PCI-DP : Programming Bus Master DMA Transfers

Introduction

This application note explains Cypress-supplied sample code that demonstrates how to use the Bus Mastering DMA controller built into the PCI-DP CY7C09449 (formerly the Anchor Chips AN3042 CO-MEM Lite) chip. All components, hardware and software, focus on one objective: to move data (quickly and with minimal loading on other parts of the PC) from Host Memory to the PCI-DP's Shared Memory, and visa versa. This is shown by the arrow labeled *TRANSFERS* in *Figure 1*.

Cypress Semiconductor provides this sample software to help you get your PCI-DP design done and into production. Since we've already done much of the system programming for you, you can concentrate on your application, not on low-level system software.

Typical **sustained** performance for this code is 75 MB/s for Reads from Host Memory, and 100 MB/s for Writes to Host Memory.

If you are pressed for time, you can use the code as provided, treating it as a black box, without delving into how it actually works. You can focus mostly on your code that *calls* doDMA. See the PCI-DP DMA Feature Summary and Quick Start.

However, we think you will find it useful to understand how the PCI-DP fits into the PC architecture to help you diagnose problems.

If you are using the PCI-DP in an embedded system that doesn't use Windows, you may have to make changes at the device driver level (Kernel Mode, Ring 0), but the application-level (User Mode, Ring 3) code may be useful unchanged.

We'll cover:

- Features of the DMA Controller.
- How to get started quickly, ignoring most of the details.
- PC Architecture and the Pentium® as it applies to Bus Master DMA transfers.
- The functions testDMA and doDMA, components of Test3042.
- The way our Device Drivers help set up DMAs.
- Performance issues.
- The limitations of this sample code.

PCI-DP DMA Feature Summary

Here's a summary of the features of the PCI-DP's DMA controller (see CY7C09449PV data sheet).

- The DMA controller's registers can be accessed (initialized) from either the PCI or Local bus. This is sometimes referred to as *PCI-initiated* DMA or *Local-initiated* DMA respectively. In *Test3042,* we only show PCI-initiated DMA.
- Ownership bits (L and P) in the DMACTL register allow the Host side or Local side to claim the DMA controller (to prevent the other side from corrupting the DMA registers by accessing them at the same time).
- The DMA controller always moves data by making the PCI-DP the PCI Bus Master. The PCI-DP normally uses the PCI cache line commands MRM (Memory Read Multiple), MRL (Memory Read Line) and MWI (Memory Write and Invalidate) to optimize throughput and minimize the load on the rest of the system.
- The DMA controller always moves data a DWORD at a time, using all 32-bits (4 byte-lanes) for each data phase. Hence, the size of a transfer can be 4, 8, 12, …16 K bytes, never 1, 2, 3, 5, … bytes.

The PCI-DP's Shared Memory is either the source or destination in every DMA transfer. The other device (the destination or source) can be Host Memory, or any other PCI device that is mapped into the PCI Memory Space.

- The maximum transfer length is 16 KB.
- Most data phases occur with zero wait states (depending on the bandwidth supported by the PCs chipset and host memory). TRDY and IRDY stay asserted, causing a new DWORD to be transferred on each 33 MHz PCI clock, achieving a burst of 132 MB/s for most of any large transfer (up to 16 KB).
- The DMA controller can transfer data to/from any address in the full 32-bit PCI Memory Space.

Notes:

1. When this application note says *Host CPU* or *Host Processor*, think *Pentium*, meaning the Pentium itself and all its successors (Pentium, Pentium Pro, Pentium II, Pentium III and Celeron). Windows 9x, Windows NT and Windows 2000 uses all Pentium variants in modes which are identical for our purposes (4 KB pages, 32-bit flat model). Although the Win9x and WinNT code bases can theoretically run on older processors (386 and 486), even 486 PCs rarely have PCI buses.
2. See *Memory Space* in the Glossary

Quick Start

First, there are a few things you need to do before calling doDMA:

1. Install the driver (comemL.vxd or comemL.sys) into c:\windows\system and reboot your PC.
2. In your program:
   a. Open the driver by calling comemOpenDriverL (*not* shown in Listings 1 or 2). See the program file TestUtil.cpp that you downloaded with *Test3042*.
   b. Call comemCopyBarPtrL to get a Linear pointer to the PCI-DP so you can access its Op Regs (Operation Registers) from your program (see Listing 1).
   c. Call comemAllocContigMemL to allocate a buffer which is contiguous in both physical and linear address space (see Listing 1).
   d. Call doDMA (see Listing 1).

Then, here's what happens in doDMA (see Listing 2). For many applications, you can use doDMA unchanged.

For simplicity of explanation, let's assume the direction of transfer is *towards* the Host Memory (the PCI-DP controller will do one or more MWI [Memory Write and Invalidate] transactions):

source = a block of *n* DWORDs in PCI-DP Shared Memory (previously loaded with data).

destination = a buffer in Host Memory (previously allocated by a call to comemAllocContigMemL).

