MSC8144 PCI Example Software

by  Barbara Johnson  
NCSD DSP Applications  
Freescale Semiconductor, Inc.  
Austin, TX

In a PCI system, auto-configuration software offers ease of use for the system user by automatically configuring PCI add-in cards at power-on. This application note provides example software for use by a PCI host to configure the MSC8144 DSP as a PCI agent. This device configuration is required before any PCI transactions can occur between the host and the MSC8144.

1 PCI Basics

Peripheral component interconnect (PCI) is a standard that provides an interconnect mechanism between peripheral components, add-on devices, and memory subsystems. Developed by Intel, PCI is widely used in modern PCs to provide a way of adding peripherals such as video cards, sound cards, and network adapters on the same bus that is used to communicate with the CPU. Figure 1 shows an example PCI-based system. The CPU connects to the primary PCI bus on a PCI host bridge that translates between CPU bus cycles and PCI bus cycles. The PCI-PCI bridge connects the primary PCI bus to the secondary PCI bus. Electrical loading issues limit the number of devices that a single PCI bus can support, so PCI-PCI bridges are often used to allow the system to support more PCI devices. To support older, legacy devices, some PCs use a PCI-ISA bridge for connecting to the PCI bus.

© Freescale Semiconductor, Inc., 2006. All rights reserved.
In Figure 1, the CPU operates as the PCI host. In the host mode, the CPU configures the PCI devices attached to the bus. Because PCI uses a shared bus topology, there must be an arbitration scheme to grant bus mastership to the requesting PCI device. Arbitration is handled by the host or an external arbiter.

From the host perspective, the PCI devices are accessible through a read-write mechanism. An address space dedicated for PCI use contains a memory range for each PCI device on the bus. The host accesses the PCI devices by performing reads or writes to specific addresses in the PCI memory space, as shown in Figure 2.

Devices on the PCI bus must be configured before they can be used. For example, when a PC first boots up, each PCI device is assigned a region of PCI address space so that it becomes accessible to the CPU. After the devices are initialized, they respond to transactions that fall within their allocated memory ranges.
2 MSC8144 PCI Controller

The MSC8144 PCI controller complies with the *PCI Local Bus Specification*, Revision 2.2. It operates in agent mode and can act as initiator (master) or target (slave) device. It uses a 32-bit multiplexed address/data bus that operates at frequencies up to 66 MHz. Features of the PCI controller are as follows:

- 32-bit PCI interface
- Up to 66 MHz operation
- Agent mode
- Accesses to all PCI address spaces
- 64-bit dual-address cycles (as a target only)
- Internal configuration registers accessible from PCI and internal buses
- Contains L2 ICache-line (32 byte) buffers to allow PCI-to-memory and memory-to-PCI streaming
- Memory prefetching of PCI read accesses and support for delayed read transactions
- Posting of processor-to-PCI and PCI-to-memory writes
- Inbound and outbound address translation units for address mapping between PCI and local busses
- Supports parity
- PCI 3.3-V compatible

3 Hardware Requirements and Setup

All tests described in this document were performed on the MSC8144 application development system (MSC8144ADS), which consists of an MPC8560 host processor that connects to the MSC8144 on the PCI bus.

3.1 Requirements

The following items are required to run the examples presented in this document:

- MSC8144ADS board
- PC with CodeWarrior™ for StarCore version 3.2 or later
- PC with CodeWarrior for PowerPC™ version 8.7 or later
- USBTap for MSC8144 OCE connection
- USBTap for MPC8560 COP connection

Two sets of debugger tools are required to connect to the MSC8144 and the MPC8560. A USBTap for OCE connects the MSC8144 to the CodeWarrior for StarCore tools through JTAG. Similarly, a USBTap for COP connects the MPC8560 to the CodeWarrior for PowerQUICC tools through JTAG. Figure 3 shows the hardware setup.
3.2 Switch Settings

Table 1 shows the switch settings required to connect the MSC8144 and the MPC8560 to the debugger tools. Refer to the MSC8144 ADS User’s Manual for details about the switch settings.

![Figure 3. Hardware Setup](image)

### Table 1. MSC8144ADS Switch Settings

<table>
<thead>
<tr>
<th>Switch</th>
<th>Settings 1:8</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SW1</td>
<td>00000110</td>
<td>Default setting.</td>
</tr>
<tr>
<td>SW2</td>
<td>01101111</td>
<td>Host MPC8560 operates normally; disable JTAG chain for MSC8144 and MPC8560.</td>
</tr>
<tr>
<td>SW3</td>
<td>10010111</td>
<td>Default setting.</td>
</tr>
<tr>
<td>SW4</td>
<td>01100010</td>
<td>Default setting.</td>
</tr>
</tbody>
</table>

**Note:** 0 = ON, 1 = OFF

3.3 Board Control and Status Register Setting

The board control and status registers (BCSRx) are a group of 8-bit read/write registers that control or monitor most ADS hardware options. These registers can be accessed from the host local bus. To enable the PCI interface on the MSC8144ADS, bits 2 and 3 of BCSR1 must by set as shown in Example 1. For details on the BCSRx bit definitions, refer to the MSC8144ADS User’s Manual.

#### Example 1. Enable PCI in BCSR1

```c
void setBCSR1()
{
    uint8_t *bcsr1 = (uint8_t*)0xF8000001;
    *bcsr1 |= 0x30;
}
```

4 PCI Device Detection Example

Before the PCI host can configure each device on the bus, it must first scan the bus to determine what PCI devices or PCI-PCI bridges are on the bus. By scanning the bus, the host can determine each device part number, manufacturer, and device number on the bus. The PCI specification requires each PCI device to provide 256 bytes of configuration registers. The configuration registers supply the information needed for device configuration, including the vendor ID, device ID, command and status, revision ID, class code and header type fields, as shown in Figure 4.
On the MSC8144ADS, the MPC8560 host can access the MSC8144 configuration space registers through two registers that are memory-mapped in the MPC8560 memory space:

- **CONFIG_ADDR** Specifies the selected device configuration register to be accessed.
- **CONFIG_DATA.** Data is transferred to or from the CONFIG_DATA register.

