PCI Device Simulation for Device Driver Development

By Tom Fones, HTF Consulting tfones@htfconsulting.com

1 ABSTRACT

To this author's knowledge, no published work expounds the benefits or feasibility of software simulation of Peripheral Component Interface (PCI) devices.

In the 21st century, commercially viable PCI devices must be capable of plug-n-play, power management, web based enterprise management and scatter-gather DMA. A device simulation in software incorporating all of these features is feasible.

This article discusses the many benefits and uses of a PCI device simulator as well as how to produce one.

2 INTRODUCTION

In the last twenty-plus years the PCI bus and its derivatives have seen an explosion in adoption across all small computer platforms. There are countless PCI devices on the market. The PCI bus has been through numerous revisions for the sake of performance improvement and expansion of function.

There is a PCI Special Interest Group which has approximately one thousand members, primarily third-party hardware vendors who collaborate to evaluate and adopt standards for the different aspects of the PCI bus and interoperability with PCI devices.

To put it succinctly the PCI bus has critical mass. It will probably be around in some form for the life of the small computer.

There are several published papers in the public domain literature describing software simulations of the PCI bus itself. There **is** a description of a hardware simulator in a Windows device driver book Dekker and Newcomer. [Dekker & Newcomer, 1999]. This very admirable effort modeled a character-mode device on NT 4.0. Unfortunately it is obsolete for multiple reasons.

3 SIMULATION BENEFITS AND USAGE

A software simulation of a PCI device is most useful when the device is still being designed in the hardware lab, or when sufficient quantity is not available from manufacturing. It can save weeks or months of calendar time to develop the device software in parallel with hardware design and manufacure.

I was involved in a project a few years ago where a small company was having difficulty completing the design of their storage device and then getting quantities from the fabrication plant in Taiwan. We implemented a basic simulation of the target device, which allowed exercising the device management portion of the target device driver in user mode. Many man-months over several calendar months were dedicated to implementing the simulation and building testware around the device driver. Yet when the device(s) arrived we were exercising complete kernel-mode drivers for the first time on both Linux and Windows.

A good device simulator exercises plug-n-play and power management without a physical device. A good simulator can emulate hard to create error conditions: device failure, device power failure, or errors that the real device cannot easily be made to produce. 100% device driver source code coverage can be achieved.

A device simulator can also be a useful training tool for developing device drivers for software engineers who develop device drivers. I attended a driver development course a number of years ago where the developers of the course (Dekker & Newcomer) provided a bare-bones device simulator to students instead of a device kit. It was basically a tool for their debug lab. But it was the inspiration for developing a full-featured device simulator in software.

A device simulator enables development on a platform with good development, testing and debugging resources when the target platform may have limited tools as in the case of tablets, or limited access as in the case of embedded systems.

Lastly, the kernel module of a PCI bus and device simulation is of necessity a subset of the PCI bus driver, and therefore any bus driver which supports child devices. It may prove useful as the basis for future bus driver development.