1. Set the DMALBASE register to point to the **source**, the start of the data in Shared Memory (see line 18 of doDMA in Listing 2). A value of 0 points to the zeroth DWORD is Shared Memory.
2. Set the DMASIZE register to the number of DWORDs to be transferred (line 23).
3. Set the DMAHBASE register to the physical address of the **destination** (line 28).
4. Clear the DMA completion bit (HINT register, bit 5) (line 32).
5. Kick off the DMA by writing to just the low order byte of the DMACTL register (line 36). Here, we both set the direction and *start the DMA.*
6. Wait for the DMA complete bit to be set (HINT register, bit 5) (lines 49-66).

PC Architecture 101

Which DMA Controller?

The original type of DMA in PCs, ***System DMA***, uses the DMA controller in the PCs main chipset. Originally, the Intel 8237 chip was the System DMA controller in the IBM PCs. The functionality of the 8237 was later integrated into PC chipsets, for example, the PIIX3 (PCI ISA IDE Xcelerator) part of Intel's chipsets. Every PC has at least two System DMA controllers.

The System DMA controller is limited to transfers within the lowest 1 MB of physical memory and is slower than the PCI-DP DMA controller, having been designed originally for the ISA bus, a bus that can transfer at most 6 MB/s. Moreover, the System DMA controller can only move data between the ISA bus and Host Memory, never between PCI devices.

However, the type of DMA discussed in the remainder of this application note is ***Bus Master DMA***. *DMA* always refers to Bus Master DMA — DMA driven by the DMA controller in the PCI-DP chip, *Bus Master* because the PCI-DP becomes the PCI bus master.

The corresponding *Target* (to the PCI-DP's Bus Mastering) is normally the *Host to PCI Bridge*. DWORDs are transferred through the Host to PCI Bridge to the system main memory.

The Target device *can* be another device on the PCI bus (including other PCI-DPs), in *peer-to-peer* transfers.

Software

The code provided in Test3042 and the Device Driver has the following parts (see Figure 1):

① High-level Application Code

GUI (Graphical User Interface) routines (not discussed further in this application note).

② Low-level Application Code

The primary code discussed in this App Note (see Listing 1 on p. 6 and Listing 2 on p. 9).

③ PCI-DP Interface Library (comemLif.lib)

Major functions:

1. Allows the application to create objects of class PCI-DP. This code is architected to easily support multiple PCI-DP chips in one system.
2. Provides routines that interface to the Device Driver ④.

For the sample program Test3042, ①, ② and ③ are *statically* linked by the Visual C++ linker, producing Test3042.exe. This is indicated by the ①, ② and ③ boxes actually touching.

The short line connecting ③ and ④ indicates that ④ is *dynamically* (run time) linked to the Test3042 process when Test3042 opens the device driver. This dynamic linking occurs in the call to comemOpenDriverL in TestUtil.cpp (not shown in Listings 1 or 2).

④ Device Driver

A Ring 0 Device Driver is an essential part of our PCI-DP support because only Ring 0 code is privileged to access the Paging Unit. Application code (layers ① and ②) running at Ring 3 cannot access the Paging Unit.

By accessing the Paging Unit, the Device Driver provides two vital services to the Ring 3 application:

1. The Device Driver converts the value in the PCI-DP's BAR0 (Base Address Register 0) from a Physical to a Linear address.

   BAR0 is initialized by the BIOS at boot time. In Physical Address Space, BAR0 is the base address of all PCI-DP functions that are visible from the PCI side (the operation registers and Shared Memory). The Device Driver converts this BAR0 Physical address to a Linear address, so the application can access the PCI-DP.

2. When called upon (through the IOCTL_COMEM_ALLOCCONTIGMEM entry point), the Device Driver allocates a 16 KB block (four 4 KB Pages) that become the source or destination for PCI-DP DMA ***TRANSFERS***. The 16 KB block *must* be

contiguous in both Physical and Linear memory space. Contiguity can only be guaranteed by the device driver running in Ring 0, because it has access to the Page Tables, and the application program does not. 16 KB is the size of PCI-DP Shared Memory, hence 16 KB is the maximum size of a DMA transfer.

The actual Device Driver file in your system (normally installed in c:\windows\system) will be one of two types:

Win 9x comemL.vxd

Win NT comemL.sys

As shown by the arrows which go to the host side of ⑥, it is common for each software layer (①, ②, ③, and ④) to directly access the PCI-DP, (through the Host to PCI bridge ⑦), via a base Linear address obtained earlier from the Device Driver.

Hardware

Test3042 and other host processor software accesses the hardware components by generating memory accesses that are converted by the Paging Unit from Linear to Physical addresses.

⑤ Host memory – Is in the same Physical Address Space as PCI devices (but of course at a different location within that space).

⑥ Chipset – These are most commonly Intel Chipsets, such as the Intel® 440BX AGPset and Intel® 440ZX AGPset. See http://www.intel.com/design/chipsets .

⑦ Host to PCI Bridge – a part of the Chipset that interfaces the Pentium, Host Memory, etc., to the PCI bus.

⑧ PCI Bus – 32 bit, 33 MHz.