For example, the host addresses a particular device on the bus by writing to the CONFIG_ADDR with the bus number, device number, and the configuration register to access. Next, the host either writes the CONFIG_DATA with the value to write to the selected configuration register or the host reads the CONFIG_DATA to determine the value of the selected configuration register. Note that in the MPC8560, the CONFIG_ADDR register uses big-endian but the CONFIG_DATA register uses the little-endian convention.

---

**Figure 4. MSC8144 PCI Configuration Space Registers**

- **Device ID**
- **Vendor ID**
- **PCI Status**
- **PCI Command Config**
- **Base Class Code**
- **Subclass Code**
- **Std Prog Interface**
- **Revision ID**
- **BIST Control**
- **Header Type**
- **Latency Timer**
- **Cache Line Size**
- **PIMMR Base Address Register**
- **GPL Base Address Register 0**
- **GPL Base Address Register 1**
- **GPL Extended Base Address Register 1**
- **GPL Base Address Register 2**
- **GPL Extended Base Address Register 2**
- **Reserved**
- **Subsystem Device ID**
- **Subsystem Vendor ID**
- **Reserved**
- **Reserved**
- **Capabilities Pointer**
- **Reserved**
- **PCI Bus MAX LAT**
- **PCI Bus MIN GNT**
- **PCI Bus Interr Pin**
- **PCI Bus Interr Line**
- **Reserved**
- **PCI Function Configuration**

<table>
<thead>
<tr>
<th>Device ID</th>
<th>Vendor ID</th>
<th>Address Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCI Status</td>
<td>PCI Command Config</td>
<td>0x00</td>
</tr>
<tr>
<td>Base Class Code</td>
<td>Subclass Code</td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>BIST Control</td>
<td>Header Type</td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>PIMMR Base Address Register</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>GPL Base Address Register 0</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>GPL Base Address Register 1</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>GPL Extended Base Address Register 1</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>GPL Base Address Register 2</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>GPL Extended Base Address Register 2</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>PCI Bus MAX LAT</td>
<td>PCI Bus MIN GNT</td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>PCI Command Config</td>
</tr>
<tr>
<td>Reserved</td>
<td></td>
<td>PCI Command Config</td>
</tr>
</tbody>
</table>
To scan the bus, the host will try to read the Vendor and Device ID Configuration Registers for all valid device number values. A target is selected during a configuration access when its IDSEL signal is selected. The IDSEL signal acts as the chip select signal. On the MSC8144ADS, the MPC8560’s AD21 pin connects to the MSC8144’s IDSEL pin. This connection means that the MSC8144 has a device number of 21 on bus 0. The MPC8560 has a device number of 0 on bus 0. Selecting a device number other than 0 and 21 will return an invalid value since only the MPC8560 and MSC8144 are present on the PCI bus on the MSC8144ADS. If the device does not exist, the Vendor ID returns a 0xFFFF which indicates an invalid vendor.

Table 2 shows the CONFIG_ADDR and CONFIG_DATA register values that the MPC8560 accesses to read its own and the MSC8144’s vendor and device information.

<table>
<thead>
<tr>
<th>Access</th>
<th>Register</th>
<th>Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CONFIG_ADDR</td>
<td>0x80000000</td>
<td>Write</td>
<td>Allow a PCI configuration access when CONFIG_DATA is accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Select bus number 0</td>
<td>Select device number 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Access the PCI Vendor ID Configuration Register (offset 0x00)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONFIG_DATA</td>
<td>0x1057</td>
<td>Read</td>
<td>Vendor ID = 0x1057 Freescale Semiconductor</td>
</tr>
<tr>
<td>2</td>
<td>CONFIG_ADDR</td>
<td>0x80000002</td>
<td>Write</td>
<td>Allow a PCI configuration access when CONFIG_DATA is accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Select bus number 0</td>
<td>Select device number 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Access the PCI Device ID Configuration Register (offset 0x02)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONFIG_DATA</td>
<td>0x0009</td>
<td>Read</td>
<td>Device ID = 0x0009 MPC8560</td>
</tr>
<tr>
<td>3</td>
<td>CONFIG_ADDR</td>
<td>0x8000A800</td>
<td>Write</td>
<td>Allow a PCI configuration access when CONFIG_DATA is accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Select bus number 0</td>
<td>Select device number 21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Access the PCI Vendor ID Configuration Register (offset 0x00)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONFIG_DATA</td>
<td>0x1957</td>
<td>Read</td>
<td>Vendor ID = 0x1957 Freescale Semiconductor</td>
</tr>
<tr>
<td>4</td>
<td>CONFIG_ADDR</td>
<td>0x8000A802</td>
<td>Write</td>
<td>Allow a PCI configuration access when CONFIG_DATA is accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Select bus number 0</td>
<td>Select device number 21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Access the PCI Device ID Configuration Register (offset 0x02)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONFIG_DATA</td>
<td>0x1400</td>
<td>Read</td>
<td>Device ID = 0x1400 MSC8144</td>
</tr>
</tbody>
</table>
Example 2 shows code that runs on the MPC8560 to scan the PCI bus for devices. This code has been simplified to scan only bus number 0.

```c
void scanDevices()
{
    uint32_t i;
    uint16_t VendorID, DeviceID;
    uint32_t BusNum = 0;
    for(i = 0; i < 0x20; i++)
    {
        VendorID = getPCIConfigReg16(BusNum, i, REG_VENDORID);
        DeviceID = getPCIConfigReg16(BusNum, i, REG_DEVID);
        if(VendorID != 0xFFFF)
            { printf(" Device found: Device %x, Bus %x, DevID = %x, VendorID = %x
", i, BusNum, DeviceID, VendorID); }
    }
}
```

