Table of Contents

Monitor Module Developer's Guide

1. Introduction

Simulation monitoring provides a method for a user-developed module to gain knowledge of events and states during a simulation. This knowledge can then be used to calculate performance timing information, power utilization information, or other data useful to the user.

A Monitor module is a dynamically loaded library created by the user. When instructed (using the '-p' or '--Monitor command line switch) to do so, SimNow will load this module into the framework, initialize it, and provide information to it about any simulation activity that occurs.

SimNow provides a sample monitor module, as well as the necessary .h files to allow users to create their own monitor module. A user monitor module contains a single global exported function that SimNow will use to identify the module and create an instance of the monitor class, as well as the definition and implementation of a monitor class. The monitor class object then interacts with SimNow for all monitoring activities.

SimNow defines a class, CMonitorConsumer that the user monitor class must inherit. This class has the necessary interfaces and functionality to communicate with SimNow. The user monitor class will then override some routines, and call others, as necessary for its design.

2. Sample Source Files

Sample source files are provided in the "monitor" subdirectory of the "devel/monitor" subdirectory in the public release.

3. Building an Monitor Module

To develop Monitor Module, the developer must use an AMD64 system running either 64-bit Windows® or 64-bit Linux operating system. If you are running 64-bit Windows, you will need Visual Studio 2005 installed. Otherwise, you will need the gcc toolchain installed.

3.1 Building an Monitor Module – Linux

Building your monitor module on Linux is a simple process. Use the makefile that is included in Monitor module directory. Follow steps as below:

- Make sure you are in " devel/monitor/example" directory
- Type "make" to build the libmonitor.so shared library.
- When build is completed, libmonitor.so is automatically copied to the directory where SimNow executable is.
- Type "./simnow -p" to invoke SimNow and load this monitor module example.

3.2 Building an Monitor Module - Windows®

To build your Monitor Module on Windows the developer must have pre-requisite as below:

- AMD Athlon™ 64, AMD Phenom™, AMD Turion™ or AMD Opteron™ processor.
- Windows XP x64 Edition or Windows Server 2003 x64 Edition for AMD64.
- Microsoft Visual Studio 2005 Professional Edition
    - Microsoft Visual C++ 2005.
    - x64 Solution Platform (x64 Compilers and Tools) installed.

See SimNow User's Manual for more details.

from "devel\monitor\example" directory follow steps as below:

- First open the monitor-release.sln solution file
- Click on 'Build' to build this monitor module example project.
- When build is completed, monitor.acm is automatically copied to the directory where SimNow executable is.
- Open a command prompt, type "simnow -p" to invoke SimNow and load this monitor module example.

4. Monitor Module Interface

The monitor module must export one globally defined function that will be called by SimNow during module loading. This function is used by SimNow to identify a monitor module, and to create an instance of the monitor class. SimNow will use this instance for all communication with the monitor module.

The "pModuleName" argument supplies the fully qualified file name of the module being loaded, and the "pArgs" argument supplies any additional command line arguments supplied by the user with the --aarg command line switch.

```
extern "C" DLLEXPORT CMonitorConsumer
*SimNowGetMonitorInterface(const char *pModuleName, const char
*pArgs)
{
      printf("Monitor: Being loaded from %s. Args = <%s>\n",
pModuleName, pArgs);
      return(new CMonitor(pModuleName, pArgs));
}
```

The object "CMonitor" returned by the above function is a user written class that inherits from the CMonitorConsumer class provided by SimNow. The following details will describe this object, and the functionality it provides:

The following routines are called by the framework to initiate and terminate simulation. If overridden, be sure that these base class routines get called

```
virtual bool Initialize(IMonitorProvider *pProvider);
virtual void Terminate();
```

4.1. Routines that can be called from the user module

4.1.1. EnableEventCollection

Description:

EnableEventCollection - Create a new event queue on the given machine

Parameters:

nMachineID = Logical ID of the machine to enable event collection on.

Prototype:

virtual bool EnableEventCollection(int nMachineID);

4.1.2. DeviceCharacteristics

Description:

DeviceCharacteristics – Fills in a structure that will contain various characteristics of the device requested.