4 PROPERTIES OF PCI DEVICES RELEVANT TO SOFTWARE

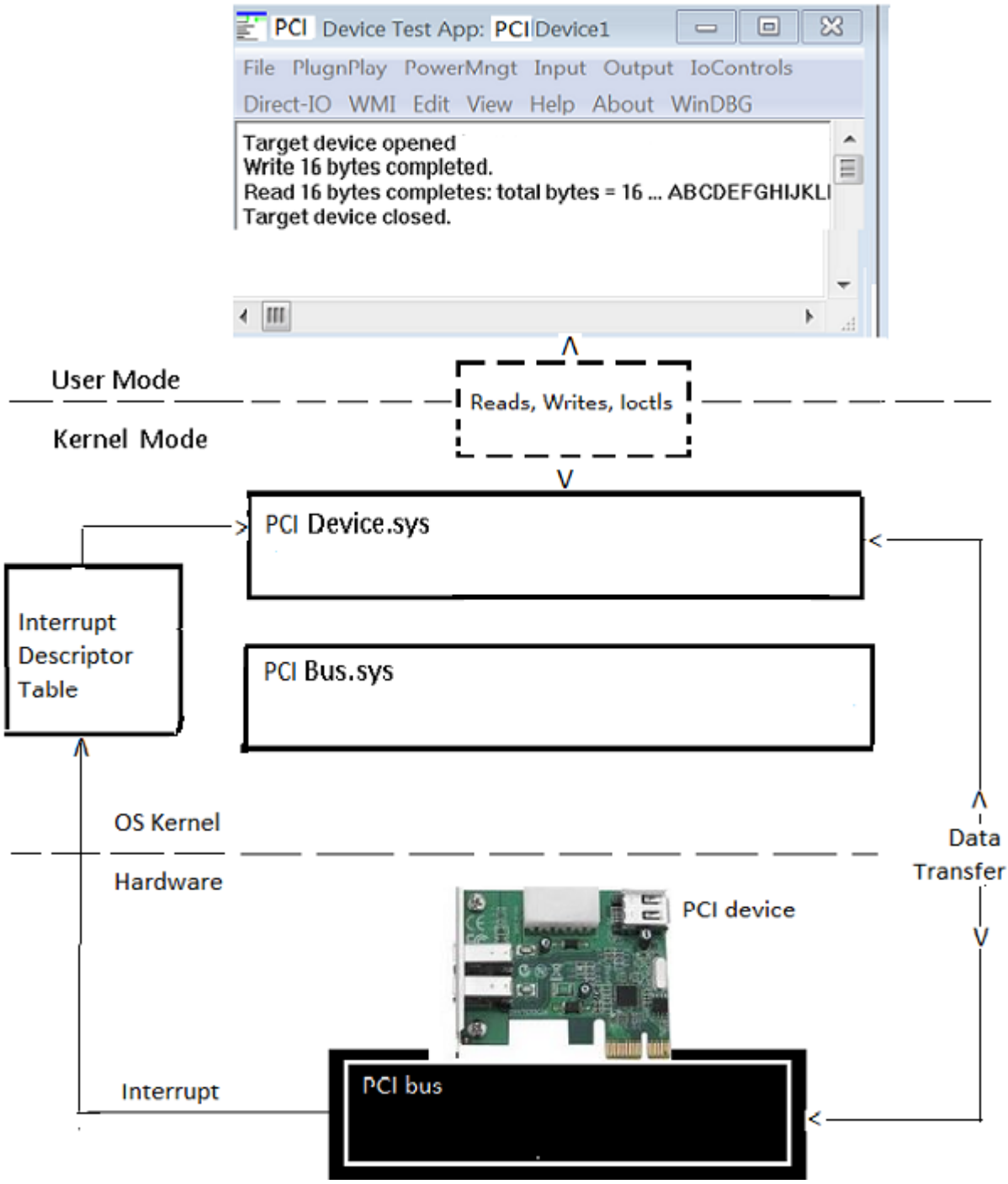
Nearly all PCI devices have these properties that driver software must manage:

1) They have a formatted configuration space containing information about identity and behavior properties.
2) They are memory-mapped.
3) They issue interrupts to the host.
4) They transfer significant chunks of data to/from the host – usually through Direct Memory Access (DMA).
5) They must conform to plug-n-play and power management expectations of the host operating system (OS).

