

## To bridge or not to bridge, that is the question

By Matthew Dharm

*The decision to use (or not use) a PCI-to-CompactPCI bridge on a CompactPCI board is one that is often undertaken without a full understanding of the pros and cons of the choice. Often, it is undertaken without even a full understanding of the different types of bridges and what benefits and limitations each offers. This selection has significant impacts on the overall functionality of the design as well as development and testing time, especially in the software domain. Too often, the primary focus of a board-level design is the processing capabilities (MHz, MIPS, and FLOPS), while the magnitude of the software development for the application is all but forgotten.*

*The goal of this article is to identify and discuss some of the relevant issues that need to be considered when making this decision in the context of a design for a system/non-system CompactPCI processor board. Many factors, both hardware and software related affect this decision. In the hardware domain, issues such as loading, placement, and hot-swap capabilities of parts place significant demands on the design and are well-known factors. However, this article will focus primarily on the less well-known software factors, as these are the most commonly overlooked during the hardware design phase. If these factors are properly considered in the context of the design requirements, development and debug time for both the hardware and software phases of the project can be significantly reduced.*

### PCI Auto-Configuration

To appreciate the issues that software brings to the table, it is necessary to understand one of the primary pieces of PCI software in existence, PCI Auto-Configuration. This software functionality is incorporated into every major OS and ROM/BIOS, and in some cases is so tightly integrated that it is virtually impossible to remove (especially in the case of operating systems which make extensive use of virtual memory functions present in modern processors).

The primary function of this feature is to configure the Base Address Registers (BARs) of each device on the PCI bus automatically and correctly, including all transparent PCI-PCI bridges. A *correct* configuration is one where every device is allocated resources, and no two devices have been allocated conflicting resources. The algorithm used generally guarantees that no resource conflict exists, provided that only one agent is configuring the bus. This guarantee can reduce debugging time significantly.

The use of this feature can drastically reduce development and testing time, especially for systems with multiple configurations of devices on the PCI bus. As important as this feature is, it's important to note that it can be disabled (although with some difficulty, depending on the operating system in use). In that case, address allocation and bus configuration must be accomplished by some other specialized code. However, this is considered

undesirable as it increases the load on development, increases debugging time, and increases time to market.

### Bus scanning

PCI Auto-Configuration is accomplished by using configuration transactions to locate and identify peripherals and bridges (bridges are identified by their PCI Class code). Each bridge is assigned a bus number and address range for the secondary bus, and a recursive algorithm is used to support topologies with an arbitrary depth. On the secondary bus, the address range is assigned to the various peripherals and sub-buses if any are present. The process continues until all devices are configured.

The scanning process is accomplished via *configuration transactions* that are known as *Type 0* and *Type 1* transactions. A Type 0 transaction (that relies on the IDSEL pin of a PCI device being connected properly) is used to configure devices on the local PCI bus segment. A Type 1 configuration transaction is used to configure devices that are not on the local PCI bus segment. Transparent bridges automatically forward (and convert to Type 0, if necessary) these types of cycles. Scanning is accomplished by attempting to read the *Device ID* and *Vendor ID* of devices at every configuration address (slot number). An answered request indicates the presence of a device.

It is important to note that this process relies on two simple, yet significant factors:

1. Transparent bridges provide a standardized interface to the system software, allowing full interoperability between implementations of software and specific parts.
2. The CPU can probe and identify all devices via configuration transactions.

As will be seen later, both of these facts will come into play when considering how various board architectures react to this common software module.

### Address space management

Address space allocation is a significant portion of PCI Auto-Configuration. While many devices support 64-bit addressing, the vast majority of operating systems and applications can only handle 32-bit addressing (4 Gbyte). Typically, the first allocation is made to the local system memory, meaning the address space starts off very crowded with main memory possibly occupying 1-2 Gbytes of space. This is 25 to 50 percent of the available address range. Most implementations of this feature take care to make efficient use of the remaining address space, although exact algorithms to accomplish this differ between implementations. The PCI Local Bus specification goes into more detail about how a device indicates to a configuration agent the amount of space required.

