuantumLeap Semantics
Synchronizing

17.9 vmirtNextSymbolByAddr

Prototype

```
vmiSymbolCP vmirtNextSymbolByAddr(
    vmiProcessorP processor,
    vmiSymbolCP   prev
);
```

Description

This function is an iterator that, given a symbol in an object file associated with a processor, returns the *next* symbol in that object file in increasing simulated address order. If `NULL` is passed as the `prev` parameter, the function returns the first symbol found in the first object file associated with the processor.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

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

// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.
4. To get symbols for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtNextSymbolByAddrFile` functions instead.

QuantumLeap Semantics
Synchronizing

17.10 vmirtPrevSymbolByAddr

Prototype

```
vmiSymbolCP vmirtPrevSymbolByAddr(
    vmiProcessorP processor,
    vmiSymbolCP   prev
);
```

Description

This function is an iterator that, given a symbol in an object file associated with a processor, returns the *previous* symbol in that object file in decreasing simulated address order. If `NULL` is passed as the `prev` parameter, the function returns the last symbol found in the first object file associated with the processor.
The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

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

// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByReverseAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtPrevSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.
4. To get symbols for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtPrevSymbolByAddrFile` functions instead.

QuantumLeap Semantics

Synchronizing

17.11 vmirtNextSymbolByNameFile

Prototype

```
vmiSymbolCP vmirtNextSymbolByNameFile(
    vmiSymbolFileCP symbolFile,
    vmiSymbolCP     prev
);
```

Description

This function is an iterator that, given a symbol in an object file, returns the *next* symbol in that object file in alphabetic order. If `NULL` is passed as the `prev` parameter, the function returns the alphabetically first symbol found in the first object file.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

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

// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByName(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf(“new symbol file %s\n”, vmirtGetSymbolFileName(file));

        while((symbol=vmirtNextSymbolByNameFile(file, symbol))) {
            vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
            vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
            vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
            vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
            vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.

QuantumLeap Semantics
Synchronizing

17.12 vmirtNextSymbolByAddrFile

Prototype

```
vmiSymbolCP vmirtNextSymbolByAddrFile(
    vmiSymbolFileCP symbolFile,
    vmiSymbolCP     prev
);
```

Description

This function is an iterator that, given a symbol in an object file, returns the *next* symbol in that object file in increasing simulated address order. If `NULL` is passed as the `prev` parameter, the function returns the first symbol found in the first object file.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

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

// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf(“new symbol file %s\n”, vmirtGetSymbolFileName(file));

        while((symbol=vmirtNextSymbolByAddrFile(file, symbol))) {
            vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
            vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
            vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
            vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
            vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.

QuantumLeap Semantics
Synchronizing

17.13 vmirtPrevSymbolByAddrFile

Prototype

```
vmiSymbolCP vmirtPrevSymbolByAddrFile(
    vmiSymbolFileCP symbolFile,
    vmiSymbolCP     prev
);
```

Description

This function is an iterator that, given a symbol in an object file, returns the *previous* symbol in that object file in decreasing simulated address order. If `NULL` is passed as the `prev` parameter, the function returns the last symbol found in the object file.

The `vmiSymbolCP` object may be further queried by functions `vmirtGetSymbolName`, `vmirtGetSymbolAddr`, `vmirtGetSymbolLoadAddr`, `vmirtGetSymbolType`, `vmirtGetSymbolBinding` and `vmirtGetSymbolSize`.

Example

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

// write information about all symbols in all symbol files associated with the
// processor
static void vmic_QueryAllSymbolsByReverseAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiSymbolCP symbol = 0;

        vmiPrintf(“new symbol file %s\n”, vmirtGetSymbolFileName(file));

        while((symbol=vmirtPrevSymbolByAddrFile(file, symbol))) {
            vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
            vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
            vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
            vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
            vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
            vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiSymbolCP` object is persistent, which means that it may be saved in a data structure for future reference if required.
3. Symbol iterator functions will find *all* entries in an ELF symbol table. Some of these will be the path and file names of the compilation units compiled into the ELF file, not real symbols. To skip these entries, use `vmirtGetSymbolType()` and refer to the `vmiSymbolTypeE` enumerations, ignoring (for example) entries of type `VMI_SYMBOL_TYPE_FILE`.

QuantumLeap Semantics
Synchronizing

17.14 vmirtGetSymbolName

Prototype

```
const char *vmirtGetSymbolName(vmiSymbolCP symbol);
```

Description

This function returns the symbol name of a `vmiSymbolCP` object.

Example

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

// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
        vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
        vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
        vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
        vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The returned string will persist until the simulation finishes.

QuantumLeap Semantics

Synchronizing

17.15 vmirtGetSymbolAddr

Prototype

```
Addr vmirtGetSymbolAddr(vmiSymbolCP symbol);
```

Description

This function returns the *virtual* address of a `vmiSymbolCP` object, i.e. the address at which of that symbol is expected to be found in an executing program. Note that this can be distinct from the *load* address of the symbol in some cases.

Example

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

// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
        vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
        vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
        vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
        vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. See also function `vmirtGetSymbolLoadAddr`, which returns the symbol *load* address.

QuantumLeap Semantics

Synchronizing

17.16 vmirtGetSymbolLoadAddr

Prototype

```
Addr vmirtGetSymbolLoadAddr(vmiSymbolCP symbol);
```

Description

This function returns the *load* address of a `vmiSymbolCP` object, i.e. the address at which that symbol was *loaded into memory*. Note that this can be distinct from the *virtual* address of the symbol in some cases.

Example

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

// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
        vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
        vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
        vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
        vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. See also function `vmirtGetSymbolAddr`, which returns the symbol *virtual* address.

QuantumLeap Semantics

Synchronizing

17.17 vmirtGetSymbolType

Prototype

```
ordSymbolType vmirtGetSymbolType(vmiSymbolCP symbol);
```

Description

This function returns the type of a `vmiSymbolCP` object. The type is a member of an enumeration defined in `ImpPublic/include/host/ord/ordTypes.h`:

```
typedef enum ordSymbolTypeE {
    ORD_SYMBOL_TYPE_SECTION,
    ORD_SYMBOL_TYPE_NONE,
    ORD_SYMBOL_TYPE_OBJECT,
    ORD_SYMBOL_TYPE_FUNC,
    ORD_SYMBOL_TYPE_FILE
} ordSymbolType;
```

Symbol type information is typically used when writing introspection or profiling utilities.

Example

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

// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
        vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
        vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
        vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
        vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. Symbol types are not available in all file formats. The type will default to `ORD_SYMBOL_TYPE_NONE`.

QuantumLeap Semantics

Synchronizing

17.18 vmirtGetSymbolBinding

Prototype

```
ordSymbolBinding vmirtGetSymbolBinding(vmiSymbolCP symbol);
```

Description

This function returns the binding of a `vmiSymbolCP` object. The type is a member of an enumeration defined in `ImpPublic/include/host/ord/ordTypes.h`:

```
typedef enum ordSymbolBindingE {
    ORD_SYMBOL_BIND_LOCAL,
    ORD_SYMBOL_BIND_WEAK,
    ORD_SYMBOL_BIND_GLOBAL
} ordSymbolBinding;
```

Symbol binding information is typically used when writing introspection or profiling utilities.

Example

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

// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf("Symbol '%s':\n", vmirtGetSymbolName(symbol));
        vmiPrintf("virtual address: 0x%llx\n", vmirtGetSymbolAddr(symbol));
        vmiPrintf("load address   : 0x%llx\n", vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf("type           : 0x%x\n",   vmirtGetSymbolType(symbol));
        vmiPrintf("binding        : 0x%x\n",   vmirtGetSymbolBinding(symbol));
        vmiPrintf("size           : 0x%llx\n", vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. Binding is not reported by all file formats. The result defaults to `ORD_SYMBOL_BIND_LOCAL`.

QuantumLeap Semantics

Synchronizing

17.19 vmirtGetSymbolSize

Prototype

```
Addr vmirtGetSymbolSize(vmiSymbolCP symbol);
```

Description

This function returns the size in bytes of the symbol specified by a `vmiSymbolCP` object.