⑨ A PCI-DP and Local-side Circuit on an add-in board plugged into the PCI Bus.

⑩ The Local Circuit. Can consist of:

- a complex state machine, or nearly any microprocessor
- a memory subsystem, which can consist of a mix of memories
- any application circuitry.

Note:

3. Intel never uses the terms Paging Unit per se in their documentation. They talk about the low-level components of the paging mechanism, such as Page Tables, Page Directory, etc., but never specifically about a Paging Unit. This is probably because the functionality of a paging unit is performed by multiple components, some on the Pentium chip, the Page Tables in Host Memory, etc. See the Glossary for the definition of Paging Unit.

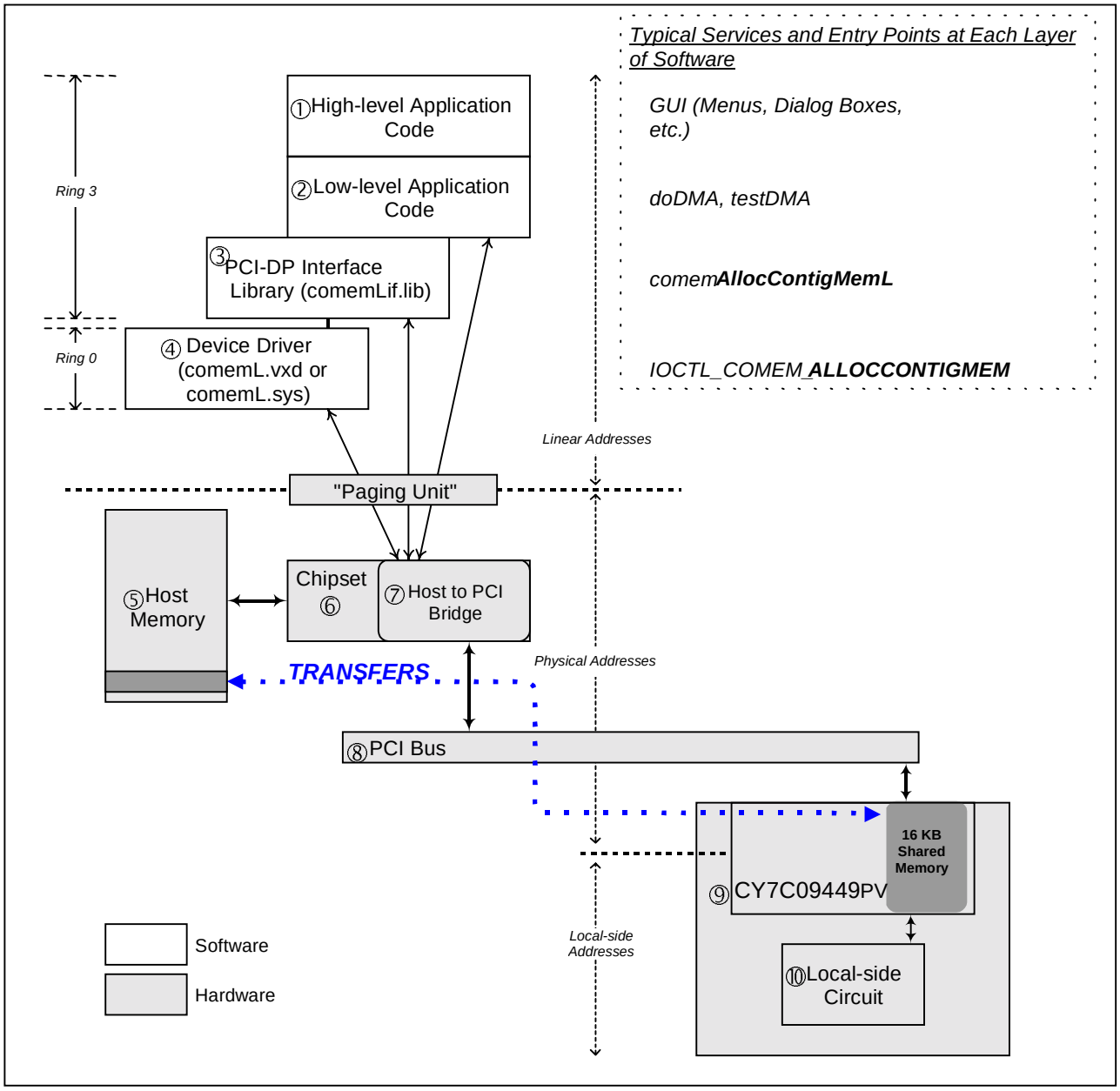

Figure 1. PC Components Involved in Bus Master DMA Transfers

Notes about *Figure 1*:

- In general, each software layer calls services in lower layers. However, after a layer has obtained a Linear Address (a 'C' pointer) to the PCI-DP, it is free to read or write directly to the PCI-DP without 'calling through' lower layers. See the three bidirectional arrows that point to the *Host to PCI Bridge*, which originate at ② and ③, as well as ④. Also see code in Listing 2. For example, statements such as line 23 in Listing 2

  ```
  g_regp->dmasize = length - 4;
  ```

  access the PCI-DP directly.