It may sound gratuitous to make all memory addressable as a PCI target, but most software applications assume that

the majority of memory is reachable by bus-master capable peripherals. However, not all memory must be accessible simultaneously. The actual requirement is that when a given PCI peripheral is initialized, the memory that is allocated for the peripheral (generally from a dynamic memory pool) can be located anywhere in memory (so as to place few restrictions on the virtual memory implementation). Once configured, the peripheral only needs to be able to access that allocated memory range for normal operation, a significant distinction since it means that a peripheral board only needs to access a small window of memory on the host. In the case of an intelligent peripheral card, the window of memory need only occupy a small amount of local address space.

### **Bridges: Transparent, non-transparent, and universal**

PCI-to-PCI bridges come in three major varieties:

- Transparent
- Non-transparent
- Universal

Parts that implement any of these types can also support the electrical requirements of a CompactPCI application, including 33/66/100/133 MHz operation, 32/64-bit, conventional PCI vs. PCI-X, and Hot Swap. Thus, the designer is not inherently forced to select a specific type of bridge for a CompactPCI application.

#### **Transparent bridges**

This type of bridge is commonly found on non-intelligent peripheral cards (with multiple PCI chips behind the bridge) or on x86 system slot boards. It provides two asymmetric PCI interfaces (known as *primary* and *secondary*) that both use the same address space. This type of bridge will forward memory and I/O transactions between the interfaces and can be configured with one range for memory, prefetchable memory, and I/O addresses on each side of the bridge. The Intel 21154 is a typical example of this type of bridge.

The asymmetry of the bridge is that it does not respond to configuration transactions on the secondary interface (usually the part does not even have an IDSEL pin on the secondary interface). This bridge automatically forwards Type 1 and Type 0 configuration transactions as appropriate (secondary and subordinate bus numbers are programmed by software during scanning), and presents a standardized software configuration interface of its own.

#### **Non-transparent bridges**

This type of bridge is often found on *intelligent peripheral* cards (cards with a local CPU and memory pool). The non-transparent bridge shares very little with the transparent variety beyond the basic electrical interface. It also provides two PCI interfaces, but these interfaces use independent address spaces. A non-transparent bridge will forward memory and I/O transactions between the two interfaces as appropriate (often these parts provide some limited number of *windows* of addresses for this), and will provide address translation for each transaction. The Intel 21555 is a typical example of this type of bridge.

The exact nature of the windows and address translation will vary from part to part. Generally, however, some local address range on one side of the bridge is *mapped* onto an equal-sized address range on the other side. While the sizes of the windows are often limited by specific implementation, they can generally map from any address to any other address, fulfilling the requirement that an intelligent peripheral be able to access any section of host memory.

This type of bridge will typically respond to configuration transactions on both the primary and secondary interface, but will not forward Type 1 transactions. This prevents a configuration agent on one side of the bridge from being able to detect PCI devices on the other side. In other words, the bridge is not transparent to configuration transactions, and this creates a separation of PCI buses that is often useful to system design.

The part will generate Type 0 and Type 1 transactions on either interface, but this requires a specialized action on the part of software drivers. There is no standardized software configuration interface for non-transparent bridges. Note that since configuration transactions are not automatically forwarded and there is no standardized software interface, PCI Auto-Configuration software cannot probe and/or configure through one of these bridges. Thus, some extra software must be written to configure this type of bridge (and devices beyond it, if applicable).

#### **Universal**

The universal bridge is perhaps the least-often used type of bridge, but it is also the most feature-rich type of bridge. The universal bridge (not to be confused with universal voltage parts) can act as a transparent or non-transparent bridge depend-

ing on reset-time configuration. Most commonly for intelligent boards, transparent mode is used in the system slot and non-transparent mode is used in the non-system slot. The PLX HB8 is a typical example of this type of bridge.

#### **Sample application**

The application we are considering is a CompactPCI board, much like the typical high-performance processor boards used in many of today's real-world applications. The requirements of this design are:

- The board must be functional in any slot of the chassis, and the desire is to be able to use this design as either a *host* board, as an intelligent peripheral, or co-processor.
- The final chassis configuration needs to support at least three boards of this design, with the ability to support other standard CompactPCI peripheral cards.
- The software should be as similar as possible across the boards. Optimally, the same software will be used for each board, and that software should use as many standard components (as opposed to specialized components) as possible.