Example

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

// write information about all symbols in the object file
static void vmic_QueryAllSymbolsByAddr(cpuxP cpux) {

    vmiSymbolCP symbol = 0;

    while((symbol=vmirtNextSymbolByAddr((vmiProcessorP)cpux, symbol))) {
        vmiPrintf(“Symbol ‘%s’:\n”, vmirtGetSymbolName(symbol));
        vmiPrintf(“virtual address: 0x%llx\n”, vmirtGetSymbolAddr(symbol));
        vmiPrintf(“load address   : 0x%llx\n”, vmirtGetSymbolLoadAddr(symbol));
        vmiPrintf(“type           : 0x%x\n”,   vmirtGetSymbolType(symbol));
        vmiPrintf(“binding        : 0x%x\n”,   vmirtGetSymbolBinding(symbol));
        vmiPrintf(“size           : 0x%llx\n”, vmirtGetSymbolSize(symbol));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. Depending on the symbol type, the size might correspond to the size of a represented data object, the length of a function or the total size of a section. Not all symbol types have a specified size, depending on the file format. The size will default to zero.

QuantumLeap Semantics

Synchronizing

18 Application Dwarf Line Number Information Access

This section describes functions allowing Dwarf-format line number information from an of the application program loaded by a processor model to be accessed. This information is only present if the application program has been compiled for debug.

These functions can be very useful when writing debug routines. They are also of use when writing processor semihosting plugins. These routines are available only with Imperas Professional Tools.

18.1 vmirtGetFLByAddr

Prototype

```
vmiFileLineCP vmirtGetFLByAddr(vmiProcessorP processor, Addr simAddr);
```

Description

Given a processor and an address, this function searches for the address in the Dwarf information for any object file loaded by the processor. If the address is found, the function returns an object of type `vmiFileLineCP` describing the object at that address. If the address is not found in the object file, or the processor had no object file specified when it was loaded, or the application was not compiled for debug, the function returns `NULL`.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

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

// write debug message giving current line
static void vmic_CurrentLine(cpuxP cpux) {

    // get current program counter
    Addr          simPC = vmirtGetPC((vmiProcessorP)cpux);
    vmiFileLineCP fl    = vmirtGetFLByAddr(((vmiProcessorP)cpux, simPC);

    if(fl) {
        vmiPrintf(“Scope of 0x%llx is:\n”, simPC);
        vmiPrintf(“file   : %s\n”,     vmirtGetFLFileName(fl));
        vmiPrintf(“line   : %u\n”,     vmirtGetFLLineNumber(fl));
        vmiPrintf(“address: 0x%llx\n”, vmirtGetFLAddr(fl));
    } else {
        vmiPrintf(“Scope of 0x%llx is unknown\n”, simPC);
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

18.2 vmirtNextFLByAddr

Prototype

```
vmiFileLineCP vmirtNextFLByAddr(
    vmiProcessorP processor,
    vmiFileLineCP prev
);
```

Description

This iterator allows all Dwarf file/line records in the object file associated with the processor to be processed in increasing simulated address order.

The first file/line record in the first object file associated with the processor is found by passing `NULL` as the `prev` parameter; subsequent file/line records are found by passing the previous file/line record found as the `prev` parameter.

To get file/line records for all object files associated with the processor use the `vmirtNextSymbolFile` and `vmirtNextFLByAddrFile` functions instead.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

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

// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtNextFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf("New FL record:\n");
        vmiPrintf("file   : %s\n",     vmirtGetFLFileName(fl));
        vmiPrintf("line   : %u\n",     vmirtGetFLLineNumber(fl));
        vmiPrintf("address: 0x%llx\n", vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.
3. To get file/line records for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtNextFLByAddrFile` functions instead.

QuantumLeap Semantics

Synchronizing

18.3 vmirtPrevFLByAddr

Prototype

```
vmiFileLineCP vmirtNextFLByAddr(
    vmiProcessorP processor,
    vmiFileLineCP prev
);
```

Description

This iterator allows all Dwarf file/line records in the object file associated with the processor to be processed in decreasing simulated address order.

The last file/line record in the first object file associated with the processor is found by passing `NULL` as the `prev` parameter; subsequent file/line records are found by passing the previous file/line record found as the `prev` parameter.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

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

// write information about all lines in the object file
static void vmic_QueryAllLinesByReverseAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtPrevFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf(“New FL record:\n”);
        vmiPrintf(“file   : %s\n”,     vmirtGetFLFileName(fl));
        vmiPrintf(“line   : %u\n”,     vmirtGetFLLineNumber(fl));
        vmiPrintf(“address: 0x%llx\n”, vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.
3. To get file/line records for all object files associated with the processor rather than just the first object file use the `vmirtNextSymbolFile` and `vmirtNextFLByAddrFile` functions instead.

QuantumLeap Semantics

Synchronizing

18.4 vmirtNextFLByAddrFile

Prototype

```
vmiFileLineCP vmirtNextFLByAddrFile(
    vmiSymbolFileCP symbolFile,
    vmiFileLineCP   prev
);
```

Description

This iterator allows all Dwarf file/line records in the object file to be processed in increasing simulated address order. The first file/line record is found by passing `NULL` as the `prev` parameter; subsequent file/line records are found by passing the previous file/line record found as the `prev` parameter.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

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

// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiFileLineCP fl = 0;

        vmiPrintf(“new symbol file %s\n”, vmirtGetSymbolFileName(file));

        while((fl=vmirtNextFLByAddrFile(file, fl))) {
            vmiPrintf(“New FL record:\n”);
            vmiPrintf(“file   : %s\n”,     vmirtGetFLFileName(fl));
            vmiPrintf(“line   : %u\n”,     vmirtGetFLLineNumber(fl));
            vmiPrintf(“address: 0x%llx\n”, vmirtGetFLAddr(fl));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

18.5 vmirtPrevFLByAddrFile

Prototype

```
vmiFileLineCP vmirtNextFLByAddrFile(
    vmiSymbolFileCP symbolFile,
    vmiFileLineCP   prev
);
```

Description

This iterator allows all Dwarf file/line records in the object file to be processed in decreasing simulated address order. The last file/line record in the first object file associated with the processor is found by passing `NULL` as the `prev` parameter; subsequent file/line records are found by passing the previous file/line record found as the `prev` parameter.

The `vmiFileLineCP` object maybe further queried by functions `vmirtGetFLFileName`, `vmirtGetFLLineNumber` and `vmirtGetFLAddr`.

Example

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

// write information about all lines in the object file
static void vmic_QueryAllLinesByReverseAddr(cpuxP cpux) {

    vmiSymbolFileCP file = 0;

    while((file=vmirtNextSymbolFile((vmiProcessorP)cpux, file))) {

        vmiFileLineCP fl = 0;

        vmiPrintf(“new symbol file %s\n”, vmirtGetSymbolFileName(file));

        while((fl=vmirtPrevFLByAddrFile(file, fl))) {
            vmiPrintf(“New FL record:\n”);
            vmiPrintf(“file   : %s\n”,     vmirtGetFLFileName(fl));
            vmiPrintf(“line   : %u\n”,     vmirtGetFLLineNumber(fl));
            vmiPrintf(“address: 0x%llx\n”, vmirtGetFLAddr(fl));
        }
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.
2. The `vmiFileLineCP` object is persistent, which means that it may be saved away in a data structure for future reference if required.

QuantumLeap Semantics

Synchronizing

18.6 vmirtGetFLFileName

Prototype

```
const char *vmirtGetFLFileName(vmiFileLineCP fl);
```

Description

This function returns the file name of a `vmiFileLineCP` object. This is the file name of the file in which the item corresponding to the file/line record was defined.

Example

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

// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtNextFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf(“New FL record:\n”);
        vmiPrintf(“file   : %s\n”,     vmirtGetFLFileName(fl));
        vmiPrintf(“line   : %u\n”,     vmirtGetFLLineNumber(fl));
        vmiPrintf(“address: 0x%llx\n”, vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.

QuantumLeap Semantics

Synchronizing

18.7 vmirtGetFLLineNumber

Prototype

```
Uns32 vmirtGetFLLineNumber(vmiFileLineCP fl);
```

Description

This function returns the line number of a `vmiFileLineCP` object. This is the line number in the file at which the item corresponding to the file/line record was defined.

Example

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

// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtNextFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf(“New FL record:\n”);
        vmiPrintf(“file   : %s\n”,     vmirtGetFLFileName(fl));
        vmiPrintf(“line   : %u\n”,     vmirtGetFLLineNumber(fl));
        vmiPrintf(“address: 0x%llx\n”, vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.

QuantumLeap Semantics

Synchronizing

18.8 vmirtGetFLAddr

Prototype

```
Addr vmirtGetFLAddr(vmiFileLineCP fl);
```

Description

This function returns the simulated address of a `vmiFileLineCP` object. This is the address in the application executable file at which the item corresponding to the file/line record is located.