4.1 PCI DEVICE HOST COMPUTER ENVIRONMENT

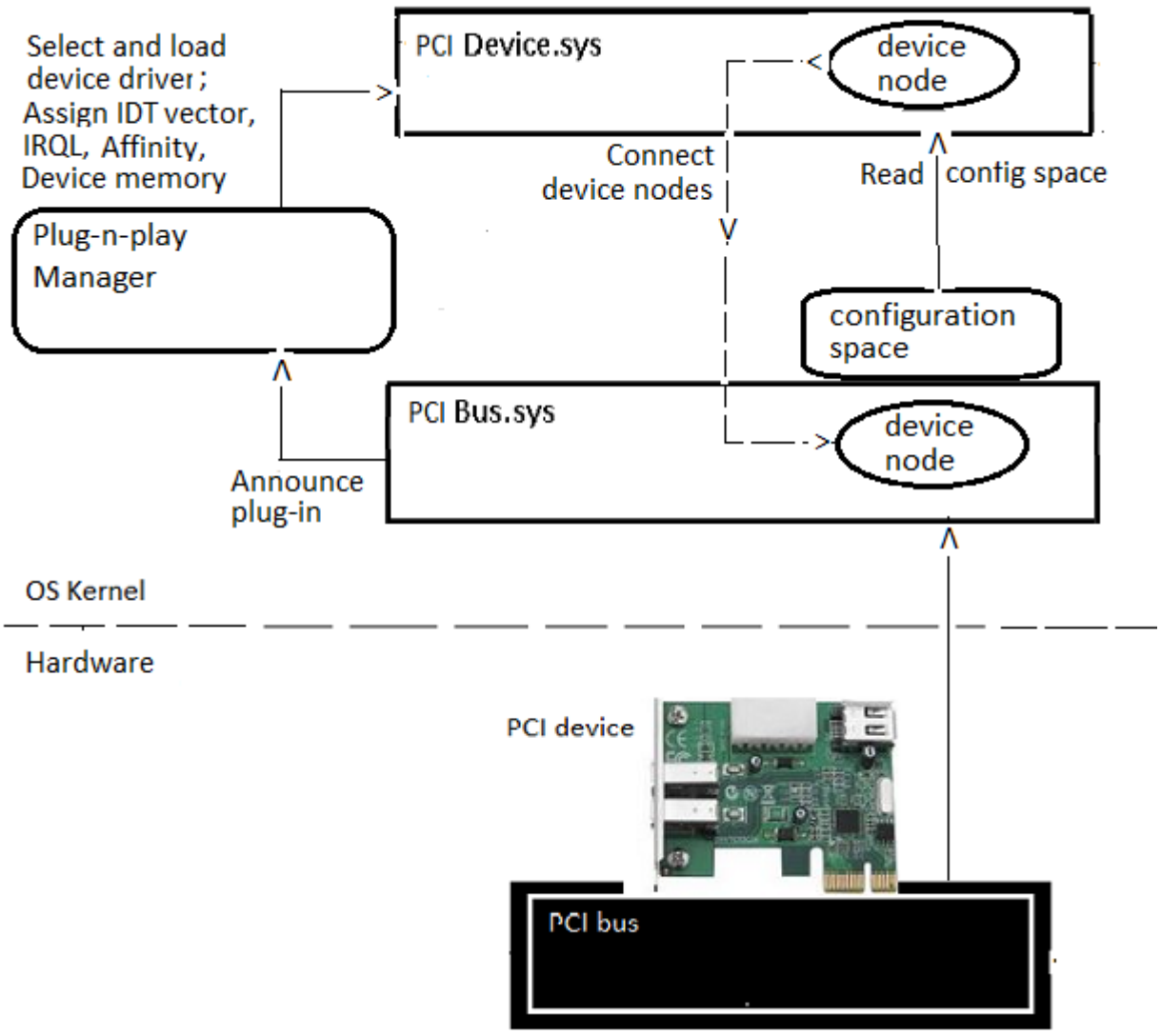
The diagram on the following page shows the host computer environment of a PCI device. From top to bottom we have: a test application which wishes to communicate with the device, the device driver that the test app interacts with, the bus driver that manages the PCI bus, and the PCI bus itself with the target device mounted onto it.

Observe that the test app interacts with the device (driver) through opens, reads, writes and Input-output controls (IOCTL)s. Also that the PCI device issues an interrupt to the host through the PCI bus, and the OS kernel delivers this interrupt to the device driver through a table lookup. Finally, observe that data is transferred – DMA or buffered – between the target device and driver, through the PCI bus.