For the sake of simplicity, the design has been reduced to three major components shown in Figure 1. These components are:

- Processor
- Memory
- Host Bridge

While each of these elements is often actually two or more parts, for the sake of discussion they are simplified. The exact type of CPU or Host Bridge isn't relevant, but assume that 2 Gbytes of memory are to be available to all PCI peripherals when the board is used as a host.

#### **Bridgeless design**

First consider a design that does not use a PCI bridge at all, as seen in Figure 1. A design of this type often offers a cost advantage over a design that uses a bridge. This design requires that the host bridge meet the electrical requirements of the interface (as mentioned earlier), functionality that is not always available for a given part. Also, the host bridge may or may not accept configuration transactions from the PCI interface (for example, the Intel P64H2 does not have an IDSEL pin, while the Marvell MV-64360 does). This immediately creates a situation where the designer may be forced to tradeoff desirable host bridge features and functions in

# SPECIAL FEATURE: BRIDGING

order to select a part that meets the electrical requirements. This is a tradeoff that could seriously compromise the performance of the finished product.

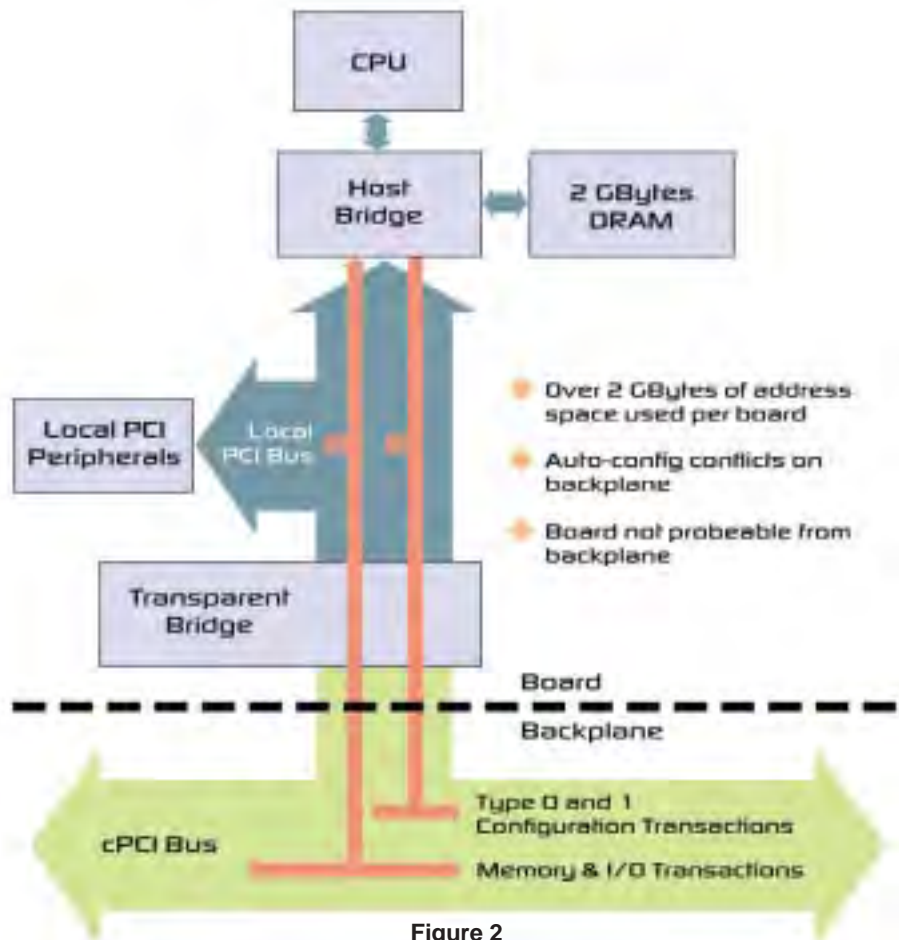
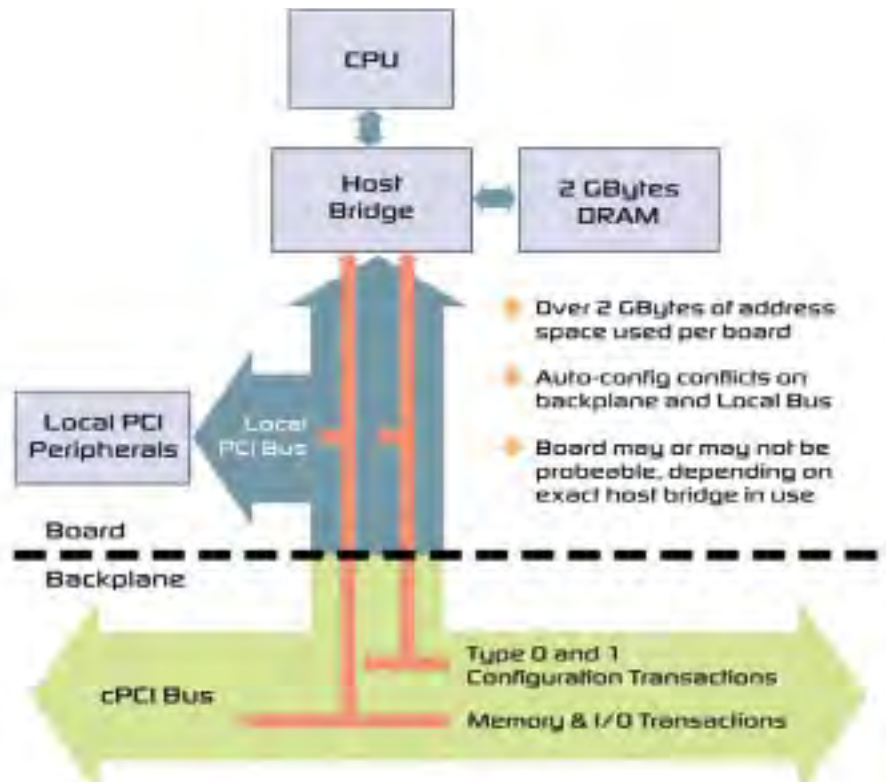
Consider a configuration that uses multiple boards of this design. It immediately becomes evident that an addressing problem exists on the backplane. There is a total of 6 Gbytes of memory attached to the CompactPCI bus, a capability that is well beyond the 32-bit address space. If this configuration used an implementation of PCI Auto-Configuration, the algorithm would fail when address resources were exhausted. In addition, the board software must be intelligent enough so that only one board in a chassis performs the auto-configuration algorithm. Otherwise, multiple instances of the feature will fight each other.