Example

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

// write information about all lines in the object file
static void vmic_QueryAllLinesByAddr(cpuxP cpux) {

    vmiFileLineCP fl = 0;

    while((fl=vmirtNextFLByAddr((vmiProcessorP)cpux, fl))) {
        vmiPrintf(“New FL record:\n”);
        vmiPrintf(“file   : %s\n”,     vmirtGetFLFileName(fl));
        vmiPrintf(“line   : %u\n”,     vmirtGetFLLineNumber(fl));
        vmiPrintf(“address: 0x%llx\n”, vmirtGetFLAddr(fl));
    }
}
```

Notes and Restrictions

1. This routine is only available for products with an Imperas Professional Tools license.

QuantumLeap Semantics

Synchronizing

19 Licensing

This section describes functions allowing models to be licensed using the Imperas license manager daemon. In order to use these functions, a license key for the model needs to be created for use with the Imperas daemon – contact Imperas for more information.

19.1 vmirtGetLicense

Prototype

```
Bool vmirtGetLicense(const char *name);
```

Description

This routine attempts to check out a license for the current model. The `Bool` return code indicates whether the license was successfully checked out. This function ***must*** be called from within a processor model constructor. If the 'name' field is NULL, then the license feature is derived from the simulator product name and the variant or name of the model, otherwise the 'name' string is checked out from the license server.

Example

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

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

    // Check for valid CPU license
    const char *name = "CPUX_LICENSE_FEATURE"
    if (!vmirtGetLicense(name)) {
        vmiMessage("F", "LIC_NA2", "%s", vmirtGetLicenseeErrString(name));
    }

    . . . etc . . .
}
```

Notes and Restrictions

1. Must be called from the model constructor.

QuantumLeap Semantics

Synchronizing

19.2 vmirtGetLicenseErrString

Prototype

```
const char *vmirtGetLicenseErrString(const char *name);
```

Description

When function `vmirtGetLicense` 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.

The `name` argument must be the same as what was passed on the failing `vmirtGetLicense` call - either `NULL` or the name of the license to check out.

Example

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

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

    // Check for valid CPU license
    const char *name = "CPUX_LICENSE_FEATURE"
    if (!vmirtGetLicense(name)) {
        vmiMessage("F", "LIC_NA2", "%s", vmirtGetLicenseeErrString(name));

    }
    . . . etc . . .
}
```

Notes and Restrictions

1. Must be called from the model constructor.
2. Should only be called after `vmirtGetLicense` has failed.
3. Must be passed the same argument that was used on the failed `vmirtGetLicense` call.

QuantumLeap Semantics

Synchronizing

20 View Provider Interface

This section describes functions to create and provide view objects.

20.1 vmirtGetProcessorViewObject

Prototype

```
vmiViewObjectP vmirtGetProcessorViewObject(vmiProcessorP processor);
```

Description

Return the base view object for a processor.

Example

```
#include "vmi/vmiRt.h"
#include "vmi/vmiView.h"
#include "vmi/vmiAttrs.h"
#include "vmi/vmiTypes.h"
#include "vmi/vmiMessage.h"

TBD
```

Notes and Restrictions

TBD

QuantumLeap Semantics

Synchronizing

20.2 vmirtAddViewObject

Prototype

```
vmiViewObjectP vmirtAddViewObject(
    vmiViewObjectP parent,
    const char    *name,
    const char    *description
);
```

Description

Create a view object object.

`parent` is a pointer to the parent.

`description` may be 0.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiView.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

TBD
```

Notes and Restrictions

TBD

QuantumLeap Semantics

Synchronizing

20.3 vmirtSetViewObjectConstValue

Prototype

```
void vmirtSetViewObjectConstValue(
    vmiViewObjectP   object,
    vmiViewValueType type,
    void            *pValue
);
```

Description

Set constant value for view object (value copied at time of call).

`pValue` is a pointer to item.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiView.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

TBD
```

Notes and Restrictions

TBD

QuantumLeap Semantics

Synchronizing

20.4 vmirtSetViewObjectRefValue

Prototype

```
void vmirtSetViewObjectRefValue(
    vmiViewObjectP   object,
    vmiViewValueType type,
    void            *pValue
);
```

Description

Set value pointer for view object (pointer dereferenced each time value is viewed). Use this to associate a view object with a C variable in the model such that the variable is automatically read when the view object is evaluated.

`pValue` is a pointer to item in persistent memory (must be valid for lifetime of object)

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiView.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

TBD
```

Notes and Restrictions

TBD

QuantumLeap Semantics

Synchronizing

20.5 vmirtSetViewObjectValueCallback

Prototype

```
void vmirtSetViewObjectValueCallback(
    vmiViewObjectP object,
    vmiViewValueFn valueCB,
    void          *userData
);
```

Description

Set value callback for view object.

`valueCB` will be passed the `userData` value and should be declared using the `VMI_VIEW_VALUE_FN` macro:

```
VMI_VIEW_VALUE_FN(valueCB) {
 ...
}
```

See the documentation for the vmiviewGetViewObjectValue function the VMI View Function Reference Manual for more info on what this function is expected to return.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiView.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

TBD
```

Notes and Restrictions

TBD

QuantumLeap Semantics

Synchronizing

20.6 vmirtAddViewAction

Prototype

```
void vmirtAddViewAction(
    vmiViewObjectP  object,
    const char     *name,
    const char     *description,
    vmiViewActionFn actionCB,
    void           *userData
);
```

Description

Add an action to a view object object.

`actionCB` will be passed the `userData` value and should be declared using the `VMI_VIEW_ACTION_FN` macro:

```
VMI_VIEW_ACTION_FN(actionCB) {
 ...
}
```

`description` may be 0.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiView.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

TBD
```

Notes and Restrictions

TBD

QuantumLeap Semantics

Synchronizing

20.7 vmirtAddViewEvent

Prototype

```
vmiViewEventP vmirtAddViewEvent(
    vmiViewObjectP object,
    const char    *name,
    const char    *description
);
```

Description

Add an event to a view object.

`description` may be 0.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiView.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

TBD
```

Notes and Restrictions

TBD

QuantumLeap Semantics

Synchronizing

20.8 vmirtNextViewEvent

Prototype

```
vmiViewEventP vmirtNextViewEvent(
    vmiViewObjectP object,
    vmiViewEventP  old
);
```

Description

Iterate through the view events on a view object.

`old` should be set to 0 for the first call, then the returned value used for each subsequent call until 0 is returned.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiView.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

...
  vmiViewEventP v = NULL;
  while ((v = vmirtNextViewEvent(object, v))) {
      // use v here
  }
...
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

20.9 vmirtTriggerViewEvent

Prototype

```
void vmirtTriggerViewEvent(vmiViewEventP event);
```

Description

Trigger a view event.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiView.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

TBD
```

Notes and Restrictions

TBD

QuantumLeap Semantics

Synchronizing

20.10 vmirtDeleteViewObject

Prototype

```
void vmirtDeleteViewObject(vmiViewObjectP object);
```

Description

Delete a view object.

Example

```
#include “vmi/vmiRt.h”
#include “vmi/vmiView.h”
#include “vmi/vmiAttrs.h”
#include “vmi/vmiTypes.h”
#include “vmi/vmiMessage.h”

TBD
```

Notes and Restrictions

TBD

QuantumLeap Semantics

Synchronizing

21 Runtime Commands

A runtime command is part of a model or plugin which can be executed by the simulator. Its typical use is to change the mode of operation of the model or to print information from inside it.

Creating a command installs a C callback function in the model that will be called when the command is executed. A command can be given arguments, which are either passed to the callback function exactly as given, or parsed inside the simulator by a standard parser and the resulting converted values passed to the callback function. The latter strategy is preferable because it results in standardized argument handling, allows the simulator to generate help messages and lets other tools use the command in a standard way.

To receive raw arguments exactly as supplied to the command, use `vmirtAddCommand`() to define the command.

To receive arguments that have been validated and converted from strings to the desired types, use `vmirtAddCommandParse`() to define the command, and use `vmirtAddArg`() to define the argument names, types and usage.

Commands can be called in several ways:

- From the platform ICM API. See the function `icmCallCommand`() in *OVPsim_and_CpuManager_User_Guide.doc* and the simulator option `--callcommand` in *Imperas_Simulation_Guide.doc*
- From a simulator control file or simulator command line, using –callcommand.
- From the Imperas multicore debugger. Commands appear in the TCL interpreter.
- From a graphical interface.