- *All* Software lives in the domain of Linear Addresses, whereas all hardware lives in the domain of Physical Addresses.
- Win9x and WinNT use a flat model (32-bit protected mode). There are exceptions, which are buried in Windows internals and normally don't concern us. For example, Win9x still uses *16-bit* protected mode for much of the code in User.exe (one of several key Windows executable files). Test3042 and all new applications that you write should be compiled into flat-model 32-bit code.

Other Important Firmware and Software Components

- The Plug and Play (PnP) BIOS (a part of the System BIOS).

  At boot time, during the first second after Reset is released, the PnP BIOS scans the PCI bus and assigns a unique Physical Address range ('Region') to each PCI BAR (Base Address Register). This process is called *enumeration.*

  Since the PnP BIOS is a fixed component on the motherboard, it 'knows' which resources are required by devices on the motherboard (such as Host Memory), and can reserve those, assigning the remaining Physical Addresses to add-in devices.

  See the PCI 2.2 Specification (Ref. *PCI Local Bus Specification, Revision 2.2. PCI Special Interest Group, Portland, OR*.) for details about the device scan, BAR operation, etc.

  When Windows boots, it may decide to redo what the PnP BIOS did. For example, if Windows doesn't think that the device has a valid device driver (Windows can't find a match in the Device Driver Database), it will do the 'safe' thing, disabling the device by writing zero to all its BARs.
- Windows – Called on by the application and the Device Driver for *many* services.
- comemL.inf – Is the major file to tell the Windows installation code how to match the PCI-DP to the correct Device Driver.
- The Registry.
- (Win9x Only) The Device Driver Database, which consists all .bin files in the directory c:\windows\inf. These files are a compiled database of all the .inf files in the c:\windows\inf tree. You can delete them to force the Plug and Play system to rebuild this database during the next reboot.

The Sample Code

See Listings 1 and 2. Important code is in **bold type**. Since the code is heavily commented, please see the code for details.

The code consists of two C++ functions: doDMA and testDMA. (If you don't know C++, don't panic. We've used very few of the C++ extensions to C.)

testDMA is intended to be called by higher level code after the device driver has been opened, then testDMA calls doDMA.

Performance

Typical **sustained** performance for this code is 75 MB/s for Reads from Host Memory, and 100 MB/s for Writes to Host Memory.

By taking certain shortcuts (not shown in Listings 1 and 2), we have been able to push Write performance to over 120 MB/sec., remarkably close to the 132 MB/s theoretical limit of a 32-bit 33-MHz PCI bus. The overhead to set up the next transaction, and pauses during a transfer, is extremely low for this code and the PCI-DP.

Large transfers (16 KB) are fastest because the key factor in overall speed is the time to set up the next transfer.

Most of the time to set up the next transfer is consumed by transactions on the PCI bus itself, not execution of the code in the Pentium core. When you reach the optimization phase of your project, try to minimize the actual DMA setup and polling transactions on the bus. They are *much* slower than execution of instructions in the Pentium core.

For example, a write to an PCI-DP Op Reg takes about **6** PCI clocks = 180 ns. In the same time (assume a 300-MHz Pentium = 3 ns instruction execution time, and DMA code running from the Pentium's instruction cache), the processor core will execute about **60** instructions. If you can eliminate one PCI transaction, you can save the time it takes to run about 60 instructions!

Limitations of This Sample Code

Interrupts

doDMA (Listing 2) polls the PCI-DP for DMA completion. At the expense of considerable complexity, an interrupt-driven DMA device driver could be written.

There is a possibility (though not a certainty) that *overall system performance* would be improved using interrupts, because interrupts would allow processor cycles that are consumed in the spin loop (waiting for DMA completion) to be used to by other threads to improve overall system responsiveness.

Throughput across the PCI bus, however, would probably *not* be improved, because PCI throughput is mostly limited by:

- DMA setup time
- PCI bandwidth consumed by other PCI devices
- The ability of the Target device (the Host to PCI Bridge and the Host Memory system) to source or sink the data. Newer chipsets can be more efficient than older ones, because they use a variety of techniques to sustain zero wait-state PCI bursts.
- The load on Host Memory by other agents, for example, the AGP bus, and instruction fetches that must actually fetch from Host Memory (instruction fetches that missed in both Level 1 and Level 2 caches).

Polling is reasonably efficient because DMAs are done quickly — 16 KB transfers complete in about 150 µs. Interrupts may not improve performance much because of the overhead involved in interrupt processing:

- saving and restoring processor state
- cache thrashing when the context is switched to the Interrupt Service Routine (ISR) and back.

Data To/From the Local-Side Memory

In some applications, it may be OK to leave the data in Shared Memory and access it from the Local-side Circuit *in place* without moving to other memory on the Local Side. However, in most high-speed applications, an extra step would be required: to move the data from the Shared Memory to a Local-side memory so the Shared Memory can be used for the next load of data.

The code in doDMA does nothing to prod the Local-side Circuit to move the data from 3042 Shared Memory to its own local memory.