5 Memory Allocation Example

A transaction with the MSC8144 as the target is an inbound transaction. When the MSC8144 boots from PCI, the boot code sets up the three inbound windows for M2, M3, and DDR memory as shown in Figure 5. The MSC8144 PCI boot code configures the base addresses in local memory and the sizes of these inbound windows. The PCI inbound translation address register (PITARn) defines the base address of the inbound translation windows in the MSC8144 memory space. The PCI inbound window attribute register (PIWARn) defines the size of a window as well as other properties and enables that window. It is the host that allocates memory to each device on the PCI bus in a device-independent manner. The host allocates memory by creating a memory map in the PCI memory space. It writes to each device GPL base address register (GPLBARx) in the PCI configuration space to create a mapping between the PCI view and the device local memory view.

```
Figure 5. MSC8144 Inbound Window Configuration at Boot
```

```
```
Memory Allocation Example

To determine the size requirements of the MSC8144 inbound windows, the MPC8560 writes to the MSC8144 GPLBARx in the PCI configuration space as shown in Figure 4. These base address registers specify the mapping of the device inbound windows in the PCI space. The host first reads these base address registers to get the initial setting. It then writes all 1’s to these base address registers and reads them back to determine the memory size required by the inbound window. Then the host can allocate memory for each device in the PCI space.

Table 3 shows the steps by which the MPC8560 determines the MSC8144 inbound window 0 memory requirement. The MPC8560 addresses the MSC8144 GLBAR0 register by writing to the CONFIG_ADDR register. Then it reads the CONFIG_DATA to get the value of the GPLBAR0, which defines the inbound window 0 base address register in the PCI memory space. The host must save this value to restore later. Next, the host writes 0xFFFFFFFF to the MSC8144 GPLBAR0 and reads back the register. The number of bits set determines how much address space is required. For example, a GPLBAR0 value of 0xFFFF8000 has the upper 13 bits of the address register set, indicating a size of $2^{(32-13)} = 512$ Kbytes. Now, the host knows that it must allocate 512 Kbytes of memory in the PCI memory space before it can access the MSC8144 inbound window 0. The host must then assign an address in the PCI space because the GPLBAR0 now contains the sizing information. These steps should be repeated for GPLBAR1 and GPLBAR2 registers to determine the memory requirements for inbound windows 1 and 2.

### Table 3. Memory Allocation by MPC8560

<table>
<thead>
<tr>
<th>Access</th>
<th>Register</th>
<th>Value</th>
<th>Access Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CONFIG_ADDR</td>
<td>0x8000A814</td>
<td>Write</td>
<td>Allow a PCI configuration access when CONFIG_DATA is accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Select bus number 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Select device number 21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Access the GPL Base Address Register 0 (offset 0x14)</td>
</tr>
<tr>
<td></td>
<td>CONFIG_DATA</td>
<td>0x00000000</td>
<td>Read</td>
<td>GPLBAR0 = 0x00000000 inbound window 0 in PCI space</td>
</tr>
<tr>
<td>2</td>
<td>CONFIG_ADDR</td>
<td>0x8000A814</td>
<td>Write</td>
<td>Allow a PCI configuration access when CONFIG_DATA is accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Select bus number 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Select device number 21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Access the GPL Base Address Register 0 (offset 0x14)</td>
</tr>
<tr>
<td></td>
<td>CONFIG_DATA</td>
<td>0xFFFFFFFF</td>
<td>Write</td>
<td>GPLBAR0 = 0xFFFFFFFF</td>
</tr>
<tr>
<td>3</td>
<td>CONFIG_ADDR</td>
<td>0x8000A814</td>
<td>Write</td>
<td>Allow a PCI configuration access when CONFIG_DATA is accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Select bus number 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Select device number 21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Access the GPL Base Address Register 0 (offset 0x14)</td>
</tr>
<tr>
<td></td>
<td>CONFIG_DATA</td>
<td>0xFFFF800000</td>
<td>Read</td>
<td>GPLBAR0 = 0xFFFF80000 Upper 13 bits are set</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Size = $2^{(32-13)} = 512$ Kbytes</td>
</tr>
<tr>
<td>4</td>
<td>CONFIG_ADDR</td>
<td>0x8000A814</td>
<td>Write</td>
<td>Allow a PCI configuration access when CONFIG_DATA is accessed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Select bus number 0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Select device number 21</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Access the GPL Base Address Register 0 (offset 0x14)</td>
</tr>
<tr>
<td></td>
<td>CONFIG_DATA</td>
<td>0xC0000000</td>
<td>Write</td>
<td>Reassign new GPLBAR0 = 0xC0000000</td>
</tr>
</tbody>
</table>
In this example, the host assigns a chunk of memory in the PCI memory space to map the MSC8144 inbound window in M2. The host configures the MSC8144 GPLBAR0 = 0xC0000000 to give a one-to-one mapping between the PCI view and the MSC8144 local view, as shown in Figure 6.

![Figure 6. Example MSC8144 GPLBAR0 Configuration](image)

Example 3 shows the function called by the MPC8560 to calculate the size of a device inbound window.

**Example 3. Determine Memory Requirement**

```c
uint32_t getWindowSize(uint32_t BusNum, uint32_t DevNum, uint32_t Reg)
{
    uint32_t new, orig;
    uint32_t size;

    // Read CfgReg
    orig = getPCIConfigReg32(BusNum, DevNum, Reg);

    // Write all 1’s
    setPCIConfigReg32(BusNum, DevNum, Reg, 0xFFFFFFFF);

    // Read back to determine size
    new = getPCIConfigReg32(BusNum, DevNum, Reg);

    // Restore orig register value
    setPCIConfigReg32(BusNum, DevNum, Reg, orig);

    // Calculate size required by agent
    if (new & 1)
        size = (~new | 3) + 1; // I/O space
    else
        size = (~new | 0xF) + 1; // Memory space

    return size;
}
```

6  MSC8144 Inbound Configuration Example

An inbound transaction in which the MSC8144 is the target means that the host MPC8560 is the bus master or the initiator performing an outbound transaction. Outbound transactions require address translation to map transactions from the internal address space of the MPC8560 to the external PCI address space.
MSC8560 registers to handle the address translation task are as follows:

- PCI outbound translation address register (POTAR\textsubscript{n}) selects the base address of the external PCI address space for hits in the outbound window.
- PCI outbound window base address register (POWBAR\textsubscript{n}) points to the base address of the outbound window in the MPC8560 local address space.
- PCI outbound window attributes register (POWAR\textsubscript{n}) enables the address translation window, specifies the transaction type, and defines the window size.