Arguments to `vmirtAddCommand` & `vmirtAddCommandParse` control how the command will appear in a graphical environment (if at all).

21.1 vmirtAddCommand

Prototype

```
void vmirtAddCommand(
    vmiProcessorP    processor,
    const char      *name,
    const char      *help,
    vmirtCommandFn   commandCB,
    vmiCommandAttrs  attrs
);
```

Description

Add a command to the given processor. The arguments are as follows:

`processor`	The current processor.
`name`	The name used to call the command.
`help`	Describe the arguments required by the command.
`commandCB`	The function to be called when the command is executed.
`attrs`	Control how the command appears in a graphical interface.

When the command is called, the function `commandCB` will be called in the model. It is passed the processor context, the number of arguments supplied to the command (`argc`) and arguments as an array of strings called `argv[]`. `argv[0]` is the command name.

Example

```
#include “vmi/vmiRt.h”

static VMIRT_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]);
    }
}

VMI_CONSTRUCTOR_FN(constructor) {
...
    vmirtAddCommand(
        processor,
        "myCommand",
        "<filename> <duration> [option]",
        myCommand,
        VMI_CT_DEFAULT
    );
}
```

Notes and Restrictions

Strings passed to the callback will not persist after the callback is complete. Commands created using this function can be made visible in a graphical environment, but their arguments and usage will not be visible.

QuantumLeap Semantics

Synchronizing

21.2 vmirtAddCommandParse

Prototype

```
vmiCommandP vmirtAddCommandParse(
    vmiProcessorP       processor,
    const char         *name,
    const char         *help,
    vmirtCommandParseFn commandCB,
    vmiCommandAttrs     attrs
);
```

Description

Add a command to the given processor. The arguments are as follows:

`processor`	The current processor.
`name`	The name used to call the command.
`help`	Describe the arguments required by the command.
`commandCB`	The function to be called when the command is executed.
`attrs`	Control how the command appears in a graphical interface.

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 called function is passed the processor, 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`().

The argument value structure array contains the following information:

Field name	Contains
name	argument name (without the -)
type	data type
isSet	true if the parser found this argument in the call
changed	true if the argument has a static value which was modified by this call.
value	a union of possible value types. Must be accessed as the right type.

If the arguments to the command have persistent values which are of interest to the user, `vmirtAddArg` can be used to inform the simulator how to retrieve them (see `vmirtAddArg`).

Example

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

// This is the place that stores permanent values of the flags (if required)
typedef struct staticValuesS {
    Addr   arg1Value;
    Bool   arg2Value;
    Float  arg3Value;
    Int32  arg4Value;
    Char  *arg5Value;
} staticValues;

staticValues *sv = malloc(sizeof(staticValues));
```

```
Bool myStaticBoolValue;

//
// Called by simulator to retrieve the value of this argument
// (for display by a GUI for example)
//
static VMIRT_ARG_VALUE_FN(arg6ValueFn) {
    *valuePtr = myStaticBoolValue;
}

static VMIRT_COMMAND_PARSE_FN(myCommand) {
    vmiPrintf("myCommand was called, with parsed values:\n");
    int i;
    for(i= 1; i < argc) {
        vmiPrintf("   arg %d is %s\n", i, arg[i].name);
        switch() {
            case VMI_CA_ADDRESS:
                 vmiPrintf("   address = 0x%llx\n", arg[i].u.addr);
                 break;
            case VMI_CA_FLAG:
                 vmiPrintf("   bool    = %s\n", arg[i].u.flag ? “True” : “False”);
                 break;
            case VMI_CA_FLOAT:
                 vmiPrintf("   float   = %f\n", arg[i].u.flt);
                 break;
            case VMI_CA_INTEGER:
                 vmiPrintf("   integer = %d\n", arg[i].u.integer);
                 break;
            case VMI_CA_STRING:
                 vmiPrintf("   string = %s\n", arg[i].u.string);
                 break;
            default:
                 break;
       }
       vmiPrintf(“       %s\n”,  argv[i].isSet     ? “Set”     : “Unset”     );
       vmiPrintf(“       %s\n\n”,  argv[i].changed ? “Changed” : “Unchanged” );
    }
}

VMI_CONSTRUCTOR_FN(constructor) {
...
vmiCommandP cmd = vmirtAddCommandParse(
    processor,
    "myCommand",
    "Do the operation ‘myCommand’",
    myCommand,
    VMI_CT_DEFAULT
);

vmiAddArg(
    cmd,                    // command handle
    “arg1”,                 // name of argument
    “arg1 is the address”,  // help
    VMI_CA_ADDRESS,         // type
    VMI_CAA_MENU,           // will appear in a GUI
    &sv->arg1Value
);

vmiAddArg(
    cmd,                    // command handle
    “arg2”,                 // name of argument
    “arg2 is a boolean”,            // help
    VMI_CA_FLAG,                    // type
    VMI_CAA_MENU | VMI_CAA_ENABLE,  // will appear in a GUI, turns the feature on
    &sv->arg2Value
);

vmiAddArg(
    cmd,                    // command handle
    “arg3”,                 // name of argument
    “arg3 is a float”,      // help
```

```
    VMI_CA_FLOAT,           // type
    VMI_CAA_MENU,           // will appear in a GUI
    &sv->arg3Value
);

vmiAddArg(
    cmd,                    // command handle
    "arg4",                 // name of argument
    "arg4 is an integer",   // help
    VMI_CA_INTEGER,         // type
    VMI_CAA_MENU,           // will appear in a GUI
    &sv->arg4Value
);

vmiAddArg(
    cmd,                    // command handle
    "arg5",                 // name of argument
    "arg5 is a string",     // help
    VMI_CA_STRING,          // type
    VMI_CAA_MENU,           // will appear in a GUI
    &sv->arg5Value
);


vmiAddArg(
    cmd,                     // command handle
    "arg6",                  // name of argument
    "arg6 is a boolean",     // help
    VMI_CA_INTEGER,          // type
    VMI_CAA_MENU
   |VMI_CAA_VALUE_CALLBACK,  // will appear in a GUI. Can be retrieved by callback
    arg6ValueFn              // function to be used
);
```

Example usage

```
-callcommand "procA/mycommand  –arg3 1.5 –arg2 Y –arg4 44 –arg5 maybe –arg6 –arg1
0xFF000000"

[output]
myCommand was called, with parsed values:
   arg[0] is arg1
    address = 0xFF000000
    Set
    Changed

   arg[1] is arg2
    boolean =  True
    Set
    Changed

   arg[2] is arg3
    float   =  1.500000
    Set
    Changed

   arg[3] is arg4
    integer =  44
    Set
    Changed

   arg[4] is arg5
    string   =  maybe
    Set
    Changed

   arg[5] is arg6
    boolean = True
    Set
    Changed
```

Notes and Restrictions
The array of argument values passed to the callback will not persist after the callback is complete.
When a command is called, its arguments need not be provided in the order they were specified.

QuantumLeap Semantics
Synchronizing

21.3 vmirtAddArg

Prototype