Parameters:
nMachineID = Logical ID of the machine to query
nDeviceID = Logical device ID of the device to query
pBuffer = Pointer to a DEVICECHARACTERISTICS derived buffer to contain the result
pSize = Pointer to an int size of the buffer. On return, has the size of the returned buffer

Note: You can initialize *pSize to 0, and on return, (even if the return value is <false>) if *pSize is > 0, then you know that a particular device has responded, and *pSize is the size of the structure buffer needed to contain the resulting structure.

The actually buffer pointer passed should be one of the descendants of a DEVICECHARACTERISTICS structure, dependent on the type of device being queried.

Prototype:
Virtual bool DeviceCharacteristics(int nMachineID, UINT64 nDeviceID, DEVICECHARACTERISTICS *pBuffer, int *pSize);

4.1.3. GetEventQueue

Description:
GetEventQueue - Returns a pointer to the event queue for the given machine ID

Parameters:
nMachineID = Logical ID of the machine whose queue we want.
Returns a pointer to the first EVENTQUEUEINFO block if successful.
NULL if an error occurred.
Note: The caller is responsible for walking the list to find the specific queue of interest.

Prototype:
virtual EVENTQUEUEINFO *GetEventQueue(int nMachineID);

4.1.4. UpdateEventOptions

Description:
UpdateEventOptions - Update the event collection options for the given event to the given bits

Parameters:
nMachineID = Logical ID of the machine to create the event queue for
nDeviceID = Logical device ID of the device to query
nEventID = 32 bit ID of the event being modified

nEventOptions= Bitmask of events that should be placed in this queue

Returns <true> if ok, <false> if an error occurs.

Note: Setting an "options" value of all zeros is defined as turning the event off - Any other setting controls the various options available for this event (event specific)

Prototype:
virtual bool UpdateEventOptions(int nMachineID, UINT64 nDeviceID, UINT32 nEventID, UINT16 nEventOptions);

4.1.5. RequestCallbackDelayed

Description:
RequestCallbackDelayed - Request a callback when a certain amount of time has elapsed

Parameters:
nMachineID = Logical ID of the machine to create the event on
nTimeDelay = Amount of time (in nanoseconds) in the future to schedule the delay

Returns <true> if ok, <false> if an error occurs

Note: The time delay is expressed in nanoseconds of simulated time. A value of ((UINT64)-1) cancels any previous callback.

Prototype:
virtual bool RequestCallbackDelayed(int nMachineID, UINT64 nTimeDelay);

4.1.6. CurrentSimulationTime

Description:
CurrentSimulationTime - Returns the current simulation time for the indicated logical machine

Parameters:
nMachineID = Logical ID of the machine to get the time for

Returns ((UINT64) -1) if an error occurred, otherwise it returns the simulation time, in nanoseconds

Prototype:
virtual UINT64 CurrentSimulationTime(int nMachineID);

4.1.7. ExactCurrentSimulationTime

Description:

ExactCurrentSimulationTime - Returns the current simulation time for the indicated logical machine

Parameters:

nMachineID = Logical ID of the machine to get the time for

Returns ((UINT64) -1) if an error occurred, otherwise it returns the simulation time, in nanoseconds

NOTE: Only use this function if you are sure you are executing out of the main execution thread for the indicated nMachineID. If there is any chance that you are not, you must use CurrentSimulationTime() instead.

Prototype:

virtual UINT64 ExactCurrentSimulationTime(int nMachineID);

4.1.8. ProcessorStatistics

Description:

ProcessorStatistics - Returns information about the given processor simulation

Parameters:

nMachineID	= Logical ID of the machine to get the information for
nCpuID	= Logical ID (starting at 0) for which core to get information for
pInfo	= Pointer to the PROCESSORSTATISTICS structure to fill in
nCount	= Size of the PROCESSORSTATISTICS structure, in bytes

Returns <true> if ok, <false> if the machine or cpu does not exist, or the structure given is too short to contain the data

Prototype:

virtual bool ProcessorStatistics(int nMachineID, int nCpuID, PROCESSORSTATISTICS *pInfo, int nCount);

4.1.9. InstructionCount

Description:

InstructionCount - Return the # of instructions the processor executing.