Software customization to this level is contrary to design goals. Even if the cost of the board is reduced, the shortsighted nature of the design selection virtually guarantees that the software development for the operating system and application will be long, difficult, and full of bugs. The inability to use a standard PCI Auto-Configuration module virtually guarantees that the design will be *brittle* (easily broken by slight changes) and inflexible (only functional for the specific design parameters given).

## Transparent design

The second design to consider (Figure 2) utilizes a transparent bridge to connect the local PCI bus on the board to the CompactPCI backplane. The transparent bridge provides the electrical characteristics required to implement Hot Swap, thus freeing the rest of the design from this requirement. This allows more freedom for the designer to select the most appropriate parts for a given design. On non-intelligent peripheral cards (Ethernet, SCSI, or other non-CPU carrying boards), the bridge is generally oriented with the primary interface connected to the CompactPCI bus. On an intelligent card (used as either a host or intelligent peripheral), the bridge is most commonly oriented with the primary interface connected to the local host bridge on the board.

In this configuration however, the boards used as peripherals can no longer be seen by the host since the transparent bridge does not accept configuration transac-



# SPECIAL FEATURE: BRIDGING

tions from the secondary interface that is connected to the CompactPCI bus. This makes PCI Auto-Configuration useless, as each board can only see the standard CompactPCI peripherals (not each other).