```
vmiArgP vmirtAddArg(
    vmiCommandP         command,
    const char         *name,
    const char         *help,
    vmiArgType          type,
    vmiArgAttrs         attrs,
    Bool                mandatory,
    Void               *dataPointer
);
```

Description

Add an argument specification to the passed command. When the command is called the command parser will recognize an argument of this name (preceded by '-') and take the next item as its value, converting it according to type and putting it in the value array passed to the command callback.

`name` The name of the argument to be recognized by the parser (preceded by '-')
`help` Description of the argument.
`type` The data-type of the argument which should be to one of the following values:

Name	Meaning
VMI_CA_ADDRESS	An address, currently a 64-bit unsigned integer
VMI_CA_ENUMERATION	Not supported – do not use.
VMI_CA_FLAG	Boolean. If no value follows, taken to be true. Otherwise values 1,T,Y or 0,F,N are accepted as true or false respectively.
VMI_CA_FLOAT	Floating point number
VMI_CA_INTEGER	A 32-bit signed integer
VMI_CA_STRING	A null-terminated C string.

`Attrs` A bit-mask used to modify the behavior of the command when used with a graphical user interface. The 32-bit value should be constructed with the following macros from `vmiCommand.h`:

Name	Meaning
VMI_CAA_DEFAULT	Not used in a graphical interface
VMI_CAA_MENU	Is used in a graphical interface
VMI_CAA_ENABLE	This argument is used to enable the feature controlled by the command
VMI_CAA_VALUE_CALLBACK	The data pointer refers to a callback function rather than to the data.

`dataPointer` Refers to the location in the model that permanently stores the argument value, or to a function that can be called to retrieve the value, depending on the value of the `attrs` bit defined by VMI_CAA_VALUE_CALLBACK.
If `dataPointer` is set to null then a graphical interface will be unable to display the current value of this argument.

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

21.4 Command attributes

The last argument to `vmirtAddCommand` and `vmirtAddCommandParse` is a bitmask used to modify the behavior of the command when used with a graphical user interface. The 32-bit value should be constructed with the following macros from `vmiCommand.h`:

macro	**meaning**
VMI_CT_DEFAULT	Do not use this command in graphical interfaces
VMI_CT_MASK	mask for the command type:
VMI_CT_QUERY	command used to query the model, causing an immediate response.
VMI_CT_STATUS	command used to query the model. Should be called each time the simulator stops.
VMI_CT_MODE	command changes the mode of the model
VMI_CO_MASK	Command object type mask. Puts the command in a category depending on the object the command operates on.
VMI_CO_CACHE	Cache models
VMI_CO_CPU	Processor models
VMI_CO_CONTEXT	Application code stack trace
VMI_CO_DIAG	Model diagnostics
VMI_CO_FUNCTION	Application source code functions
VMI_CO_GIC	Interrupt controller
VMI_CO_LINE	Application code source lines
VMI_CO_LKM	Loadable kernel modules
VMI_CO_MEMORY	Memory
VMI_CO_OS	Operating system, RTOS or scheduler
VMI_CO_TASK	Operating system tasks or processes
VMI_CO_TLB	Translation buffer
VMI_CO_SYMBOL	Application code source symbols
VMI_CA_MASK	Command action type mask. Puts the command in a category according to its action.
VMI_CA_CONTROL	Command controls how the model operates (e.g. on or off)
VMI_CA_COVER	Command associated with code coverage
VMI_CA_PROFILE	Command associated with code profiling
VMI_CA_QUERY	Command to query the model
VMI_CA_REPORT	Command to print a report
VMI_CA_TRACE	Command to control tracing
VMI_CT_HELP	If set, the simulator will add the –help argument and supply its response

22 Processor Register, Mode, Exception and Port Access Utilities

A set of functions is available that allows processor registers to be read and written using the register interface presented using `vmiRegInfo` objects. Similarly, functions are available allowing processor mode and exception state to be queried using the interface presented using `vmiModeInfo` and `vmiExceptionInfo` objects. It is also possible to query the port connections of a processor using the interface presented by `vmiBusPort`, `vmiNetPort` and `vmiFifoPort` objects.

This interface is typically used in intercept libraries to query and update processor state without requiring access to processor private data structures.

22.1 vmirtGetRegGroupByName

Prototype

```
vmiRegGroupCP vmirtGetRegGroupByName(
    vmiProcessorP processor,
    const char   *name
);
```

Description

This function returns a pointer to a `vmiRegGroupCP` structure implementing the named register group. The returned pointer can subsequently be used with `vmirtGetNextRegInGroup`, allowing iteration of all registers in a group.

If a register group with the specified name is not found, a null pointer is returned.

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiRegGroupCP group = vmirtGetRegGroupByName(processor, "GPR");
    vmiRegInfoCP  reg   = NULL;
    Uns32         i     = 0;

    // count all defined GPRs
    while((reg=vmirtGetNextRegInGroup(processor, group, reg))) {
        object->regNum++;
    }

    // allocate array for register handles
    object->regs = STYPE_CALLOC_N(vmiRegInfoCP, object->regNum);

    // fill array of GPRs for later use
    while((reg=vmirtGetNextRegInGroup(processor, group, reg))) {
        object->regs[i++] = reg;
    }

    . . . etc . . .
 }
```

Notes and Restrictions

1. Register groups can only be found for processors that define the `regGroupCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetRegGroupByName` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.2 vmirtGetNextRegGroup

Prototype

```
vmiRegGroupCP vmirtGetNextRegGroup(
    vmiProcessorP processor,
    vmiRegGroupCP previous
);
```

Description

Given a processor and a `vmiRegGroupCP` structure implementing a register group in that processor, this function returns the *next* defined register group for that processor. The returned pointer can subsequently be used with `vmirtGetNextRegInGroup`, allowing iteration of all registers in a group. The function returns `NULL` if there are no more register group definitions.

Typically, `vmirtGetNextRegGroup` will be used in the constructor of a semi-hosting library to initialize pointers used to access registers in that group.

Notes and Restrictions

1. Registers groups can only be iterated for processors that define the `regGroupCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextRegGroup` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.3 vmirtGetRegByName

Prototype

```
vmiRegInfoCP vmirtGetRegByName(
    vmiProcessorP  processor,
    const char    *name
);
```

Description

This function returns a pointer to a `vmiRegInfoCP` structure implementing the named register. The returned pointer can subsequently be used with `vmirtRegRead` and `vmirtRegWrite`. This allows the registers within a processor model to be accessed without requiring any knowledge of the internal structure of that processor model, other than the names of the registers.

If a register with the specified name is not found, a null pointer is returned.

Typically, `vmirtGetRegByName` will be used in the constructor of a semi-hosting library to initialize pointers used to access the registers used in the calling conventions of that processor.

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    // first three argument registers (standard ABI)
    object->args[0] = vmirtGetRegByName(processor, “a0”);
    object->args[1] = vmirtGetRegByName(processor, “a1”);
    object->args[2] = vmirtGetRegByName(processor, “a2”);

    . . . etc . . .
  }
```

Notes and Restrictions

1. In VMI versions prior to 5.0.0, this function was called `vmiosGetRegDesc` and prototyped in file `vmiOSLib.h`. It has been moved to `vmiRt.h` because it is of more general applicability than semihosting libraries alone.

QuantumLeap Semantics

Non-self-synchronizing

22.4 vmirtGetNextReg

Prototype

```
vmiRegInfoCP vmirtGetNextReg(
    vmiProcessorP  processor,
    vmiRegInfoCP   previous
);
```

Description

Given a processor and a `vmiRegInfoCP` structure implementing a register in that processor, this function returns the *next* defined register for that processor. The returned pointer can subsequently be used with `vmirtRegRead` and `vmirtRegWrite`. If `NULL` is passed as the `previous` argument, the *first* register definition in the processor is returned. The function returns `NULL` if there are no more register definitions.

Typically, `vmirtGetNextReg` will be used in the constructor of a semi-hosting library to initialize pointers used to access those registers.

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiRegInfoCP reg = NULL;
    Uns32        i   = 0;

    // count all defined registers
    while((reg=vmirtGetNextReg(processor, reg))) {
        object->regNum++;
    }

    // allocate array for register handles
    object->regs = STYPE_CALLOC_N(vmiRegInfoCP, object->regNum);

    // fill array of registers for later use
    while((reg=vmirtGetNextReg(processor, reg))) {
        object->regs[i++] = reg;
    }

    . . . etc . . .
  }
```

Notes and Restrictions

1. Registers can only be iterated for processors that define the `regInfoCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextReg` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.5 vmirtGetNextRegInGroup

Prototype

```
vmiRegInfoCP vmirtGetNextRegInGroup(
    vmiProcessorP processor,
    vmiRegGroupCP group,
    vmiRegInfoCP  previous
);
```

Description

Given a processor and a `vmiRegInfoCP` structure implementing a register in that processor, this function returns the *next* defined register for that processor in the given group. The returned pointer can subsequently be used with `vmirtRegRead` and `vmirtRegWrite`. If `NULL` is passed as the `previous` argument, the *first* register definition in the processor group is returned. The function returns `NULL` if there are no more register definitions.

Typically, `vmirtGetNextRegInGroup` will be used in the constructor of a semi-hosting library to initialize pointers used to access those registers.