Parameters:

nMachineID = Logical ID of the machine to get the information for

nCpuID = Logical ID (starting at 0) for which core to get information for

Note: This returns the same value that is available in the processor statistics structure

Prototype:
virtual UINT64 InstructionCount(int nMachineID, int nCpuID);

4.1.10. IdleCount

Description:
IdleCount - Return the # of cycles spent "idle" by the given processor

Parameters:
nMachineID = Logical ID of the machine to get the information for
nCpuID = Logical ID (starting at 0) for which core to get information for

Note: This returns a value that is the aggregate of the various idle counts available in the processor statistics structure (i.e. I/O count, HLT count, etc.)

Prototype:
virtual UINT64 IdleCount(int nMachineID, int nCpuID);

4.1.11. SetProcessorIPC

Description:
SetProcessorIPC - Set the IPC used by one or more processors

Parameters:
nMachindID = Logical ID of the machine whose CPUs are to be adjusted
nCpuID = Logical ID (starting at 0) of the CPU to modify. Use -1 to change all cpus
nInstr = # of instructions executed per # of cycles given
nCycles = # of cycles to use to execute <nInstr> instructions

Prototype:
virtual bool SetProcessorIPC(int nMachineID, int nCpuID, UINT64 nInstr, UINT64 nCycles);

4.1.12. QuantumEndConfig

Description:
QuantumEndConfig - Set the parameters for QuantumEnd callback interval

Parameters:
nMachindID = Logical ID of the machine that set the notification interval when processors are synchronized.
nTime = Minimum time (in nanoseconds) for notification inverval
nError = Allowable error (in nanoseconds) such that nTime + nError is the maximum interval

Prototype:
virtual bool QuantumEndConfig (int nMachineID, UINT64 nTime, UINT64 nError);

Note:
Setting nTime = 0 will result in a notification on every sync point.
Setting nError = 0 will result in a notification every nTime ns.
Setting nError < SyncQuantum may introduce new sync points.
Setting nError >= SyncQuantum will not introduce new sync points.
Setting nTime = 0, nError = 0 will disable all notifications.

4.1.13. MonitorGetPState

Description:
MonitorGetPState - Returns the value of P-State of the processor

Parameters:
nMachindID = Logical ID of the machine whose CPUs are to be adjusted
nCpuID = Logical ID (starting at 0) of the CPU.

Prototype:
virtual UINT64 MonitorGetPState(int nMachineID, int nCpuID);

4.1.14. MonitorSetPState

Description:
MonitorSetPState - Set the P-State of the particular processor

Parameters:
nMachindID = Logical ID of the machine whose CPUs are to be adjusted
nCpuID = Logical ID (starting at 0) of the CPU

nPState =desired P-State

Prototype:

virtual bool MonitorSetPState(int nMachineID, int nCpuID, int nPState)

4.2. Routines that should be implemented by the user module

4.2.1. InitalizeMachine

Description:
InitalizeMachine - A new SimNow machine is being created

Parameters:
nMachineID = Logical ID of the machine being created

Prototype:
virtual void InitializeMachine(int nMachineID) = 0;

4.2.2. TerminateMachine

Description:
TerminateMachine - An existing SimNow machine is being terminated

Parameters:
nMachineID = Logical ID of the machine being terminated
pQueueInfo = Pointer to the first event queue info block for this machine (or NULL if no queue)

NOTE: Event queues on this machine are still valid when this routine is called. They are released automatically after this function returns.

NOTE: The logical machine has been removed from the machine queue, so calls that require a machine ID (such as GetEventQueue) will fail.

Prototype:
virtual void TerminateMachine(int nMachineID, EVENTQUEUEINFO *pQueueInfo) = 0;

4.2.3. CodeInjection

Description:
CodeInjection - Allow for code injection into the translation cache

Parameters:
nMachineID = Logical machine ID of the machine translating the code
nDeviceID = Logical device ID of the device to query

pInjector = Pointer to ICodeInjector that can be used to inject code

Note: Neither CPU ID nor DEVICE ID is provided at this point.