In this configuration, PCI Auto-Configuration is actually harmful. With this design, the requirement that only a single agent attempt to configure the bus has been violated, leading to resource conflicts if each board attempts to configure standard peripherals on the CompactPCI bus. Avoiding this was one of the primary goals of the auto-configuration feature. Thus, PCI Auto-Configuration must be disabled and that is potentially very difficult.

Local PCI peripherals still need to be configured however, adding to the software burden. Due to this fact, each board must be individually configured to use a specific address range to prevent conflict on the bus. This design requires extensive specialized software and introduces potential compatibility problems with other peripherals.

### Non-transparent design

The third design to consider (Figure 3) utilizes a non-transparent bridge to connect the local PCI bus to the CompactPCI backplane. Just like the transparent bridge, the non-transparent bridge provides the electrical features required for implementing Hot-Swap capability, and the designer is again free to make the most appropriate part selection for the design. In general, a non-transparent bridge used in this manner is oriented with the primary interface connected to the CompactPCI bus.

This type of bridge can be detected and configured from either interface, but since the non-transparent bridge does not pass Type 1 configuration cycles automatically, there is no automatic configuration of the backplane bus from the host (no conflict between agents attempting to configure the bus exist too). This allows the use of a standard PCI Auto-Configuration software implementation on each board, but creates the additional burden of developing a PCI configuration system to configure the backplane from one (and only one) of the boards, usually the system slot board. This customized software is significantly less extensive than some of the other customized software mentioned in this document, as there is no need to disable or bypass any existing functionality.