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiRegGroupCP group = vmirtGetRegGroupByName(processor, "GPR");
    vmiRegInfoCP  reg   = NULL;
    Uns32         i     = 0;

    // count all defined GPRs
    while((reg=vmirtGetNextRegInGroup(processor, group, reg))) {
        object->regNum++;
    }

    // allocate array for register handles
    object->regs = STYPE_CALLOC_N(vmiRegInfoCP, object->regNum);

    // fill array of GPRs for later use
    while((reg=vmirtGetNextRegInGroup(processor, group, reg))) {
        object->regs[i++] = reg;
    }

    . . . etc . . .
 }
```

Notes and Restrictions

1. Registers can only be iterated for processors that define the `regInfoCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextRegInGroup` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.6 vmirtRegRead

Prototype

```
Bool vmirtRegRead(
    vmiProcessorP processor,
    vmiRegInfoCP  regDesc,
    void         *result
);
```

Description

Given a processor and a register description, this function copies the current value of the register to the result buffer. This is typically used in an intercepted function call to obtain the value of a function argument using the standard processor ABI.

Example

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

static VMIOS_INTERCEPT_FN(exitCB) {

    Int32 status;
    Int32 result = 0;

    // obtain function arguments
    vmirtRegRead(processor, object->arg1, &status);

    // implement exit
    vmirtExit(processor);

    // write back results
    setErrnoAndResult(processor, object, result, context);
}
```

Notes and Restrictions

1. It is the caller's responsibility to ensure that the buffer is large enough to hold the register value.
2. In VMI versions prior to 5.0.0, this function was called `vmiosRegRead` and prototyped in file `vmiOSLib.h`. It has been moved to `vmiRt.h` because it is of more general applicability than semihosting libraries alone.

QuantumLeap Semantics

Non-self-synchronizing

22.7 vmirtRegWrite

Prototype

```
Bool vmirtRegWrite(
    vmiProcessorP processor,
    vmiRegInfoCP  regDesc,
    const void   *value
);
```

Description

Given a processor and a register description, this function sets the current value of the register using the passed value. This is typically used in an intercepted function call to return a function result using the standard processor ABI.

Example

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

static VMIOS_INTERCEPT_FN(timeCB) {

    Uns32 tvAddr;

    // obtain function arguments
    getArg(processor, object, 0, &tvAddr);

    // implement gettimeofday
    vmiosTimeBuf timeBuf = {0};
    vmiosGetTimeOfDay(processor, &timeBuf);

    // write back results
    vmirtRegWrite(processor, object->result, &timeBuf.sec);
}
```

Notes and Restrictions

1. It is the caller's responsibility to ensure that the buffer is large enough to hold the register value.
2. In VMI versions prior to 5.0.0, this function was called `vmiosRegWrite` and prototyped in file `vmiOSLib.h`. It has been moved to `vmiRt.h` because it is of more general applicability than semihosting libraries alone.

QuantumLeap Semantics

Non-self-synchronizing

22.8 vmirtGetCurrentMode

Prototype

```
vmiModeInfoCP vmirtGetCurrentMode(vmiProcessorP processor);
```

Description

Given a processor, this function returns the *current* defined mode for that processor. The mode is described using a structure defined in `vmiDbg.h` as:

```
typedef struct vmiModeInfoS {
    const char *name;          // mode name
    Uns32       code;          // model-specific mode code
    const char *description;   // description string
} vmiModeInfo;
```

Notes and Restrictions

1. Current mode will only be reported for processors that define the `getModeCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetCurrentMode` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.9 vmirtGetNextMode

Prototype

```
vmiModeInfoCP vmirtGetNextMode(
    vmiProcessorP processor,
    vmiModeInfoCP previous
);
```

Description

Given a processor and a `vmiModeInfoCP` structure implementing a mode in that processor, this function returns the *next* defined mode for that processor. The function returns `NULL` if there are no more mode definitions.

Typically, `vmirtGetMode` will be used in the constructor of a semi-hosting library to initialize pointers used to access those modes.

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiModeInfoCP mode = NULL;
    Uns32         i    = 0;

    // count all defined modes
    while((mode=vmirtGetNextMode(processor, mode))) {
        object->modeNum++;
    }

    // allocate array for mode handles
    object->modes = STYPE_CALLOC_N(vmiModeInfoCP, object->modeNum);

    // fill array of modes for later use
    while((mode=vmirtGetNextMode(processor, mode))) {
        object->modes[i++] = mode;
    }

    . . . etc . . .
  }
```

Notes and Restrictions

1. Modes can only be iterated for processors that define the `modeInfoCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextMode` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.10 vmirtGetCurrentException

Prototype

```
vmiExceptionInfoCP vmirtGetCurrentException(vmiProcessorP processor);
```

Description

Given a processor, this function returns the *current* defined exception for that processor. The exception is described using a structure defined in `vmiDbg.h` as:

```
typedef struct vmiExceptionInfoS {
    const char *name;          // exception name
    Uns32       code;          // model-specific exception code
    const char *description;   // description string
} vmiExceptionInfo;
```

Notes and Restrictions

1. Current exception will only be reported for processors that define the `getExceptionCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetCurrentException` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.11 vmirtGetNextException

Prototype

```
vmiExceptionInfoCP vmirtGetNextException(
    vmiProcessorP      processor,
    vmiExceptionInfoCP previous
);
```

Description

Given a processor and a `vmiExceptionInfoCP` structure implementing an exception in that processor, this function returns the *next* defined exception for that processor. The function returns `NULL` if there are no more exception definitions.

Typically, `vmirtGetException` will be used in the constructor of a semi-hosting library to initialize pointers used to access those exceptions.

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiExceptionInfoCP exception = NULL;
    Uns32              i         = 0;

    // count all defined exceptions
    while((exception=vmirtGetNextMode(processor, exception))) {
        object->exceptionNum++;
    }

    // allocate array for mode handles
    object->exceptions = STYPE_CALLOC_N(vmiExceptionInfoCP, object-> exceptionNum);

    // fill array of exceptions for later use
    while((exception=vmirtGetNextException(processor, exception))) {
        object->exceptions[i++] = exception;
    }

    . . . etc . . .
  }
```

Notes and Restrictions

1. Exceptions can only be iterated for processors that define the `exceptionInfoCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextException` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.12 vmirtGetBusPortByName

Prototype

```
vmiBusPortP vmirtGetBusPortByName(
    vmiProcessorP  processor,
    const char    *name
);
```

Description

This function returns a pointer to a `vmiBusPort` structure implementing the named port. See section 4.1 for more information about bus port structures.

If a port with the specified name is not found, a null pointer is returned.

Typically, `vmirtGetBusPortByName` will be used in the constructor of a semi-hosting library to enable access to information associated with that port (for example, the memory domain).

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiBusPortP mpCorePort   = vmirtGetBusPortByName(processor, "mpCoreDomain");
    memDomainP  mpCoreDomain = mpCorePort->domain;

    vmirtAddReadCallback (mpCoreDomain, 0, 0, -1, monitorRead,  NULL);
    vmirtAddWriteCallback(mpCoreDomain, 0, 0, -1, monitorWrite, NULL);

    . . . etc . . .
  }
```

Notes and Restrictions

1. Bus ports can only be found for processors that define the `busPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetBusPortByName` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.13 vmirtGetNextBusPort

Prototype

```
vmiBusPortP vmirtGetNextBusPort(
    vmiProcessorP  processor,
    vmiBusPortP    previous
);
```

Description

Given a processor and a `vmiBusPortP` structure defining a bus port in that processor, this function returns the *next* defined bus port for that processor. The function returns `NULL` if there are no more bus port definitions. See section 4.1 for more information about bus port structures.

Typically, `vmirtGetNextBusPort` will be used in the constructor of a semi-hosting library to initialize pointers used to access those bus ports.

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiBusPortP port = NULL;
    Uns32       i    = 0;

    // count all defined bus ports
    while((port=vmirtGetNextBusPort(processor, port))) {
        object->portNum++;
    }

    // allocate array for port handles
    object->ports = STYPE_CALLOC_N(vmiBusPortP, object->portNum);

    // fill array of ports for later use
    while((port=vmirtGetNextBusPort(processor, port))) {
        object->ports[i++] = port;
    }

    . . . etc . . .
 }
```

Notes and Restrictions

1. Bus ports can only be iterated for processors that define the `busPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextBusPort` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.14 vmirtGetNetPortByName

Prototype

```
vmiNetPortP vmirtGetNetPortByName(
    vmiProcessorP  processor,
    const char    *name
);
```

Description

This function returns a pointer to a `vmiNetPort` structure implementing the named port. See section 4.3 for more information about net port structures.

If a port with the specified name is not found, a null pointer is returned.

Typically, `vmirtGetNetPortByName` will be used in the constructor of a semi-hosting library to enable access to information associated with that port (for example, the description).

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiNetPortP port = vmirtGetNetPortByName(processor, "int0");

    if(port) {
        vmiPrintf("found net port %s (%s)\n", port->name, port->description);
    }

    . . . etc . . .
 }
```

Notes and Restrictions

1. Net ports can only be found for processors that define the `netPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNetPortByName` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.15 vmirtGetNextNetPort

Prototype

```
vmiNetPortP vmirtGetNextBusPort(
    vmiProcessorP  processor,
    vmiNetPortP    previous
);
```

Description

Given a processor and a `vmiNetPortP` structure defining a net port in that processor, this function returns the *next* defined net port for that processor. The function returns `NULL` if there are no more net port definitions. See section 4.3 for more information about net port structures.