Prototype:

virtual void CodeInjection(int nMachineID, UINT64 nDeviceID, CCodeInjector *pInjector) = 0;

4.2.4. MemoryInjection

Description:

MemoryInjection - Allow for memory events logging to the queue

Parameters:

nMachineID = Logical machine ID of the machine translating the code
nDeviceID = Logical device ID of the device to query
pInjector = Pointer to ICodeInjector that can be used to log memory

Prototype:

virtual void MemoryInjection(int nMachineID, UINT64 nDeviceID, CCodeInjector *pInjector) = 0;

4.2.5. ExcInjection

Description:

ExcInjection - Allow for exception events logging to the queue

Parameters:

nMachineID = Logical machine ID of the machine translating the code
nDeviceID = Logical device ID of the device to query
pInjector = Pointer to ICodeInjector that can be used to log exceptions

Prototype:

virtual void ExcInjection(int nMachineID, UINT64 nDeviceID, CCodeInjector *pInjector) = 0;

4.2.6. HeartBeatInjection

Description:

HeartBeatInjection - Allow for heart beat events being placed in the queue

Parameters:

nMachineID = Logical machine ID of the machine translating the code
nDeviceID = Logical device ID of the device to query
pInjector = Pointer to ICodeInjector that can be used to log heart beat events

Prototype:
virtual void HeartBeatInjection(int nMachineID, UINT64 nDeviceID, CCodeInjector *pInjector) = 0;

4.2.7. BranchInjection

Description:
BranchInjection - Allow for branch events being placed in the queue. Note: only LWP branch instructions feature is supported.

Parameters:
nMachineID = Logical machine ID of the machine translating the code
nDeviceID = Logical device ID of the device to query
pInjector = Pointer to ICodeInjector that can be used to log heart beat events

Prototype:
virtual void BranchInjection(int nMachineID, UINT64 nDeviceID, CCodeInjector *pInjector) = 0;

4.2.8. CodeTranslation

Description:
CodeTranslation - Hook called every time a new instruction is being translated for later execution

Parameters:
nMachineID = Logical machine ID of the machine translating the code
nDeviceID = Logical device ID of the device to query
pInjector = Pointer to ICodeInjector that can be used as needed
pTranslateStruc = Pointer to CODETRANSLATESTRUCT containing information about this instruction, and allowing the hook to tag the instruction for later tagged callout

Prototype:
virtual void CodeTranslation(int nMachineID, UINT64 nDeviceID, CCodeInjector *pInjector, CODETRANSLATESTRUCT *pTranslateStruc) = 0;

4.2.9. TaggedExecution

Description:
TaggedExecution - Notify the monitor module that previously tagged instruction is executed.

Parameters:

nMachineID = Logical machine ID of the machine translating the code
nDeviceID = Logical device ID of the device to query
pInjector = Pointer to ICodeInjector that can be used as needed
pParam = Tag attached to this instruction

Prototype:
virtual void TaggedExecution(int nMachineID, UINT64 nDeviceID, CCodeInjector *pInjector, void *pParam) = 0;

Note: Calling pInjector->BuildInstructionInfo will not return matching INSTRUCTIONINFO with the instruction that is tagged.

4.2.10. RegisterEventQueue

Description:
RegisterEventQueue - An event queue is being registered by a device in the given logical machine

Parameters:
nMachineID = Logical ID of the machine owning the device that is registering the queue
pQueueInfo = Pointer to EVENTQUEUEINFO that is being registered

Note: The EVENTQUEUEINFO structure has already been linked into the chain of event queues for this logical machine

Prototype:
virtual void RegisterEventQueue(int nMachineID, EVENTQUEUEINFO *pQueueInfo) = 0;

4.2.11. UnregisterEventQueue

Description:
UnregisterEventQueue - An event queue is being unregistered by a device in the given logical machine

Parameters:
nMachineID = Logical ID of the machine owning the device that is un-registering the queue
pQueueInfo = Pointer to EVENTQUEUEINFO that is being registered

NOTE: The EVENTQUEUEINFO structure has not yet been unlinked from the chain of event queues for this logical machine

Prototype:

virtual void UnregisterEventQueue(int nMachineID, EVENTQUEUEINFO *pQueueInfo) = 0;

4.2.12. DeliverEventQueue

Description:

DeliverEventQueue - A specific event queue needs to be emptied.