Table 4 shows the outbound window 1 of size 512 Kbytes starting at 0x80000000 in the MPC8560 local address space for translation to the external PCI address starting at 0xC0000000.

**Table 4. MPC8560 PCI Outbound Register Settings**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POTAR1</td>
<td>0x000C0000</td>
<td>Set base address 0xC0000000 as the translated address in the PCI address space</td>
</tr>
<tr>
<td>POWBAR1</td>
<td>0x00080000</td>
<td>Set base address 0x80000000 as the outbound address from the MPC8560</td>
</tr>
<tr>
<td>POWAR1</td>
<td>0x80044012</td>
<td>Enable the outbound translation window 1</td>
</tr>
</tbody>
</table>

Figure 7 depicts the MPC8560 outbound address translation mapping. An access to the MPC8560 local address starting at 0x80000000 is routed to the PCI memory space starting at 0xC0000000. Notice that the PCI address is configured to be the same as the local address for a simple one-to-one mapping. The left side of the dotted line gives register settings on the MPC8560 side, and the right side of the dotted line gives register settings on the MSC8144 side. Note that the MPC8560 POTAR must have the same value as the MSC8144 GPLBAR.
Before the MSC8144 DSP can respond to memory accesses, the MPC8560 processor must configure the MSC8144 PCI command configuration register, which has an offset of 0x04 in the PCI configuration space. The MEM bit must be set to allow the MSC8144 to respond to memory accesses. Figure 8 shows an example in which the MSC8144 performs an inbound transaction. The left screenshot shows the MPC8560 writing the value 0x11223344 to its local outbound window. This address is translated to the PCI address that is mapped to the MSC8144 local inbound window. The right screenshot shows the MSC8144 reading the same value from its local inbound window.

1. MPC8560 writes data to outbound window

```c
outbound = (uint32_t *)OutboundLocal;
*outbound = 0x11223344;
```

2. MSC8144 reads data from inbound window

```c
data = *inbound;
*outbound = 0xdeadbeef;
```

**Figure 8. MSC8144 Inbound Example**

### 7 MSC8144 CCSR Mapping Example

Before we look at an example MSC8144 outbound transaction, let us first consider how to configure the host MPC8560 to view the MSC8144 configuration control and status registers (CCSR) map. The CCSR address space includes control and status registers for DMA, CLASS, DDR, clock, I²C, timers, TDM, GPIO, PCI, RapidIO, and so on.

External masters do not need the location of a device’s CCSR memory space. Instead, they access a device’s CCSR through a window defined by the PIMMR base address configuration register in the configuration register space, as shown in Figure 4. The PIMMR defines the address for accessing the local CCSR memory space of a device. It specifies an address in the PCI space where the CCSR space is mapped.

The CCSR memory space is 32 Mbytes. Subtracting 32 Mbytes from the top of the MSC8144 address space gives the CCSR base address of 0xFE000000. In this example, the host configures PIMMR to 0x30000000 through the CONFIG_ADDR and CONFIG_DATA configuration access registers. The host also configures another outbound window at 0x90000000 so that transactions from this space are mapped to the PCI space at 0x30000000. Figure 9 shows a diagram of the CCSR address space mapping, and Table 5 shows how the MPC8560 configures the MSC8144 PIMMR base address.

---

**Figure 9. CCSR Address Space Mapping**

**Table 5. PIMMR Base Address Configuration**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONFIG_ADDR</td>
<td>0x30000000</td>
</tr>
<tr>
<td>CONFIG_DATA</td>
<td>0x90000000</td>
</tr>
</tbody>
</table>
For example, if the MPC8560 needs to read the MSC8144 PCI error status register (PCI_ESR), then the MPC8560 accesses the outbound window that maps to the MSC8144 CCSR space. The PCI_ESR is at 0xFFF7A000 in the MSC8144 local space. With a CCSR base address of 0xFE000000, the PCI_ESR has an offset of 0x01F7A000 from the CCSR base. To access this register, the MPC8560 needs to read the following address:

\[ \text{PCI}_{-}\text{ESR} = 0x90000000 + (0xFFF7A000 - 0xFE000000) = 0x91F7A000 \]

As you can see, the PIMMR allows an external master to access a device’s internal memory-mapped registers without knowing where the CCSR resides. This is especially useful because the CCSR base address is programmable so that the PCI host can read and write the MSC8144 memory-mapped registers.
8 MSC8144 Outbound Configuration Example

In Section 5, “Memory Allocation Example,” we noted that the MSC8144 configures three inbound windows for M2, M3, and DDR at bootup. The boot code does not configure the outbound windows, so the host must configure them. Knowing how the MPC8560 can access the MSC8144 memory-mapped registers, we allow the MPC8560 to set up the MSC8144 outbound window using the PIMMR. The MSC8144 defines an outbound memory window in the address range 0xE0000000–0xE7FFFFFF. Both the configuration access registers CONFIG_ADDR and CONFIG_DATA fall within this 128 Mbyte window. If the address is not 0xE7FFFFF0 (CONFIG_ADDR) or 0xE7FFFFF4 (CONFIG_DATA), then the transaction is forwarded to the PCI port.

When the MSC8144 initiates a transaction, the PCI outbound base address register (POBARn) defines the location of the outbound translation in MSC8144 memory space. The POTARn defines the starting-point of the outbound translation address in the destination PCI memory space. The POCMRn defines the size of an outbound translation window, defines properties, and enables that window.