PCI Device Test App: PCI Device1
File PlugnPlay PowerMngt Input Output IoControls
Direct-IO WMI Edit View Help About WinDBG
Target device opened
Write 16 bytes completed.
Read 16 bytes completes: total bytes = 16 ... ABCDEFGHIJKLI
Target device closed.
User Mode
Kernel Mode
Reads, Writes, Ioctls
PCI Device.sys
Interrupt Descriptor Table
PCI Bus.sys
OS Kernel
Hardware
Data Transfer
PCI device
PCI bus
Interrupt

4.2 PCI Device and Bus interaction with the Host Kernel

The following diagram shows the PCI bus driver and a PCI device driver interacting with the host kernel. Observe that device configuration space is maintained by the bus driver and is available to the appropriate device driver through kernel services. Also that the bus driver informs the kernel Plug-and-Play (PnP) manager of a new device instance on the bus. Finally observe that the kernel PnP manager assigns all resources needed by the device driver for the device.



4.3 CONFIGURATION SPACE

There is a 256 byte formatted configuration space in host memory for every device on a PCI bus. This memory is maintained by the bus driver. Configuration space contains two sets of information: identity and properties.

The identity information is: Vendor_Id, Device_Id, Revision_Id and Class Code. This information is sufficient to indicate which device driver should be assigned to the device, and what set of features are supported by the device.

The main properties that enable the device to function are: Interrupt pin, Interrupt line, Base Address Registers and Capabilities Pointer. The interrupt pin indicates which of two pins the (legacy) interrupt is wired into on the host motherboard. The interrupt line indicates which of four lines on the PCI bus will signal the interrupt. The Base Address Registers indicate the physical address of the device registers and other banks of device memory that require a mapping into physical address space by the host. The Capabilities Pointer is not a pointer but an offset from the beginning of configuration space to where a set of device capability parameters is located.

4.4 MEMORY-MAPPING

A PCI device will have a set of read-writeable registers that the host will use to program the device and determine device state. In addition, there may be other banks of memory on the device for other purposes. Each of these banks of device memory are assigned a physical address by the OS. Examples are device firmware accessible by the host and the Controller Memory Buffer of Non-Volatile Memory express (NVMe) devices.

We say that these memory banks are mapped into the host's physical address space. These addresses are stored in the Base Address Registers (0 – 5) in device configuration space.

4.5 Host Interrupt

When a device has completed a programmed i/o, or has a problem state to report, it issues an interrupt to the host hardware. This may be done by changing the voltage on a particular pin on the bus (legacy interrupt), or setting a specific value into a specific location of memory (Message Signaled Interrupt - MSI). The host OS is responsible for delivering the interrupt to the kernel software module which manages the device.

4.6 DMA Data Transfer

PCI devices usually transfer significant chunks of data to/from the host random access memory (RAM). This is usually done through DMA. DMA works without requiring clock cycles from the processor or buffer copying.

A DMA-capable PCI device will typically have a DMA controller which is capable of processing a Scatter-Gather List (SGL) in ram, formatted in a specific way. This list will contains physical addresses on the host, offsets on the device, transfer lengths and direction of transfer.

4.7 Plug-n-Play and Power Management

The PCI bus and device must perform several PnP and Power Management (PM) functions. They must plug in or unplug, and transfer to/from several different power states. A PCI device must change power state in response to system or device power state changes initiated by the OS.