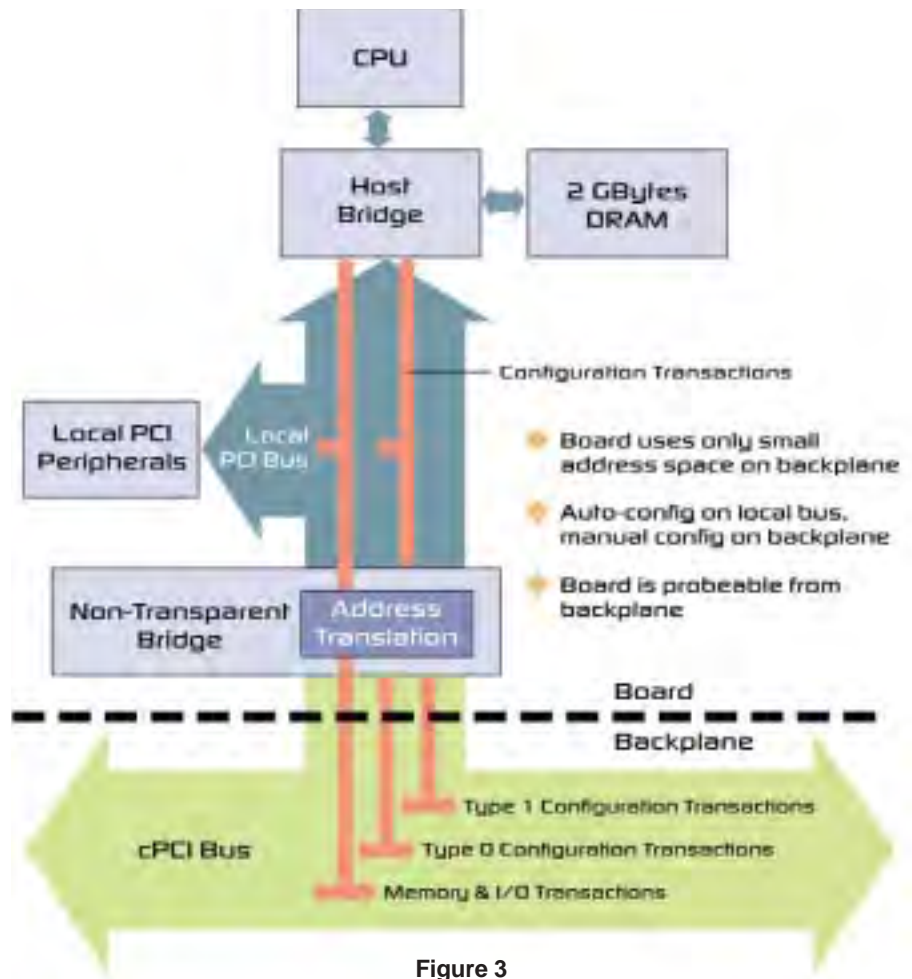


Figure 3

Since each bridge in the chassis can provide address translation, address space conflicts are avoided. The backplane address space can be configured into any one of a number of reasonable configurations that allow all peripherals to access all of the host main memory, thus fulfilling the requirements stated earlier. Commonly, the host will map all of memory into the CompactPCI address space, while each peripheral will map a small window of CompactPCI address into the local PCI addresses space.

### Universal design

The final design to consider utilizes a universal bridge. Since this bridge has two radically different modes of operation, it can bring the benefits of both the transparent and non-transparent bridge to the board design and chassis operation.

As a system-slot board (Figure 4), the universal bridge acts as a transparent bridge. Thus, the backplane shares the same address space as the system-slot board and configuration transactions are

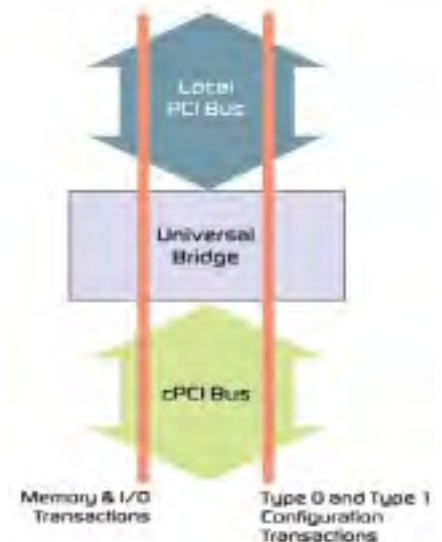


Figure 4

passed through the bridge with no address translation. This allows a standard PCI Auto-Configuration implementation to function for the local PCI bus as well as the CompactPCI bus. Peripherals have access to host memory, as required, and

standard CompactPCI peripherals will function with this design.

As a non-system slot (peripheral) board, the universal bridge acts as a non-transparent bridge. This *protects* the local PCI bus from address space conflicts with the CompactPCI bus, while allowing a standard PCI Auto-Configuration implementation to be used. A small piece of specialized software is required to map CompactPCI addresses into the local PCI space, but this is the smallest piece of custom software of any that have been discussed so far.

### Conclusion

In the designs that have been examined, one of them required the major task of removing a feature that is central to many modern operating systems. One of them required the addition of a moderate amount of software, and the final required only a minimum of software. Neither of the final two designs required major modification to the operating system and given this discovery, it is clear what designs are more favorable than others.

The design goals of the hypothetical applications explored in this article are taken from typical applications of CompactPCI SBCs. These include network packet processing for a variety of tasks (NAT/PAT, encryption, compression, routing, etc.), data co-processing (audio/video compression and decompression), or even the creation of test equipment (multi-source network flow generation and analysis). In

many of these applications, both scalability and compatibility with third-party add-on devices is a hard requirement.

These types of widely varying environments for an SBC dictate that a board design must be flexible and functional with as little trouble for the application designer. Using common software modules to accomplish relatively standard tasks efficiently and correctly (like PCI Auto-Configuration) promotes this ideal. Using a board design that makes the use of this software both practical and easy completes the goal, creating a board-level product that not only delivers the processing capabilities desired, but also allows relatively easy development of the application software.



**Matthew Dharm** is a senior software designer at Momentum Computer, Inc., which specializes in quick turnaround on design and delivery of application specific processor and high performance I/O boards. Matthew has been actively involved in the development of embedded systems and open standards since 1998. He holds a B.S. in computer science with distinction from Harvey Mudd College. Matthew is also the author and maintainer of the Linux USB Mass Storage driver.

For further information, contact Matthew at:

**Momentum Computer**  
1815 Aston Ave.  
Suite 107  
Carlsbad, CA 92008-7310  
Tel: 760-431-8663  
E-mail: mdharm@momenco.com  
Web site: www.momenco.com