Mapping the MSC8144 outbound window to the PCI space with a one-to-one mapping means that both the POBAR0 and POTAR0 registers are set to map a window at 0xE0000000. The MPC8560 writes to these registers through the PIMMR mapping in the PCI space. The MSC8144 can access the POTAR0 locally at 0xFFF7A100. Based on the settings for mapping the CCSR map, the MPC8560 can access the MSC8144 POTAR0 from its local space at 0x91F7A100. Similarly, it can access the MSC8144 POBAR0 and POCMR0 registers from 0x91F7A108 and 0x91F7A110, respectively.

\begin{equation}
POBAR0 = 0x90000000 + (0xFFF7A108 - 0xFE000000) = 0x91F7A108
\end{equation}

\begin{equation}
POCMR0 = 0x90000000 + (0xFFF7A110 - 0xFE000000) = 0x91F7A110
\end{equation}

\begin{equation}
POTAR0 = 0x90000000 + (0xFFF7A100 - 0xFE000000) = 0x91F7A100
\end{equation}

Suppose the MPC8560 receives inbound transactions at address 0x00000000 so that the mapping is as shown in Figure 10. The MPC8560 configures the MSC8144 outbound window register settings as shown in Table 6. Its inbound window register settings are shown in Table 7.
Figure 10. MSC8144 Outbound / MPC8560 Inbound Address Mapping

Table 6. MSC8144 PCI Outbound Registers Settings

<table>
<thead>
<tr>
<th>MSC8144 Register</th>
<th>Address as Visible to MSC8144</th>
<th>Address as Visible to MPC8560</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POTAR0</td>
<td>0xFFF7A100</td>
<td>0x91F7A100</td>
<td>0x000E0000</td>
<td>Set base address 0xE00000000 as the translated address in the PCI memory space</td>
</tr>
<tr>
<td>POBAR0</td>
<td>0xFFF7A108</td>
<td>0x91F7A108</td>
<td>0x000E0000</td>
<td>Set base address 0xE00000000 as the outbound window 0 in the MSC8144 local memory space</td>
</tr>
</tbody>
</table>
| POCMR0           | 0xFFF7A110                      | 0x91F7A110                    | 0xA0F5501B  | Enable the inbound translation window 0  
|                  |                                 |                               |             | Map window 0 to PCI memory space  
|                  |                                 |                               |             | Enable streaming  
|                  |                                 |                               |             | Set translation window 0 size as 128 MB                                     |

Table 7. MPC8560 PCI Inbound Registers Settings

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PITAR1</td>
<td>0x00000000</td>
<td>Set base address 0x00000000 as the translated address in the local MPC8560 memory space</td>
</tr>
<tr>
<td>PIWBAR1</td>
<td>0x000E0000</td>
<td>Set base address 0xE0000000 as the address in the PCI memory space</td>
</tr>
</tbody>
</table>
| PIWAR1   | 0xA0F5501B  | Enable the inbound translation window 1  
|                  |             | Inbound window 1 is prefetchable  
|                  |             | Target interface is local memory  
|                  |             | Enable read and write with snooping  
|                  |             | Set translation window 1 size as 256 MB                                    |
Before the MSC8144 can initiate PCI accesses, the MPC8560 must configure the MSC8144 PCI bus command register, which has an offset of 0x04 in the PCI configuration space as shown in Figure 4. The BMST bit must be set so that the MSC8144 behaves as a bus master.

In Figure 11, the MSC8144 performs an outbound transaction. The bottom screenshot shows the MSC8144 writing the value 0xDEADBEEF to its local outbound window. This address is translated to the PCI address that is mapped to the MPC8560 local inbound window. The top screenshot shows the MPC8560 reading the same value from its local inbound window.

### 9 Cache Line Size

The MSC8144 PCI controller has an internal cache line of 32 bytes. Although the cache line size register in the configuration space is writable, only the value 8 is valid. This value indicates a cache line of 8 doublewords or 32 bytes. When the MSC8144 acts as a target, the bus command PCI MEMORY READ fetches a cache line of data. 32 bytes of data are fetched, regardless of the size requested by the initiator. In the MSC8144, there is no difference between the PCI MEMORY READ and PCI MEMORY READ LINE commands because the entire cache line is fetched in both cases. The PCI MEMORY READ MULTIPLE command is also similar, but it supports prefetching. This command causes a prefetch of the next cache line.

### 10 Latency Timer

The minimum grant (MIN_GNT) space register defines the minimum time, in increments of 250 ns, during which the master retains ownership of the bus for adequate performance. This read-only register is useful in determining the value to be programmed into the bus master latency timer (LT) configuration register. Because the MSC8144 PCI controller is a bridge between PCI and local memory, it does not have specific requirements for the LT when it operates as a bus master. Therefore, MIN_GNT is hard-wired to zero.
Configuration software should configure LT according to system requirements. The LT value is system-dependent and this value should be tuned to maximize utilization without starving the other PCI bus masters. For example, leaving the LT at zero may require the master to rearbitrate for the bus for long data transfers, but setting it to the maximum value may potentially keep other masters from accessing the bus.

11 Interrupt Handling

In many PCI devices, the INTA, INTB, INTC, and INTD pins signal interrupts to the PCI bus. The interrupt pin configuration register in the configuration space at offset 0x3D indicates which of these four pins the device uses. However, the MSC8144 does not implement these four PCI interrupt pins. It is recommended that the general-purpose interrupt request lines IRQ0:15 and INT_OUT be used to route interrupt sources.

12 MPC8560 Host Configuration Code

/////////////////////////////////////////////////////////////////////////
// Switch settings:
// SW1 00000110 PCI boot
// SW2 01101111 Disable JTAG chain
// SW3 10010111 PCI (256M DDR)
// SW4 01100010
/////////////////////////////////////////////////////////////////////////

#include <stdio.h>
#include "mpc8560pci.h"

// Check which ADx pin is connected to IDSEL
#define MSC8144_PCIDEVICENUM21
#define MPC8560_PCIDEVICENUM0

// PCI device id for MSC8144
#define MSC8144_PCIDEVID0x1400
#define MPC8560_PCIDEVID0x0009
#define MPC8560_PCIEVIDOR0x1057
#define MSC8144_PCIEVIDOR0x1957