Typically, `vmirtGetNextNetPort` will be used in the constructor of a semi-hosting library to initialize pointers used to access those net ports.

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiNetPortP port = NULL;
    Uns32       i    = 0;

    // count all defined net ports
    while((port=vmirtGetNextNetPort(processor, port))) {
        object->portNum++;
    }

    // allocate array for port handles
    object->ports = STYPE_CALLOC_N(vmiNetPortP, object->portNum);

    // fill array of ports for later use
    while((port=vmirtGetNextNetPort(processor, port))) {
        object->ports[i++] = port;
    }

    . . . etc . . .
 }
```

Notes and Restrictions

1. Net ports can only be iterated for processors that define the `netPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextNetPort` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.16 vmirtGetFifoPortByName

Prototype

```
vmiFifoPortP vmirtGetFifoPortByName(
    vmiProcessorP  processor,
    const char    *name
);
```

Description

This function returns a pointer to a `vmiFifoPort` structure implementing the named port. See section 4.4 for more information about FIFO port structures.

If a port with the specified name is not found, a null pointer is returned.

Typically, `vmirtGetFifoPortByName` will be used in the constructor of a semi-hosting library to enable access to information associated with that port (for example, the description).

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiFifoPortP port = vmirtGetFifoPortByName(processor, "fifoOut");

    if(port) {
        vmiPrintf("found FIFO port %s (%s)\n", port->name, port->description);
    }

    . . . etc . . .
 }
```

Notes and Restrictions

1. FIFO ports can only be found for processors that define the `fifoPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetFifoPortByName` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

22.17 vmirtGetNextFifoPort

Prototype

```
vmiFifoPortP vmirtGetNextFifoPort(
    vmiProcessorP  processor,
    vmiFifoPortP   previous
);
```

Description

Given a processor and a `vmiFifoPortP` structure defining a FIFO port in that processor, this function returns the *next* defined FIFO port for that processor. The function returns `NULL` if there are no more FIFO port definitions. See section 4.4 for more information about FIFO port structures.

Typically, `vmirtGetNextFifoPort` will be used in the constructor of a semi-hosting library to initialize pointers used to access those FIFO ports.

Example

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

static VMIOS_CONSTRUCTOR_FN(constructor) {

    vmiFifoPortP port = NULL;
    Uns32        i    = 0;

    // count all defined FIFO ports
    while((port=vmirtGetNextFifoPort(processor, port))) {
        object->portNum++;
    }

    // allocate array for port handles
    object->ports = STYPE_CALLOC_N(vmiFifoPortP, object->portNum);

    // fill array of ports for later use
    while((port=vmirtGetNextFifoPort(processor, port))) {
        object->ports[i++] = port;
    }

    . . . etc . . .
 }
```

Notes and Restrictions

1. FIFO ports can only be iterated for processors that define the `fifoPortSpecsCB` callback in the processor attributes structure. If this function is not defined, `vmirtGetNextFifoPort` returns `NULL`.

QuantumLeap Semantics

Non-self-synchronizing

23 Semihost and Debugger Integration Support Functions

This section describes functions that are used when integrating OVP processor models with client debug or analysis environments, and also when implementing semihost libraries.

23.1 vmirtDisassemble

Prototype

```
const char *vmirtDisassemble(
    vmiProcessorP  processor,
    Addr           simPC,
    vmiDisassAttrs attrs
);
```

Description

Given a processor and a simulated address, this function returns the disassembled instruction at that address, as interpreted by the processor in its current state. The result is valid only until the next call to `vmirtDisassemble`.

Note that for processors with distinct instruction set states (for example ARM/Thumb or MIPS/microMIPS) the disassembly is performed in the *current processor state*. There is no guarantee that the instruction at the given address is a valid instruction in that state.

The `attrs` bitfield argument is used to indicate the type of disassembly required, defined by type `vmiDisassAttrs` in file `vmiTypes.h` as follows:

```
typedef enum vmiDisassAttrsE {
    DSA_NORMAL   = 0x00000000,  // normal disassembly
    DSA_UNCOOKED = 0x00000001,  // model-specific uncooked format
    DSA_BASE     = 0x00000002,  // use base model disassembly (not intercept)
    DSA_MODEL    = 0x80000000,  // model-specific mask
} vmiDisassAttrs;
```

A value of `DSA_NORMAL` indicates that disassembly in the normal display format is required.

A value of `DSA_UNCOOKED` indicates that disassembly in *uncooked* format is required. Uncooked format can be used to obtain information about the instruction that is easier to parse by downstream tools. The exact format of uncooked disassembly is model-specific.

A value of `DSA_BASE` indicates that disassembly should be performed using the disassembly callback associated with the processor model only. If this bit is absent, then disassembly functions associated with intercept libraries will be used in preference, and the processor model disassembly function will be used only if no intercept library disassembly function generates a non-`NULL` result.

A value of `DSA_MODEL` combined with any other numeric value by bitwise-or indicates that some processor-specific disassembly format is required.

Example

The following example is extracted from the standard ARM AngelTrap semihost library. It shows how the result of `vmirtDisassemble` is used to detect `SVC` and `HLT` instructions that must be handled by the semihost library.

```
static Bool matchOpcode(const char *disass, const char *opcode) {

    // get opcode length
    Uns32 opBytes = strlen(opcode);

    // does opcode match value in disassembly?
    return !strncmp(disass, opcode, opBytes) && (disass[opBytes]==' ');
}

static VMIOS_MORPH_FN(armOSOperation) {

    // disassemble this instruction
    const char *disass = vmirtDisassemble(processor, thisPC, DSA_UNCOOKED);

    if(matchOpcode(disass, "svc") || matchOpcode(disass, "bkpt")) {

        // svc/bkpt instruction: get condition and constant
        armCondition cond      = parseCond(disass);
        Uns32        arg       = parseConst(disass);
        Bool         thumbMode = parseThumbMode(disass);

        // ARM Angel semi-hosting calls are indicated by:
        //  SVC  0x123456 on Cortex-A and -R in ARM mode or by
        //  SVC  0xab     on Cortex-A and -R in Thumb mode or by
        //  BKPT 0xab     on Cortex-M (which only supports Thumb mode)
        Uns32 angelCode = thumbMode ? AngelSVC_THUMB : AngelSVC_ARM;

        // only intercept if code is angel code
        if(arg == angelCode) {
            *context  = "armAngelSyncTrap";
            *opaque   = True;
            *userData = (void *)cond;
            return armAngelSyncTrap;
        }

    } else if(matchOpcode(disass, "hlt")) {

        // hlt instruction: get constant
        Uns32 arg = parseConst(disass);

        // only intercept if code is angel code
        if(arg == AngelHLT) {
            *context  = "armAngelSyncTrap";
            *opaque   = True;
            *userData = (void *)ARM_C_AL;
            return armAngelSyncTrap;
        }

    } else if(matchOpcode(disass, "b")) {

        // get condition and target
        armCondition cond = parseCond(disass);
        Uns32        arg  = parseTarget(disass);

        // intercept branch-to-self
        if(arg == thisPC) {
            *context  = "armAngelSyncTrap";
            *opaque   = True;
            *userData = (void *)cond;
            return armAngelBTS;
        }
    }

    return 0;
}
```

Notes and Restrictions

None.

QuantumLeap Semantics
Synchronizing

24 Debugger Integration Support Functions

This section describes functions that are used when integrating OVP processor models with client debug or analysis environments. These functions are available only with the Imperas Professional Tools.

24.1 vmirtEvaluateGDBExpression

Prototype

```
const char *vmirtEvaluateGDBExpression(
    vmiProcessorP processor,
    const char   *expression,
    Bool          usePhysicalDomain,
    Bool          quiet
);
```

Description

This function evaluates an expression using a GDB attached to the given processor. The result is returned as a string, which remains valid only until the next call to `vmirtEvaluateGDBExpression`. The argument `usePhysicalDomain` specifies whether any memory accesses implied by the expression use physical addresses (`True`) or virtual addresses (`False`). The quiet argument causes GDB errors resulting from evaluation of the expression to be suppressed.

Example

Contact Imperas for usage examples and details of Professional Tools.

Notes and Restrictions

1. This function is only available with Imperas Professional Tools.

QuantumLeap Semantics

Synchronizing

24.2 vmirtEvaluateCodeLocation

Prototype