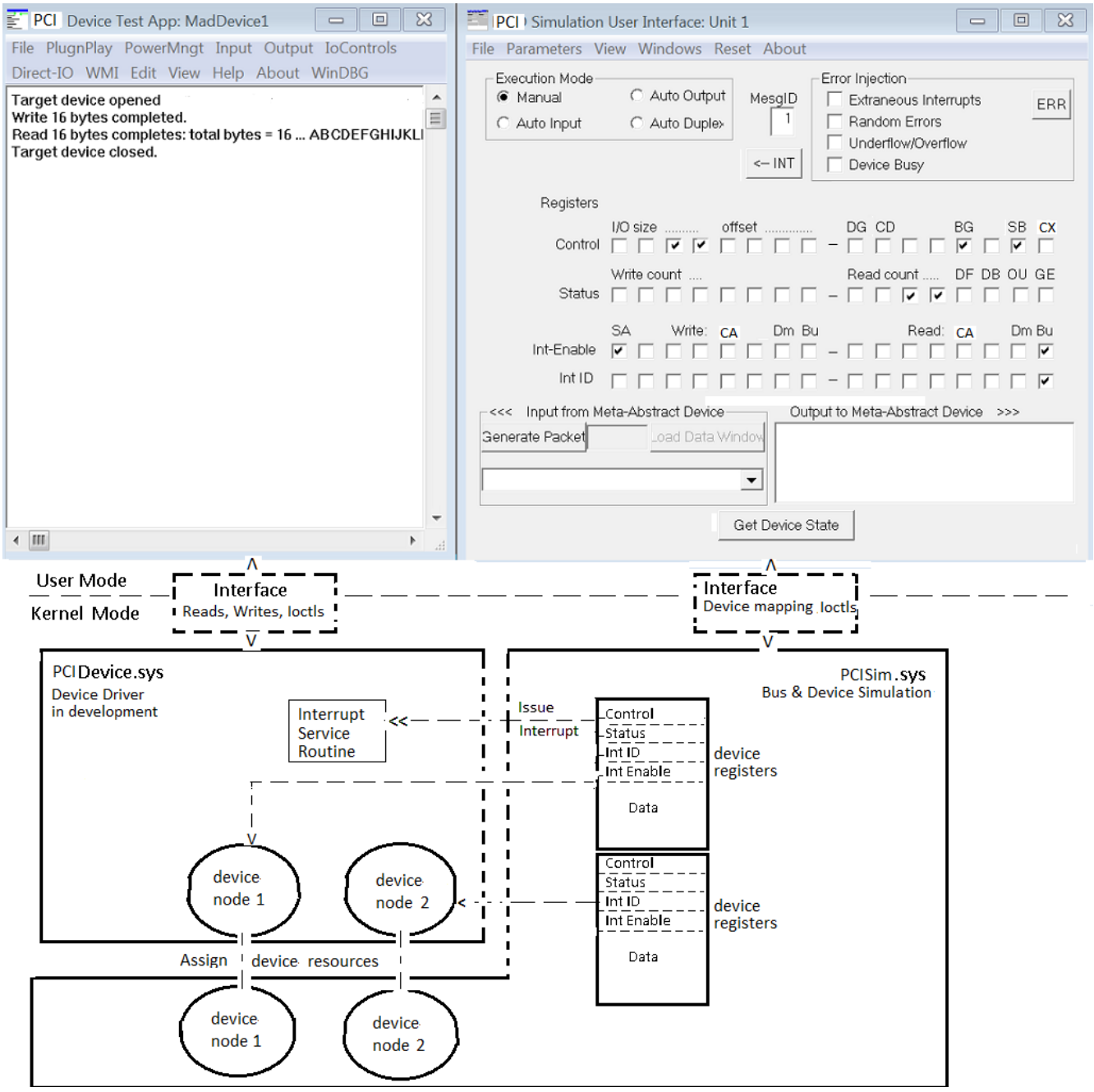
5 SIMULATED PCI DEVICE IMPLEMENTATION

A software simulation of a PCI bus and device must perform the following functions:

1) Present the 'device' to the host OS so that the natural PnP processing proceeds.
2) Provide configuration resources to the target device driver – simulating the PnP Manager. These include: address of the device registers, address(es) of other bank(s) of memory, Interrupt Descriptor Table (IDT) vector, device Interrupt Request Line (IRQL) and processor affinity.
3) Allocate sufficient memory for one or more devices and provide a mapping to "device memory" for both the target device driver and any device simulation UI program.
4) Invoke the target driver's Interrupt Service Routine (ISR) exactly as the OS kernel would do so.
5) Behave like the PCI bus or PCI device in response to external events – triggered by the UI program(s).