#define READ_UINT8(data, arg)       data = (uint8_t)(arg)
#define READ_UINT16(data, arg)      data = (uint16_t)(arg)
#define READ_UINT32(data, arg)      data = (uint32_t)(arg)
#define GET_UINT8(arg)              (uint8_t)(arg)
#define GET_UINT16(arg)             (uint16_t)(arg)
#define GET_UINT32(arg)             (uint32_t)(arg)
#define WRITE_UINT8(arg, data)      arg = (uint8_t)(data)
#define WRITE_UINT16(arg, data)     arg = (uint16_t)(data)
#define WRITE_UINT32(arg, data)     arg = (uint32_t)(data)

uint32_t IMMR8144Local = 0x90000000;
uint32_t OutboundLocal = 0x80000000;
uint32_t InboundLocal = 0x00000000;
uint32_t IMMR8144PCI = 0x30000000;
uint32_t OutboundPCI = 0xC0000000;
```c
uint32_t InboundPCI = 0xE0000000;
uint32_t PCI8144BASE = 0x01F7A000;

msc8144_pci_regs *msc8144;
mpc8560_pci_regs *mpc8560 = (mpc8560_pci_regs*)0x40008000;

void setBCSR1();
uint32_t SwapLong(uint32_t);
uint32_t ConstructConfigWord (uint32_t ,uint32_t , uint32_t);
void setPIMMR(uint32_t);
void scanDevices();
uint32_t getWindowSize(uint32_t , uint32_t , uint32_t);
void setOutbound1(uint32_t , uint32_t , uint32_t);
void setOutbound2(uint32_t , uint32_t , uint32_t);
uint32_t calcWindowSize(uint32_t);
void writePCIConfigReg(uint32_t , uint32_t , uint32_t , uint32_t);
void setPCIConfigReg32(uint32_t , uint32_t , uint32_t , uint32_t);
void setPCIConfigReg16(uint32_t , uint32_t , uint32_t , uint16_t);
void setPCIConfigReg8(uint32_t , uint32_t , uint32_t , uint8_t);
uint32_t readPCIConfigReg(uint32_t , uint32_t , uint32_t);
uint32_t getPCIConfigReg32(uint32_t , uint32_t , uint32_t);
uint32_t getPCIConfigReg16(uint32_t , uint32_t , uint32_t);
uint16_t getPCIConfigReg8(uint32_t , uint32_t , uint32_t);

uint32_t SwapLong(uint32_t value)
{
    value = ((value & 0xFF)         << 24) |
             ((value & 0xFF00)       <<  8) |
             ((value & 0xFF0000UL)   >>  8) |
             ((value & 0xFF000000UL) >> 24) ;
    return value;
}

uint32_t ConstructConfigWord (uint32_t BusNum,uint32_t DevNum,uint32_t RegNum)
{
    uint32_t value;
    uint32_t FuncNum=0x0;

    value = (
        ((BusNum  & 0xFF)  << 16) |
        ((DevNum  & 0xFF)  << 11) |
        ((FuncNum & 0xFF)  <<  8) |
        ((RegNum  & 0xFC)       ) |
        ENABLE |
    ) ;

    return value;
}

void writePCIConfigReg(uint32_t BusNum, uint32_t DevNum, uint32_t Reg, uint32_t Data)
{
    uint32_t cfg_addr;
    uint32_t data32;

    data32 = SwapLong(Data);
    if ((Reg & 0x3) == 1)
        data32 = data32 >> 8;
```
else if ((Reg & 0x3) == 2)
data32 = data32 >> 16;
else if ((Reg & 0x3) == 3)
data32 = data32 >> 24;

cfg_addr = ConstructConfigWord(BusNum, DevNum, Reg);
WRITE_UINT32(mpc8560->config_addr, cfg_addr);
WRITE_UINT32(mpc8560->config_data, data32);
}

void setPCIConfigReg32(uint32_t BusNum, uint32_t DevNum, uint32_t Reg, uint32_t Data)
{
    writePCIConfigReg(BusNum, DevNum, Reg, Data);
}

void setPCIConfigReg16(uint32_t BusNum, uint32_t DevNum, uint32_t Reg, uint16_t Data)
{
    writePCIConfigReg(BusNum, DevNum, Reg, Data);
}

void setPCIConfigReg8(uint32_t BusNum, uint32_t DevNum, uint32_t Reg, uint8_t Data)
{
    writePCIConfigReg(BusNum, DevNum, Reg, Data);
}

uint32_t readPCIConfigReg(uint32_t BusNum, uint32_t DevNum, uint32_t Reg)
{
    uint32_t cfg_addr;
    uint32_t cfg_data;
    cfg_addr = ConstructConfigWord(BusNum, DevNum, Reg);
    WRITE_UINT32(mpc8560->config_addr, cfg_addr);
    READ_UINT32(cfg_data, mpc8560->config_data);
    cfg_data = SwapLong(cfg_data);
    if ((Reg & 0x3) == 1)
        cfg_data = cfg_data >> 8;
    else if ((Reg & 0x3) == 2)
        cfg_data = cfg_data >> 16;
    else if ((Reg & 0x3) == 3)
        cfg_data = cfg_data >> 24;
    return cfg_data;
}

uint32_t getPCIConfigReg32(uint32_t BusNum, uint32_t DevNum, uint32_t Reg)
{
    return (uint32_t)readPCIConfigReg(BusNum, DevNum, Reg);
}

uint16_t getPCIConfigReg16(uint32_t BusNum, uint32_t DevNum, uint32_t Reg)
{
    return (uint16_t)readPCIConfigReg(BusNum, DevNum, Reg);
}
uint8_t getPCIConfigReg8(uint32_t BusNum, uint32_t DevNum, uint32_t Reg)
{
    return (uint8_t)readPCIConfigReg(BusNum, DevNum, Reg);
}

void scanDevices()
{
    uint32_t i;
    uint16_t VendorID, DeviceID;
    uint32_t BusNum = 0;

    for(i = 0; i < 0x100; i++)
    {
        VendorID = getPCIConfigReg16(BusNum, i, REG_VENDORID);
        DeviceID = getPCIConfigReg16(BusNum, i, REG_DEVID);
        if(VendorID != 0xFFFF)
        {
            printf(" Device found: Device %x, Bus %x, DevID = %x, VendorID = %x\n",
                   i, BusNum, DeviceID, VendorID);
        }
    }
}

void setBCSR1()
{
    uint8_t *bcsrc1 = (uint8_t*)0xF8000001;
    *bcsrc1 |= 0x30;
}

uint32_t getWindowSize(uint32_t BusNum, uint32_t DevNum, uint32_t Reg)
{
    uint32_t new, orig;
    uint32_t size;