```
Bool vmirtEvaluateCodeLocation(
    vmiProcessorP processor,
    const char   *expression,
    Addr         *address,
    char        **sourceFile,
    unsigned     *sourceLine
);
```

Description

Given a processor and an expression, this function evaluates the expression using a GDB attached to the processor to yield an address, source file and source line result. The expression should be of a form suitable for use with the GDB `break` command. The purpose of this function is to enable client debuggers to place breakpoints appropriately.

Example

Contact Imperas for usage examples and details of Professional Tools.

Notes and Restrictions

1. This function is only available with Imperas Professional Tools.

QuantumLeap Semantics

Synchronizing

24.3 vmirtInstructionBytes

Prototype

```
Uns32 vmirtInstructionBytes(vmiProcessorP processor, Addr simPC);
```

Description

Given a processor and a simulated address, this function returns the size in bytes of the instruction at that address, as interpreted by the processor in its current state.

Note that for processors with distinct instruction set states (for example ARM/Thumb or MIPS/microMIPS) the result is determined in the *current processor state*. There is no guarantee that the instruction at the given address is a valid instruction in that state.

Example

Contact Imperas for usage examples and details of Professional Tools.

Notes and Restrictions

1. This function is only available with Imperas Professional Tools.

QuantumLeap Semantics

Synchronizing

24.4 vmirtAddRegisterWatchCallback

Prototype

```
vmiWatchHandleP vmirtAddRegisterWatchCallback(
    vmiProcessorP processor,
    vmiRegInfoCP  reg,
    vmiRegValueFn valueCB,
    void         *userData
);
```

Description

Given a processor and a `vmiRegInfoCP` descriptor for a register in that processor, this function inserts a watch point on that register. After any instruction in which the register changes, the notifier function `valueCB` is called. The notifier is prototyped in `vmiTypes.h` as:

```
#define VMI_REG_VALUE_FN(_NAME) void _NAME( \
   vmiProcessorP processor,    \
   vmiRegInfoCP  reg,          \
   Addr          simPC,        \
   void         *oldValue,     \
   void         *newValue,     \
   void         *userData      \
)
typedef VMI_REG_VALUE_FN((*vmiRegValueFn));
```

As well as the processor and register, the notifier is passed the following arguments:

1. The simulated address of the instruction that changed the register.
2. A pointer to the previous register value.
3. A pointer to the new register value.
4. A `userData` pointer, passed to `vmirtAddRegisterCallback` when the watchpoint was created.

Typically, this function is used to create customized trace output.

Example

Contact Imperas for usage examples and details of Professional Tools.

Notes and Restrictions

1. This function is only available with Imperas Professional Tools.

QuantumLeap Semantics

Synchronizing

24.5 vmirtDeleteRegisterWatchCallback

Prototype

```
void vmirtDeleteRegisterWatchCallback(vmiWatchHandleP handle);
```

Description

This function deletes a watch point previously created by `vmirtAddRegisterCallback`.

Example

Contact Imperas for usage examples and details of Professional Tools.

Notes and Restrictions

1. This function is only available with Imperas Professional Tools.

QuantumLeap Semantics

Synchronizing

25 Shared Object Access

It is sometimes useful to be able to load shared objects (or dynamically-linked libraries on Windows) into processor models. As an example, a processor model might allow user-defined instructions to be provided using a separate shared object.

This section describes functions for opening and closing shared objects, and for determining a symbol address in the loaded object. There is also a utility function to obtain any error generated by the object loader.

25.1 vmirtDLOpen

Prototype

```
vmiDLHandleP vmirtDLOpen(const char *fileName);
```

Description

This function attempts to open a shared object or dynamically-linked library of the passed name. If successful, the function returns an object of type `vmiDLHandleP`, which can be used subsequently to find symbol addresses in the loaded object (see function `vmirtDLSymbol`). If the load is unsuccessful, the function returns `NULL`; function `vmirtDLError` may be called to get further information about the cause of failure.

Example

```
#include “vmi/vmiRt.h”

Bool cpuExtInstall(cpuP cpu) {

    const char  *extFileAttr = "extFileName";
    const char  *configFn    = "cpuConfig";
    vmiDLHandleP handle;
    extConfigFn  configCB;

    // get the name of any extension file
    const char *extFileName = vmirtPlatformStringAttribute(
        (vmiProcessorP)cpu, extFileAttr
    );

    if(!extFileName) {

        // no extension file given

    } else if(!(handle=vmirtDLOpen(extFileName))) {

        // bad extension file
        vmiMessage("E", PREFIX "EXT_BFN",
            "Error loading extension file %s (%s)",
            extFileName, vmirtDLError()
        );

    } else if(!(configCB=vmirtDLSymbol(handle, configFn))) {

        // configuration function not found
        vmiMessage("E", PREFIX "EXT_SNF",
            "Configuration function %s not found in extension file %s",
            configFn, extFileName
        );

    } else {

        // .s0/.dll successfully loaded
        cpu->handle = handle;
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics
Synchronizing

25.2 vmirtDLClose

Prototype

```
Int32 vmirtDLClose(vmiDLHandleP handle);
```

Description

This function closes a shared object of dynamically-linked library previously opened by `vmirtDLOpen`. This is usually not required except in special circumstances where a library must be unloaded while the program is executing.

Example

```
#include “vmi/vmiRt.h”

Bool cpuExtUninstall(cpuP cpu) {

    if(cpu->handle) {
        vmirtDLClose(cpu->handle);
        cpu->handle = 0;
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing

25.3 vmirtDLSymbol

Prototype

```
void *vmirtDLSymbol(vmiDLHandleP handle, const char *symbol);
```

Description

This function searches for the passed symbol in a previously-opened shared object of dynamically linked library. If the symbol is found, the function returns its address; if the symbol is not found, the function returns `NULL`.

Example

```
#include “vmi/vmiRt.h”

Bool cpuExtInstall(cpuP cpu) {

    const char  *extFileAttr = "extFileName";
    const char  *configFn    = "cpuConfig";
    vmiDLHandleP handle;
    extConfigFn  configCB;

    // get the name of any extension file
    const char *extFileName = vmirtPlatformStringAttribute(
        (vmiProcessorP)cpu, extFileAttr
    );

    if(!extFileName) {

        // no extension file given

    } else if(!(handle=vmirtDLOpen(extFileName))) {

        // bad extension file
        vmiMessage("E", PREFIX "EXT_BFN",
            "Error loading extension file %s (%s)",
            extFileName, vmirtDLError()
        );

    } else if(!(configCB=vmirtDLSymbol(handle, configFn))) {

        // configuration function not found
        vmiMessage("E", PREFIX "EXT_SNF",
            "Configuration function %s not found in extension file %s",
            configFn, extFileName
        );

    } else {

        // .s0/.dll successfully loaded
        cpu->handle = handle;
    }
}
```

Notes and Restrictions

1. It is the client program’s responsibility to ensure that the returned symbol is of an appropriate type if required – for example, there is no check that a symbol that is to be called as a function is in fact a function pointer.

QuantumLeap Semantics
Synchronizing

25.4 vmirtDLError

Prototype

```
const char *vmirtDLError(void);
```

Description

This function returns any error message from the previous call to `vmirtDLOpen`, `vmirtDLLClose` or `vmirtDLSymbol`. If there was no error, `NULL` is returned

Example

```
#include “vmi/vmiRt.h”

Bool cpuExtInstall(cpuP cpu) {

    const char  *extFileAttr = "extFileName";
    const char  *configFn    = "cpuConfig";
    vmiDLHandleP handle;
    extConfigFn  configCB;

    // get the name of any extension file
    const char *extFileName = vmirtPlatformStringAttribute(
        (vmiProcessorP)cpu, extFileAttr
    );

    if(!extFileName) {

        // no extension file given

    } else if(!(handle=vmirtDLOpen(extFileName))) {

        // bad extension file
        vmiMessage("E", PREFIX "EXT_BFN",
            "Error loading extension file %s (%s)",
            extFileName, vmirtDLError()
        );

    } else if(!(configCB=vmirtDLSymbol(handle, configFn))) {

        // configuration function not found
        vmiMessage("E", PREFIX "EXT_SNF",
            "Configuration function %s not found in extension file %s",
            configFn, extFileName
        );

    } else {

        // .s0/.dll successfully loaded
        cpu->handle = handle;
    }
}
```

Notes and Restrictions

None.

QuantumLeap Semantics

Synchronizing