The diagram on the following page shows the PCI device driver in the simulation environment. Observe that the test app interacts with the PCI device driver as before. It should be completely unaware that the "device" is implemented in software. Also that there is a device node in both the simulation bus driver as well as the PCI device driver for each instance of a device. The simulation driver is under control of the user interface program.

The simulation driver maps "devices" into the device driver as the PnP manager would. The simulation driver invokes the device driver's Interrupt Service Routine (ISR) when the "device" interrupts.


PCI Device Test App: MadDevice1
File PlugnPlay PowerMngt Input Output IoControls
Direct-IO WMI Edit View Help About WinDBG
Target device opened
Write 16 bytes completed.
Read 16 bytes completes: total bytes = 16 ... ABCDEFGHIJKLI
Target device closed.
PCI Simulation User Interface: Unit 1
File Parameters View Windows Reset About
Execution Mode
Manual
Auto Output
Auto Input
Auto Duplex
MesgID
1
<-- INT
Error Injection
Extraneous Interrupts
Random Errors
Underflow/Overflow
Device Busy
ERR
Registers
Control
Status
Int-Enable
Int ID
I/O size
offset
DG CD BG SB CX
Write count
Read count
DF DB OU GE
SA Write: CA Dm Bu
Read: CA Dm Bu
<<< Input from Meta-Abstract Device
Output to Meta-Abstract Device >>>
Generate Packet
Load Data Window
Get Device State
User Mode
Kernel Mode
Interface
Reads, Writes, Ioctls
Interface
Device mapping Ioctls
PCIDevice.sys
Device Driver
in development
Interrupt
Service
Routine
Issue
Interrupt
PCISim.sys
Bus & Device Simulation
Control
Status
Int ID
Int Enable
Data
device
registers
device
node 1
device
node 2
Assign device resources
device
node 1
device
node 2

5.1 DEVICE PRESENTATION

As stated, every bus driver must indicate to the OS that a new device has arrived when it detects a plug-in. A simulation of the PCI bus would do the normal processing to create a new device node and call the appropriate kernel service to alert the OS that there is a change in configuration on the bus. This should cause the target device driver to do its normal processing for a plug-in. When an unplug occurs, the bus simulation must delete its device node and again call the appropriate kernel service to alert the OS that there is a change in bus configuration.

When a change in system or device power state is initiated by the host the device simulation should faithfully behave as the device would.

5.2 RESOURCE MANAGEMENT

The simulation must provide resource/configuration management parameters to the target device driver as if they came from the PnP manager. This includes a physical address for device registers, interrupt parameters (IDT entry, device IRQL, processor affinity) and physical addresses for any other banks of device memory that the driver needs to access.

5.3 MEMORY ALLOCATION

The simulation must allocate enough memory to represent one or more devices in ram. It must provide a mapping into a virtual address space for the target device driver and the device simulation graphical user interface (GUI) program.

5.4 Invoking an ISR

The simulation must invoke the target device driver's ISR exactly as the OS kernel would. This entails running at the proper processor affinity, acquiring the spinlock specific to the device interrupt object, raising the host execution level to device IRQL, and then passing the correct arguments to the target driver ISR. These arguments would be the interrupt object and perhaps some context information and/or an MSI message-id.

5.5 Bus and Device Operation

The simulator transfers data between the host and "device memory" as would the device. The GUI representation of the device displays the current state of the device. The simulator also behaves as does the device DMA controller. This entails processing the hardware SGL and transferring chunks of data between the host physical addresses and device memory.