    // Read CfgReg
    orig = getPCIConfigReg32(BusNum, DevNum, Reg);

    // Write all 1's
    setPCIConfigReg32(BusNum, DevNum, Reg, 0xFFFFFFFF);

    // Read back to determine size
    new = getPCIConfigReg32(BusNum, DevNum, Reg);

    // Restore orig register value
    setPCIConfigReg32(BusNum, DevNum, Reg, orig);

    // Calculate size required by agent
    if (new & 1)
        size = (~new | 3) + 1; // I/O space
    else
        size = (~new | 0xF) + 1; // Memory space

    return size;
}
void setOutbound2(uint32_t LocalAddr, uint32_t PCIAddr, uint32_t WindowSize)
{
    // PCI address
    mpc8560->potar2 = PCIAddr >> 12;

    // Local Addr
    mpc8560->powbar2 = LocalAddr >> 12;

    // Enable, memory read/write, window size
    mpc8560->powar2 = 0x80044000 | WindowSize;
}

void setOutbound1(uint32_t LocalAddr, uint32_t PCIAddr, uint32_t WindowSize)
{
    // PCI address
    mpc8560->potar1 = PCIAddr >> 12;

    // Local Addr
    mpc8560->powbar1 = LocalAddr >> 12;

    // Enable, memory read/write, window size
    mpc8560->powar1 = 0x80044000 | WindowSize;
}

tuint32_t calcWindowSize(uint32_t Size)
{
    int i;
    uint32_t WindowSize = 0;

    for(i = 0; i < 32; i++)
    {
        if(Size & 0x01)
        {
            WindowSize += i;
        }
        Size = Size >> 1;
    }
    WindowSize = WindowSize - 1;
    return WindowSize;
}

void main()
{
    uint32_t BusNum = 0;
    uint32_t *outbound, *inbound;
    uint32_t size;
    uint32_t windowstart;
    uint16_t status_8144;
    uint32_t data;

    // Set BCSR1[RGMII1EN and RGMII2EN] = 1 (Disable) to select PCI/UTP instead of RGMII
    setBCSR1();

    // Scan devices on the bus
scanDevices();

// Set 8560 as the arbiter
setPCIConfigReg16(BusNum, MPC8560_PCIDEVICENUM, REG_ARBITER, 0);

// Set latency timer to max clock cycles to generate stop
setPCIConfigReg8(BusNum, MPC8560_PCIDEVICENUM, REG_BUSLATENCY, 0xF8);

// ************************************************************
// 8560 Outbound / 8144 Inbound
// 8560 0x8000_0000 --> PCI 0x8000_0000 --> 8144 0xC000_0000
// Size 512KB
// ************************************************************

// Determine window sizes for 8144 GPLx = 0x0010_0000
size = getWindowSize(BusNum, MSC8144_PCIDEVICENUM, REG_GPLBAR0); //M2

// Reassign 8144 inbound window 0 start addr in PCI memory space
setPCIConfigReg32(BusNum, MSC8144_PCIDEVICENUM, REG_GPLBAR0, OutboundPCI);

// Read 8144 PCI base inbound window = 0x2000_0008 (prefetchable)
windowstart = getPCIConfigReg32(BusNum, MSC8144_PCIDEVICENUM, REG_GPLBAR0);

// Set outbound window local address 0x80000000, PCI address 0x20000000, 512KB size
setOutbound1(OutboundLocal, OutboundPCI, calcWindowSize(size));

// ************************************************************
// 8144 Outbound / 8560 Inbound
// 8144 0xE000_0000 --> PCI 0xE000_0000 --> 8560 0x0000_0000
// Size 256MB
// ************************************************************

// Set inbound window local address 0x00000000, PCI address 0xE0000000

// PCI address
mpc8560->piwbar1 = InboundPCI >> 12;

// Local Addr
mpc8560->pitar1 = InboundLocal >> 12;

// Enable, prefetch, target i/f local mem, r/w snoop, 256MB size
mpc8560->piwar1 = 0xA0F5501B;

// ************************************************************
// 8144 PIMMR memory mapped space
// 8560 0x9000_0000 --> PCI 0x3000_0000 --> 8144 0xFE00_0000 (IMMR)
// Size 32MB
// ************************************************************

// Determine window size and set start addr for 8144 PIMMR
size = getWindowSize(BusNum, MSC8144_PCIDEVICENUM, REG_PIMMRBACR);//IMMR 32MB

// Reassign 8144 inbound window 1 start addr in PCI memory space
setPCIConfigReg32(BusNum, MSC8144_PCIDEVICENUM, REG_PIMMRBACR, IMMR8144PCI);

// Set outbound window local address 0x00000000, PCI address 0x30000000, 32MB size
setOutbound2(IMMR8144Local, IMMR8144PCI, calcWindowSize(size));
setPCIConfigReg16(BusNum, MPC8560_PCIDEVICENUM, REG_BUSCMD, 0x0146);

setPCIConfigReg16(BusNum, MSC8144_PCIDEVICENUM, REG_BUSCMD, 0x0146);

PCI8144BASE = PCI8144BASE + IMMR8144Local;
msc8144 = (msc8144_pci_regs*)PCI8144BASE;

// MSC8144 outbound 0 enable, prefetch, streaming
msc8144->outbound[0].potar = InboundPCI >> 12;
msc8144->outbound[0].pobar = 0x0000E0000;
msc8144->outbound[0].pocmr = 0xA00F8000;

outbound = (uint32_t*)OutboundLocal;
*outbound = (uint32_t)0x11223344;

inbound = (uint32_t*)InboundLocal;
data = *inbound;

13 MPC8560 Header File

#include "os_datatypes.h"