Here's a possible double-buffered scheme to move the data from Shared Memory to Local-side memory, designed to maximize sustained throughput. (For simplicity of explanation, assume data is moving from Host Memory *to* the Local-side memory.)

The 16 KB Shared Memory is split into two halves. A single-DWORD guard band at address 0 and 0x2000 is required because of Errata #4 (Local bus Read access to the PCI-DP Shared Memory will block PCI Write access to Shared Memory in some cases).

1. PCI-DP DMA controller moves data from Host Memory to 1st half of Shared Memory.

   Simultaneously, the Local-side Circuit moves data from 2nd half of Shared Memory to the Local-side Memory (data put there from the Host Memory the last time the PCI side 'owned' the 2nd half).

2. When both halves are transferred, the processors synchronize through the mailbox registers (HLDATA or LHDATA).

3. The programs on each side change buffer pointers.

   Host points to 2nd half, Local points to 1st half.

4. Both processors would then start their transfers to/from their new half.

   Host to 2nd half, Local from 1st half.

PCI-DP DMA can be set up and initiated by the Pentium ('Host-initiated DMA'), or by the Local circuit ('Local-initiated DMA').

Sharing the PCI-DP DMA Controller

For the code in Listing 1 and 2, we've assumed that only one thread will access the DMA controller. If more than one thread on the host side wants to access the DMA Controller, or if the Local-side processor might access the DMA controller, code to perform software arbitration must be added.

Ref. 5 recommends using the VDMAD (Virtual DMA Driver, a VxD) to virtualize[4] any DMA controller.

Additionally, you may want to use the L and P bits in the DMACTL register to arbitrate between Host-side and Local-side ownership of the PCI-DP's DMA controller.

Listing 1: testDMA

```
DWORD g_linBAR[COMEM_MAX_BARS]    // Global array which will contain the Linear BAR0 and BAR1 addresses.

// This structure MUST be designated 'volatile' for the VC++ 'Release' version (as opposed to the 'Debug' version).
// If this structure is not designated 'volatile', optimizers destroy the intent of inner-loop code in doDMA below.
volatile struct op_regs_struct_42_t * g_regp;

//
// testDMA - An example function that tests the 3042's DMA controller that:
//    - allocates and deallocates the Host Mem physical pages
//    - writes patterns to the source and destination memory
//    - calls doDMA to initiate the transfer
//    - checks the result of the 3042 DMA transfer.
//
DWORD testDMA(DWORD direction, DWORD comemID)
{
    DWORD lin, phys, i, returnCode;
    int errorCnt = 0;

    printf ("\n");

    if (direction == directionToHost)
        printf ("DMA data from 3042 Shared Mem to Host Mem...\n");
    else
        printf ("DMA data from Host Mem to 3042 Shared Mem...\n");

    // Get the Linear BAR0 and BAR1 pointers (we only use BAR0 here) so we can access the 3042's registers.
    // You may want to move this out of testDMA, doing it only once at the start of your application.
    //
    // Throughout this code, 'L' at the end of function names means Lite, as in CO-MEM Lite.
    returnCode = comemCopyBarPtrL(g_linBAR, comemID);
    if (returnCode != NO_ERROR)
    {
        reportErrorCode(returnCode, "creating BAR pointers.");
        errorCnt++;
    }

    // Setup the structure template we use to access the 3042 op regs.
    g_regp = (struct op_regs_struct_42_t *) (g_linBAR[0] + OP_REGS_BASE_42);


    // Allocate 4 Pages (4 KB each, 16 KB total = size of 3042 Shared Mem) which are (must be!)
    // contiguous in Physical (and Linear) memory space.
    //
    // We pass the Physical address to the 3042's DMA controller DMAHBASE register.
```

Note:

4. To virtualize a device is to provide an intermediary VxD between the application layers and the device. (The computer science community loves the word virtual.) A VxD, a Virtual Anything (x) Driver, is then called by any thread that wants to access the device. The VxD maintains a separate state table for each thread, arbitrates for the device, and manages any conflicts that arise.

```
45      // We need the Linear address to reference the Pages from this program, which runs on the 'Linear side' of
46      // the Pentium's Paging Unit.
47      //
48      // This allocation may be done once when the application is launched.  However, here we chose to
49      // allocate and deallocate for each invocation of testDMA.  Remember to pair allocations with
50      // deallocations.
51      returnCode = comemAllocContigMemL (4,        // In:  Number of 4 KB Pages. comemAllocContigMemL can allocate 1 to
4 Pages,
52                                                   //      but must allocate 4 for this test.
53                                         &lin,     // Out: Returns the Linear address of the allocated Pages.
54                                         &phys,    // Out: Returns the Physical address of the allocated Pages.
55                                         comemID); // In:  The 3042 that 'owns' the pages.
56      if (returnCode != NO_ERROR)
57      {
58          printf("Error in comemAllocContigMemL.");
59          return (++errorCnt);
60      }
61
62      DWORD * hostLin  = (DWORD *) lin;
63      DWORD * hostPhys = (DWORD *) phys;
64
65      // For this test move the max block size. Start at the beginning of Shared Mem and use the maximum DMAsize.
66      DWORD SMstart = 0;
67      DWORD DMAsize  = 0x4000;    // 16 KB
68
69      if (direction == directionToHost)
70      {
71          // Initialize source buffer (3042 Shared Mem).
72          //
73          // initSharedMem writes to the 3042's Shared Mem as a PCI target, which is about 10 times
74          // slower than the DMA transfer.
75          initSharedMem(SMstart, DMAsize, ADDR_PATTERN, comemID);
76
77          // Write pattern to destination, so this test can't pass unless 3042 DMA controller actually moves the data.
78          // printf ("Initializing destination at Linear 0x%08x  pattern=0xFEEDBEEF\n", hostLin);
79          for (i = 0 ; i < DMAsize/4 ; i++)
80              hostLin[i] = 0xFEEDBEEF;
81      }
82      else
83      {
84          // Initialize source buffer in Host Mem
85          // printf ("Initializing source at Linear 0x%08x  DMAsize=0x%x  pattern=ADDR_PATTERN\n", hostLin, DMAsize);
86          for (DWORD i = 0 ; i < DMAsize/4 ; i++)
87              hostLin[i] = i;
88
89          // Write pattern to destination, so this test can't pass unless 3042 DMA controller actually moves the data.
90          initSharedMem (SMstart, DMAsize, FILL_PATTERN, comemID);
91      }
92
93      // Set 3042's Latency Timer to the max (at the expense of potentially causing latency issues
94      // for other PCI devices).
95      // You'll definitely want to move this to an initialization routine to do it only once.
96      PCI_CONFIG_HEADER_0  temp;
97      returnCode = comemGetPCIInfoL(&temp, comemID);
98      temp.LatencyTimer = 0xff;
99      returnCode = comemSetPCIInfoL(&temp, comemID);
100
101     // Set up to calculate the time and MB/sec ('bandwidth') for the DMA transfer.  Use the built-in
102     // Pentium high-performance timer.
103     LARGE_INTEGER llnHPTimerFreq;    // High Performance Timer: Frequency
104     LARGE_INTEGER llnHPT1;           // High Performance Timer: Time 1
105     LARGE_INTEGER llnHPT2;           // High Performance Timer: Time 2
106     LARGE_INTEGER llnT_uSec;         // Time in microseconds
107     QueryPerformanceFrequency(&llnHPTimerFreq);
108     QueryPerformanceCounter(&llnHPT1);
109
110     // Note: 'Host Mem' can be PCI Memory (physical) addresses other than those assigned to Host Memory.
111     // In this example, we use Host Mem as the source and destination.  But the 3042 does
112     // support peer-to-peer DMA transfers.  Get the physical addresses of the peer PCI devices from their BARs.
113     errorCnt += doDMA ( hostPhys,        // host-side physical address
114                         SMstart,         // Shared Mem start (first byte of shared mem = address 0)
```

```
115                         DMAsize,         // size of DMA transfer (in bytes)
116                         direction,       // the direction of the transfer
117                         comemID);        // which 3042 are we driving
118
119     if (QueryPerformanceCounter(&llnHPT2))
120     {
121         llnT_uSec.QuadPart = ( (llnHPT2.QuadPart - llnHPT1.QuadPart) *      // Delta time in ticks.
122                                ((LONGLONG)1E9/llnHPTimerFreq.QuadPart)      // Adjust for the frequency.
123                              ) / (LONGLONG)1E3;
124
125         double bandwidth = (double)1E6 * (double)DMAsize / (double)llnT_uSec.LowPart;
126
127         printf("DMA completed in  %3d sec  %03d msec  %03d usec: DMA Rate = %11.2le bytes/sec\n",
128                 llnT_uSec.LowPart/(LONG)1E6,
129                (llnT_uSec.LowPart/(LONG)1E3) % 1000,
130                 llnT_uSec.LowPart            % 1000,
131                 bandwidth
132               );
133     }
134
135     if (errorCnt > 0)
136     {
137         printf ("Error: doDMA call failed hostPhys=%08x SMstart=%08x DMAsize=%08x direction=%d comemID=%d\n",
138                                           hostPhys,     SMstart,     DMAsize,     direction,   comemID);
139         // If an error, is probably fatal, but we'll attempt to forge ahead anyway.
140     }
141
142     // Check the destination.
143     if (direction == directionToHost)
144     {
145         for (i = 0 ; i < DMAsize/4 ; i++)
146         {
147             if (hostLin[i] != i)
148             {
149                 if (errorCnt++ < 8)  // Report first 8 errors only.
150                     printf ("Error: PCIAddr=%08x LocalAddr=%08x Wrote=%08x Read=%08x\n",
151                                       hostLin+i,     i,         i,        hostLin[i]);
152             }
153         }
154         if (errorCnt > 0)
155             printf ("%d errors total\n", errorCnt);
156     }
157     else
158     {
159         errorCnt += checkSharedMem (SMstart, DMAsize, ADDR_PATTERN, comemID);
160     }
161
162     // Deallocate the 4 Pages we allocated above.
163     comemDeAllocContigMemL(&lin, &phys, comemID);
164
165     return(errorCnt);
166 }   // end testDMA//
```