Parameters:

nReason	=	Various DELIVERYREASON_xxx bit flags
nMachineID	=	Logical machine ID of the event queue that is full
nQueueID	=	Logical Queue ID of the event queue that is full
pQueueInfo	=	Pointer to the event queue info block for this queue

Note: The caller will empty the queue (and possibly reset the head and tail pointers) on return, so get everything you need out of the queue before returning.

Prototype:

virtual void DeliverEventQueue(UINT64 nReason, int nMachineID, EVENTQUEUEINFO *pQueueInfo) = 0;

4.2.13. TimeCallback

Description:

TimeCallback - Called after the time elapsed from an IMonitorProvider::RequestCallbackAtTime()

Parameters:

nMachineID	=	Logical machine ID of the machine whose callback time has expired

Prototype:

virtual void TimeCallback(int nMachineID) = 0;

4.2.14. RunControl

Description:

RunControl - A simulated machine is starting/stopping

Parameters:

nMachineID	=	Logical machine ID of the machine that is being started/stopped
bStarting	=	<true> if starting, <false> if stopping

NOTE: Options for any desired events need to be requested w/UpdateEventOptions() during a RunControl with bStarting = <true>. This is because the user may have removed or added a new device and that device will not have knowledge of any options that were sent previously.

Prototype:

virtual void RunControl(int nMachineID, bool bStarting) = 0;

4.2.15. ReturnNextCommand

Description:

ReturnNextCommand - Return the next ASCII command for the simulation shell

Parameters:

If requested by the '--AInp' argument on the command line, SimNow will call this routine instead of its normal console read routine to get input commands.

nMachineID = The machine ID to which this command will apply
psResponse = Pointer to response from previous command executed (initially a null string)
psCommand = Pointer to buffer to place ASCII command to execute
nBufferSize = # of bytes in the input buffer that can be used

Returns <true> to execute the command, and then repeat this call, or false to not execute the command, and return to normal console input functionality.

Prototype:

virtual bool ReturnNextCommand(int nMachineID, const char *psResponse, char *psCommand, int nBufferSize) = 0;

4.2.16. ExternalEventTiming

Description:

ExternalEventTiming - Allows a delay to be inserted before a device transfers data or responds to certain commands

Parameters:

This must be enabled by setting the EVENT_BLOCKDEV_EXTTIMING or EVENT_PACKETDEV_EXTTIMING option bits on the Read/Write events for the specific device you wish to control. Once enabled, the device will call this routine.

nMachineID = Logical machine ID of the machine the device belongs to
nDeviceID = The device ID (same as in the queue header for the device, if it has a queue)
eAccessType = The type of transfer/access the device is performing

pEventQueue = Pointer to the event queue for this device, or NULL if no event queue is being used. This allows the analysis module to see what the read/write event (which is always the last read/write in the queue) is in order to help determine the appropriate delay (i.e. block # and # of blocks)

*pDelay = Pointer to a UINT32. The initial value is the value the device would normally use. You can update this to be any "reasonable" non-zero value.

Prototype:

virtual void ExternalEventTiming(EVENTTIMINGINFO *pInfo) = 0;

4.2.17. QuantumEnd

Description:

QuantumEnd - The simulation has reached the end of a quantum

Parameters:

nMachineID = Logical machine ID of the machine whose quantum has ended

Prototype:

virtual void QuantumEnd(int nMachineID) = 0;

4.2.18. FrequencyCallBack

Description:

FrequencyCallBack - CPU clock frequency event call back

Parameters:

nMachineID = Logical machine ID of the machine
nDeviceID = DeviceID of the device
pInjector = Pointer to ICodeInjector
mhz = Frequency in MHz

Prototype:

virtual void FrequencyCallBack(int nMachineID, UINT64 nDeviceID, ICodeInjector *pInjector, UINT32 mhz) = 0;

4.2.19. GetProcessorFreq

Description:

GetProcessorFreq - Get the frequency of the particular processor

Parameters:

nMachineID = Logical machine ID of the machine

nDeviceID = Logical ID (starting at 0) of the CPU

Prototype:

virtual UINT32 GetProcessorFreq(int nMachineID, int nCpuID) = 0;