#define ENABLE 0x80000000

// Configuration Access Registers
#define REG_VENDORID 0x0000
#define REG_DEVID 0x0002
#define REG_BUSCMD 0x0004
#define REG_BUSSTATUS 0x0006
#define REG_REVID 0x0008
#define REG_BUSPROGIF 0x0009
#define REG_SUBCLASS 0x000A
#define REG_BASECLASS 0x000B
#define REG_CACHELINESZ 0x000C
#define REG_BUSLATENCY 0x000D
#define REG_HDRTYPE 0x000E
#define REG_BISTCTL 0x000F
#define REG_PIMMRBACR 0x0010
#define REG_GPLBAR0 0x0014
#define REG_GPLBAR1 0x0018
#define REG_GPLEXTBAR1 0x001C
#define REG_GPLBAR2 0x0020
#define REG_GPLEXTBAR2 0x0024
#define REG_SUBSYSVENDORID 0x002C
#define REG_SUBSYSID 0x002E
#define REG_CAPABILITYPTR 0x0034
#define REG_INTERRLINE 0x003C
#define REG_INTERRPIN 0x003D
#define REG_MINT 0x003E
#define REG_MAXLAT 0x003F
#define REG_FUNCTION 0x0044
#define REG_ARBITER 0x0046

#define SIZE_4KB 0x0B
#define SIZE_8KB 0x0C
#define SIZE_16KB 0x0D
#define SIZE_32KB 0x0E
#define SIZE_64KB 0x0F
#define SIZE_128KB 0x10
#define SIZE_256KB 0x11
#define SIZE_512KB 0x12
#define SIZE_1MB 0x13
#define SIZE_2MB 0x14
#define SIZE_4MB 0x15
#define SIZE_8MB 0x16
#define SIZE_16MB 0x17
#define SIZE_32MB 0x18
#define SIZE_64MB 0x19
#define SIZE_128MB 0x1A
#define SIZE_256MB 0x1B
#define SIZE_512MB 0x1C
#define SIZE_1GB 0x1D
#define SIZE_2GB 0x1E
#define SIZE_4GB 0x1F

typedef struct
{
    volatile uint32_t config_addr;
    volatile uint32_t config_data;
    volatile uint8_t reserved[0xBF4];
    volatile uint32_t potar0;
    volatile uint32_t potear0;
    volatile uint32_t powbar0;
}
volatile uint8_t reserved0[0x4];
volatile uint32_t powar0;
volatile uint8_t reserved1[0xC];

volatile uint32_t tpotar1;
volatile uint32_t tpotear1;
volatile uint32_t powbar1;
volatile uint8_t reserved2[0x4];
volatile uint32_t powar1;
volatile uint8_t reserved3[0xC];

volatile uint32_t tpotar2;
volatile uint32_t tpotear2;
volatile uint32_t powbar2;
volatile uint8_t reserved4[0x4];
volatile uint32_t powar2;
volatile uint8_t reserved5[0xC];

volatile uint32_t tpotar3;
volatile uint32_t tpotear3;
volatile uint32_t powbar3;
volatile uint8_t reserved6[0x4];
volatile uint32_t powar3;
volatile uint8_t reserved7[0xC];

volatile uint32_t tpotar4;
volatile uint32_t tpotear4;
volatile uint32_t powbar4;
volatile uint8_t reserved8[0x4];
volatile uint32_t powar4;
volatile uint8_t reserved9[0x10C];

volatile uint32_t tpitar3;
volatile uint8_t reserved10[0x4];
volatile uint32_t tpibar3;
volatile uint32_t tpibear3;
volatile uint32_t tpihar3;
volatile uint8_t reserved11[0xC];

volatile uint32_t tpitar2;
volatile uint8_t reserved12[0x4];
volatile uint32_t tpibar2;
volatile uint32_t tpibear2;
volatile uint32_t tpihar2;
volatile uint8_t reserved13[0xC];

volatile uint32_t tpitar1;
volatile uint8_t reserved14[0x4];
volatile uint32_t tpibar1;
volatile uint32_t tpibear1;
volatile uint32_t tpihar1;
volatile uint8_t reserved15[0xC];

} mpc8560_pci_regs;
typedef struct
{
    volatile uint32_t potar;            /* PCI Outbound Translation Address Register */
    volatile uint8_t  reserved1[0x4];
    volatile uint32_t pobar;            /* PCI Outbound Base Address Register */
    volatile uint8_t  reserved2[0x4];
    volatile uint32_t pocmr;            /* PCI Outbound Comparison Mask Register */
    volatile uint8_t  reserved3[0x4];
} pci_outbound_window_t;

typedef struct
{
    volatile uint32_t pitar;            /* PCI Inbound Translation Address Register */
    volatile uint8_t  reserved1[0x4];
    volatile uint32_t pibar;            /* PCI Inbound Base Address Register */
    volatile uint32_t piebar;           /* PCI Inbound Extended Base Address Register */
    volatile uint32_t piwar;            /* PCI Inbound Window Attributes Register */
    volatile uint8_t  reserved2[0x4];
} pci_inbound_window_t;

typedef struct
{
    /* PCI Error Management Registers */
    volatile uint32_t pci_esr; /* PCI error status register */
    volatile uint32_t pci_ecdr; /* PCI error capture disable register */
    volatile uint32_t pci_err; /* PCI error enable register */
    volatile uint32_t pci_eatcr; /* PCI error attributes capture register */
    volatile uint32_t pci_eacr; /* PCI error address capture register */
    volatile uint32_t pci_eeacr; /* PCI error extended address capture register */
    volatile uint32_t pci_edcr; /* PCI error data capture register */
    volatile uint8_t  reserved1[0x038 - 0x01C];

    /* PCI Inbound Registers */
    pci_inbound_window_t inbound2;
    pci_inbound_window_t inbound1;
    pci_inbound_window_t inbound0;
    volatile uint8_t    reserved2[0x100 - 0x080];

    /* PCI Outbound Registers */
    pci_outbound_window_t outbound[6];
    volatile uint8_t     reserved3[0x1F8 - 0x190];

    /* PCI Control Registers */
    volatile uint32_t dtcr;             /* Discard Timer Control Register */
    volatile uint8_t     reserved4[0x200 - 0x1FC];
} msc8144_pci_regs;