5.6 Target Device Driver Requirements

The target device driver needs to be built for **simulation_mode** as distinct from **real_mode**, where real hardware exists. This is because the target device driver must exchange a few parameters with the device simulator so that the simulator can know how to interact with the driver as the OS would. These parameters are: the interrupt object, the spinlock object for the interrupt, the device IRQL, the interrupt processor affinity, and the address of the driver ISR.

Ideally, there would be no if-then-else logic to vary the execution path between simulation_mode and real_mode. There should only be some additional logic in simulation_mode.

This is **not** good because we are missing some code coverage in simulation mode.

```
#ifdef SIMULATION_MODE…  // Do it this way…
#else…                   // Do it this other way…
#endif
```

This is the preferred way. We are merely adding code in simulation mode to help the simulator.

```
#ifdef SIMULATION_MODE

    //Pass some parameters to the simulator

#endif
```

6 EXTENT OF USEFULNESS

A software simulator produced from source code has no platform constraint. The target driver, user application(s) and simulation software can be built and exercised on 32 or 64 bit; X64, Itanium or ARM architecture, as well as any recent or new release of the target OS -- Windows, Linux, or other.

As mentioned above, a simulator can even be run cross-platform. The device driver for a device which is targeted for the ARM architecture can be developed and exercised on legacy i386 or x64 before porting to the target platform, which may have minimal development and test support.

Simulation software can be run on a native hardware platform or on a virtual machine (VM) such as VBox, QEMU-KVM, VMware, or Hyper-V.

A software simulation of a memory-mapped device is not even constrained to the PCI bus. Should a truly new memory-mapping bus come along, or a significant enhancement to PCI(x) that software would be sensitive to, it is a simple matter to modify the simulation and target software for a different bus and configuration space, and then rebuild.

7 CAVEATS

A simulation from compiled source code cannot accurately model any new target device as is. It is necessary to customize the software from a prototype device simulation to one consistent with the specifications of the target device.

A software simulation cannot accurately model the timing or throughput performance of the device interacting with the bus. Register reads and writes across the bus are implemented in the simulation as memory transfers within ram. DMA transfers are implemented by buffer copying within ram. Multiple devices on the same bus will compete for bus bandwidth.

A device simulation – even with 100% code coverage - cannot be a substitute for thorough testing with the real device. New devices have their quirks and don't always behave just like the specification. Real-world devices can cause resource conflicts among themselves, leading to intermittent mis-routing of interrupts.

Of course, if the specification for a device changes, the simulation must change as well.

8 What Might It Look like? - A Working Model

The reader may have guessed by now that the author has a working model of PCI device simulation in the Linux and Windows domains. This includes char-mode and block-mode devices in Linux and generic device and storage devices on Windows. On the author's web site there is a diagram showing how the components work together and a three minute demonstration of operation. Please see the authors' web site here… ***http://www.htfsoftware.com***

9 Development Pipeline

1) Build a front end for the device simulation such that **FPGA** or **ASIC** device definitions can serve as input to generation of simulations for devices in development.
2) Add reporting capabilities so that it's possible to track what has been tested as well as performance results.

10 Conclusion

A software simulation of PCI devices is both feasible and beneficial. Chief benefits include parallel development of software and hardware, 100% code coverage, platform flexibility and hard-to-create or rarely occurring conditions produced readily.

With a prototype working on the target OS, customization and deployment can be achieved in a matter of days or a few weeks -- a device driver developer's dream.

A working model of PCI device simulation exists both in the Linux and Windows domains.

I leave it to the reader to consider the cost versus benefit.

I thank the reader for their interest.

11 GLOSSARY OF TERMS

ASIC – Application Specific Integrated Circuit - an integrated circuit customized for a particular use, rather than intended for general-purpose use. For example, a chip designed to run in a digital voice recorder is an ASIC. ASICs are fixed, they are not modifiable. As opposed to FPGAs which are field programmable. ASICs are the brains of many PCI devices.