Listing 2: doDMA

```
// doDMA
//
// Transfer up to 16KB via the 3042's DMA controller.
//
// Although in this example, we transfer between Host Mem and the 3042's Shared Mem, we could
// use doDMA *unchanged* to do peer-to-peer transfers between devices on the PCI bus.
//
DWORD doDMA (DWORD * PCIphysMem, // The PCI (physical) Memory space address (as opposed to I/O or Config space addresses).
                                 // The 3042 needs this because it knows nothing (directly) about Linear addresses.
            DWORD SMstart,       // Shared Mem start.  Is 0 to point to the zeroth DWORD of 3042 Shared Mem.
            DWORD length,        // Length in bytes.  Gets converted to DWORDs below.
            DWORD direction,     // To or From Host Mem.
            DWORD comemID)       // Which 3042 we're driving.
{
    DWORD errorCnt = 0;

    // Set the starting Shared Mem address in DMALBASE register.
    g_regp->dmalbase = SMstart;     // Give it a byte address, but since DMALBASE bits 0 and 1 are dead,
                                    // this loads a DWORD address.  The L in DMALBASE means Local, as in the
                                    // Local-side interface.

    // Set the DMASIZE.
    g_regp->dmasize = length - 4;   // Give it a byte size, but since DMASIZE bits 0 and 1 are dead,
                                    // actually loads a DWORD size.  Also, the value written to the DMASIZE register
                                    // must be reduced by one DWORD (4 bytes).

    // Set the DMAHBASE to the PCI (physical) Memory address
    g_regp->dmahbase = (DWORD) PCIphysMem;  // Once again, we give it a byte address, but because bits 0 and 1 are dead
                                            // loads a DWORD address.

    //  Clear the DMA complete bit.
    g_regp->hint &= DMA_COMPLETE_BIT;

    // Write low-order byte of DMACTL to set the direction, and to KICK OFF the DMA.
    // The only operations that MUST be done on each DMA transfer are the kick off and the wait for DMA_COMPLETE_BIT.
    g_regp->dmactl_v.dmactl_bytes[0] = (unsigned char) direction;

    // Now the 3042 DMA masters the PCI bus and moves the data.
    // Looking at the PCI bus, you'd see mostly bursts of data moved by the DMA, with occasional polls of HINT.
    // The length of the bursts is determined by the value in the 3042's Latency Timer,
    // and other PCI bus activity (including our HINT polling).

    // Wait for the DMA complete bit to be set.
    //
    // We also want to check for a timeout here, because we don't want our thread to hang with no notification
    // if something goes wrong.
    const DWORD loopsPerByte = 0x100;        // Number of times through this 'while' loop per byte of DMA.
    int timeout = length * loopsPerByte;    // Give lots of time to complete.
    while ( ((g_regp->hint & DMA_COMPLETE_BIT) != DMA_COMPLETE_BIT)
            && timeout
          )
    {
        // Reduce the frequency of HINT polling to improve performance, but poll often enough
        // that a poll occurs soon after the end of the DMA transfer (ideally just after the end).
        //
        // The optimal delay to be inserted here is dependent on:
        //   - your processor speed
        //   - the length of DMA transfer
        //   - characteristics of the PCI bus arbiter, the Latency Timer setting, and other PCI bus traffic
        //   - etc., etc.
       // A loop count of 150 works well for the 16 KB blocks, 200 MHz Pentium Pro and 440FX Chipset used to tune this example.
        for (volatile DWORD i = 0 ; i < 150 ; i++)
            ;

        timeout--;
    }

    if (timeout <= 0)
```

```
69      {
70          errorCnt++;
71          printf ("Error: timed out before DMA_COMPLETE_BIT set. errorCnt=%d\n", errorCnt);
72      }
73
74      return (errorCnt);
75  }   // end doDMA
```

Recommended Tools

- Microsoft Visual C++ 6.x. We do not intend to support older versions of VC++.
- SoftICE by Compuware® NuMega at www.numega.com .

  We've found this tool useful, especially for the following commands:

  — pci – dumps Config Regs for all PCI devices. Helps you find out if the system even sees your card (do the BARs have reasonable values?, is Bus Mastering enabled?). Many other uses.

  — peekd, poked – reads and writes to PCI devices (Physical addresses). This is a good way to bypass software that is under development and talk directly to the PCI-DP.

  — phys – displays all Linear Addresses for a given Physical Address. Is a good sanity check on the Linear Addresses returned by routines such as comemCopyBarPtrL.
- A logic analyzer to look at PCI bus transactions, preferably with PCI transaction formatting software. We currently use an HP 1660C with FuturePlus FSPCI64E support hardware and software. Although there is some elegance in dedicated PCI analyzers (VMetro, HP and others), we have found the HP 1660C and FuturePlus cost effective and functionally adequate.
- A reasonable library. See References.

Glossary

Linear address – addresses on the processor (hence program) side of the Paging Unit. See References 2, 5, and 6. Also known as User Space or Virtual Address. For the converse, see Physical address.

Memory Space – A transaction on the PCI bus can access one of three independent address spaces:

- Configuration (Config) Space – accesses the Config Regs in any device
- Memory Space – used for most transactions after the system is configured. Supports bursting.
- I/O Space – provided only for backward compatibility with x86 processor I/O space.

Because most PCI devices do not support bursting for transactions in I/O Space, it should be used only where absolutely necessary – to support legacy devices. Legacy devices (such as VGA cards) perform many operations using registers mapped into I/O space.

Note that it is possible for a Memory Space transaction to not access Host Memory. A Memory Space transaction can access Host Memory, or can access any other memory or register on the PCI bus that is mapped via the BARs into Memory Space.

A Master tells the Target which Space it wishes to access via the Command emitted by the bus master during the Address Phase (see p. 21 of Reference 3), and by the LS bit (the 'space indicator' bit) of each BAR (0 = Memory, 1 = I/O).

See Physical Address.

Paging Unit – The collection of Pentium components that perform Linear to Physical and Physical to Linear address translation. The 'Paging Unit' consists of:

- the Page Directory
- Page Tables
- assorted bits in Pentium Control Registers (CRs) that determine various paging modes
- Windows kernel code that manages the assorted tables and bits.

The CRs reside in the core of the Pentium. The Tables reside in Host Memory, but the most recently used entries are cached in the core of the Pentium (in the TLBs – Translation Lookaside Buffers).

The Paging Unit also handles:

- Demand Paging, which allows code and data to be swapped into and out of Host Memory from a disk swap file as needed (on demand). See Chapter 13 of Reference 6.
- Protection, which allows a Process to own a set of pages, prohibiting other code from accessing those pages.

Pages are often locked down by the Paging Unit, so they do not move in Physical Space or get swapped to disk. For example, pages associated with PCI devices and the 16 KB buffers allocated by comemL.vxd or comemL.sys get locked down.

Physical address – addresses as seen by hardware, on the hardware side of the Paging Unit. For our purposes this hardware is all PCI devices and host memory. On the PCI bus, a Physical Address is identical to a PCI Memory Space Address (as opposed to an I/O Space Address or a Configuration Space Address). For the converse, see Linear address.

Process – a program (application) that is protected in its own memory space. Normally when you launch an application program, Windows creates a new Process for that program. The new Process can only access memory in its own memory space, or in the operating system through calls to the Win32 API (Application Program Interface).

Rings – Access and instruction execution privilege (protection) levels controlled by Pentium processors.

- Ring 0 – Most privileged protection level on a Pentium. Also known as System Level or Supervisor Level. Code running at this level (such as comemL.vxd) can execute any instruction and access any memory or I/O location, including the page tables.
- Ring 3 – Least privileged protection level on a Pentium. Also known as User Mode or Application Level.
- Ring 1 and 2 – though built into Pentium processors, Ring 1 and 2 are not used by Windows operating systems.

Thread – a single uninterrupted flow of execution through a program.

A Process owns one or more Threads. When created, a Process owns only one Thread. This parent thread may then explicitly create ("spawn") more Threads as it runs by calling OS services. The Process would then *appear* to be doing more than one thing at a time.

Another key component in this slight of hand is the Windows Scheduler. The Scheduler time slices between all the active Threads in the system. It gets invoked periodically when timer or other interrupts occur. For example, you could write a de-

vice driver that would invoke the scheduler whenever the PCI-DP DMA completion interrupt fires.

In general, the Scheduler selects the highest priority Thread from the current list of ready-to-run Threads.

References

1. *CY7C09449PV-AC 128K Bit Dual-Port SRAM with PCI Bus Controller,* Cypress Semiconductor Corperation.
2. Intel Architecture Software Developer's Manual *Volume 3: System Programming Guide*. Intel Corporation, 1997.

   The Intel Architecture Software Developer's Manual consists of three books and three addenda, all available at http://developer.intel.com/design/pentium/manuals :

   Volume 1: Basic Architecture
   Order Number 243190

   Volume 2: Instruction Set Reference Manual
   Order Number 243191

   Volume 3: System Programming Guide
   Order Number 243192

   Addendum: Volume 1: Basic Architecture
   Order Number 243691

   Addendum: Volume 2: Instruction Set Reference Manual
   Order Number 243689

   Addendum: Volume 3: System Programming Guide
   Order Number 243690.
3. *PCI Local Bus Specification, Revision 2.2.* PCI Special Interest Group, Portland, OR.
4. Shanley, Tom, Anderson, Don. *PCI System Architecture, Fourth Edition.* MindShare, Inc., 1999. Addison-Wesley.
5. Hazzah, Karen. *Writing Windows VxDs and Device Drivers.* R&D Books, Lawrence, KS, 1997.
6. Shanley, Tom. *Protected Mode Software Architecture.* MindShare, Inc., 1996. Addison-Wesley.