DMA - Direct Memory Access - a means of transferring memory between a device and host memory in a computer without a processor being involved in buffer copying. DMA is much faster than normal memory transfers but is more difficult to set up by programming the device.

FPGA – Field Programmable Gate Array – an integrated circuit containing an array of programmable logic blocks and a hierarchy of reconfigurable interconnects that allow the blocks to be wired together, like many logic gates that can be inter-wired in different configurations. These logic blocks form the basis for computer devices which perform complex logic operations as part of their function. FPGA defined logic is the brains of many PCI devices.

GUI - Graphical User Interface – a means of interacting with a computer and its electronic devices through visual icons and indicators as opposed to text-based interfaces or text navigation.

IDT - Interrupt Descriptor Table – a table in host memory which is used to route hardware interrupts from specific devices to the appropriate interrupt service routine in a device driver which knows how to control the interrupting device.

IOCTL - Input Output Control – a command from a user program to a device through its device driver to perform some specific defined function. It may be an administrative function or an i/o which is not a simple read or write.

IRQL - Interrupt Request Level – an execution level which an interrupt service routine will run at assigned by the OS. When a processor is running at execution level X, only an interrupt at IRQL X+1 can gain execution on the same processor.

MSI - Message Signaled Interrupt – a means of triggering an interrupt to a host by writing to host memory rather than changing the voltage on a specific pin wired to the host. Non MSI interrupts are now referred to as legacy interrupts.

OS - Operating System – a software system which runs on a computer and performs all administrative functions for one or more end-users. Well known small computer OSes are: Windows, Linux, and VxWorks.

PCI - Peripheral Component Interface - a local computer bus for attaching hardware devices in a computer. Attached devices are typically expansion cards that fit into slots but may be an integrated circuit fitted onto the motherboard itself.

PM - Power Management is a feature of computers and computer peripherals that turns off the power or switches the system or device to a lower power state when inactive. As more and more computation devices have become portable and battery-powered, power management has increased in importance.

PnP - Plug-and-Play – a feature of modern OSes that allow attaching a new device and having it perform its normal functions without having to reconfigure the OS and without having to reboot the system.

Processor Affinity – is the process of binding and unbinding a process, thread or function to a particular processor or set of processors, so that they will execute only on the designated processor(s). The benefit to binding hardware interrupt functions to processors is so that multiple packets of data from a device will be in the same processors' cache. This can be a significant performance improvement.

RAM – Random Access Memory - is a form of computer data storage allowing read or write of digital data in almost the same amount of time irrespective of the physical location of data inside the memory (thus random access). RAM is normally volatile, meaning stored information is lost if power is removed. Therefore it is inappropriate for permanent storage.

SGL - Scatter-Gather List - a set of elements in memory describing a DMA transfer between the host and a device. A Scatter-Gather List is formatted specific to the requirements of the target device, and is processed by the device's DMA controller.

VM - Virtual Machine – an emulation of a computer and an OS running as a software simulation on a host machine and a VM supporting OS. Examples of VM Oses are: VMware by VMware, VBox by Oracle, and Hypervisor by Microsoft.

12 REFERENCES

Fones, Thomas ***http://www.htfsoftware.com*** *2016 (Author's web site)*

Dekker, Edward N. & Newcomer, Joseph N. Developing Windows NT Device Drivers; chapter 27, 1999

https://en.wikipedia.org/wiki/Conventional_PCI

https://en.wikipedia.org/wiki/PCI_configuration_space

A Novel Approach to PCI Simulation Using ScriptSim
http://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1176&context=theses

A PCI BUS SIMULATION FRAMEWORK AND SOME SIMULATION RESULTS ON PCI STANDARD 2.1 LATENCY LIMITATIONS *http://dl.acm.org/citation.cfm?id=612393*

Design and Simulation of a PCI Express based Embedded System
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.144.9526&rep=rep1&type=pdf