PCI and uHAL on the EBSA-285

Application Note

October 1998

Order Number: 278204-001

Information in this document is provided in connection with Intel products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications.

Intel may make changes to specifications and product descriptions at any time, without notice.

Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined." Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them.

The 21285 may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Copies of documents which have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling 1-800-548-4725 or by visiting Intel's website at http://www.intel.com.

Copyright © Intel Corporation, 1998

*Third-party brands and names are the property of their respective owners.

**ARM and StrongARM are trademarks of Advanced RISC Machines, Ltd.

Contents

Figures

Tables

1.0 Introduction

The EBSA-285 uHAL software package includes support for access to PCI via the 21285 Core Logic for SA-110 Microprocessor.

This application note should be used in conjunction with the following documents:

- *21285 Core Logic for SA-110 Microprocessor Data Sheet* (278115)
- *EBSA-285 Evaluation Board Reference Manual* (278136)
- *PCI Local Bus Specification*, Revision 2.1, available from the PCI Special Interest Group

Example code is taken from the uHAL software library distributed with StrongARM** evaluation boards.

2.0 Accessing a PCI Device

The Peripheral Component Interconnect *PCI Local Bus Specification*, Revision 2.1 is a hardware and software interface specification. This standard describes how to connect the peripheral components of a system in a structured and controlled way. The electrical interconnects and timings are specified as well as some standard device configuration registers.

The PCI bus is a little-endian bus; there is no provision to access PCI as a big-endian bus.

When reset, all devices on the PCI bus must cooperate to ensure devices can find each other and request desired resources.

The 21285 provides the software with a transparent interface to PCI devices. Each device can be assigned memory space during initialization and standard routines then allow each particular device to be located and used.

2.1 PCI Address Spaces

The processor and the peripheral devices need to access memory that is shared between them. Device driver software uses this memory to control PCI devices and exchange information with them. Typically, the shared memory contains control and status registers for the device.

The CPU’s system memory could be used for this shared memory, but if it were the CPU would have to stall every time a PCI device accessed memory. Since memory is generally available to one system component at a time, this would slow down the system. Peripheral devices should not be allowed to access main memory because a rogue device could make the system very unstable.

PCI has three distinct address spaces: PCI I/O, PCI Memory, and PCI Configuration space.

2.1.1 PCI I/O and Memory Space

Software running on the master device uses these two address spaces to communicate with peripheral devices. I/O space generally maps the registers of the peripheral so that software can check status and enable and disable activity. Memory space is used for functions that require larger amounts of memory, such as data buffers. For example, a video card would have control registers in I/O space and the video information in memory space.

Nothing can access these spaces until the PCI system has been set up and access permitted by writing the command field in the PCI configuration header.

2.1.2 PCI Configuration Space

All PCI devices have 256 bytes of configuration information. This allows other devices on the bus to check which other devices are also there and to configure them appropriately.

The PCI configuration registers allow each PCI device to be identified and initialized by the PCI master. Some values are fixed, such as the Vendor ID, Device ID, Class Code, and Revision ID. Others determine how quickly and to which PCI addresses the device responds.

The primary way to identify a PCI device is via the Class Code register, which is broken into three byte-size fields. The most significant byte, at offset 0Bh, is the base class byte, which classifies the function type of the device (that is, Bridge Device, Display Controller, etc.). The next byte, at offset 0Ah, is the subclass byte, which further classifies the device (that is, Ethernet, Token Ring, etc.). The least significant byte, at offset 09h, identifies a specific register-level programming interface (if any) so that device independent software can interact with the device. The Vendor ID register contains a code that uniquely identifies the manufacturer of the device, and the Device ID register contains a code to uniquely identify the device for the given manufacturer.

Using this information, for example, one EBSA-285 could find another EBSA-285 in a PCI system by looking for a processor class device, subclass coprocessor, with a vendor ID of 1011h (Intel) and a device ID of 1065h (21285).

These Configuration Registers appear at the start of the 21285 register space, as shown in Table 1 (the Base Address Register implementations are specific to the 21285).

Table 1. 21285 PCI Configuration Space Registers

31 24	23 16	15 8	7 0	Offset
Device ID		Vendor ID		00h
Status		Command		04h
Class Code			Revision ID	08h
BIST	Header Type	Latency Timer	Cache Line Size	0Ch
CSR Memory Base Address				10h
CSR I/O Base Address				14h
SDRAM Base Address				**18h**
Reserved (Unused Base Address)				**1Ch**
Reserved (Unused Base Address)				**20h**
Reserved (Unused Base Address)				**24h**
CardBus CIS Pointer				**28h**
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address				30h
Reserved				34h
Reserved				38h
Max Latency	Min Grant	Interrupt Pin	Interrupt Line	3Ch

2.2 Accessing Configuration Registers from Software

The EBSA-285 Evaluation Kit is shipped with software examples including the uHAL (Hardware Abstraction Layer) library. This library includes routines and data structures to access PCI devices. Configuration of PCI devices is discussed in Section 4.

There are three routines to read from PCI Configuration Space:

- SAr_PciCfgReadByte()
- SAr_PciCfgReadShort()
- SAr_PciCfgReadLong()

These routines are basically the same, but read 1, 2 or 4 bytes from the specified offset. Each routine requires a status word, to indicate that the data was properly accessed and the device exists as addressed, and returns the value read from the configuration space of the appropriate device.

Similarly, there are three routines to write to PCI Configuration Space:

- SAr_PciCfgWriteByte()
- SAr_PciCfgWriteShort()
- SAr_PciCfgWriteLong()

These routines work in exactly the same fashion as the read routines.

Example 1. Reading PCI Configuration Registers

```
/*==============================================================================
 *
 * routine:        SAr_PciCfgReadByte(U32 handle, U32 offset, PS32 pStatus)
 *
 * parameters:     handle = handle of device that we are assigning resources to
 *                 offset = offset into PCI config space for current BAR
 *                 data = value to be written
 *                 pStatus = SUCCESS/FAIL flag
 *
 * description:    This routine read a byte value from the given
 *                 PCI config register.
 *
 * calls:          SAir_CheckCfgParams(), SAir_PciMakeConfigAddress()
 *
 * returns:        data - value read.
 */
U8 SAr_PciCfgReadByte(U32 devHandle, U32 offset, PS32 pStatus)
```

```
{
PU8         pAddress ;
U8          data ;
            /* check device params */
            *pStatus = (S32)SAir_CheckCfgParams(devHandle, offset) ;
            if (*pStatus != MSG_SUCCESS)
                   return(0) ;
            /* generate the address of correct configuration space */
            pAddress = (PU8)(SAir_PciMakeConfigAddress(devHandle, offset)) ;
            /* now that we have valid params, go read the config space data */
            data = *pAddress ;
            *pStatus = MSG_SUCCESS ;
            return(data) ;
}
#define SAr_GetPciVendorID(h, s)  SAr_PciCfgReadShort(h, PCI_VENDOR_ID, s)
#define SAr_GetPciDeviceID(h, s)  SAr_PciCfgReadShort(h, PCI_DEVICE_ID, s)
#define SAr_GetPciClassID(h, s)   (SAr_PciCfgReadLong(h, PCI_REV_ID, s) >> 8)
#define SAr_GetPciRevID(h, s)     SAr_PciCfgReadByte(h, PCI_REV_ID, s)
#define SAr_GetPciBAR(h, o, s)    SAr_PciCfgReadLong(h, PCI_MEM_BAR + (o * 4), s)
#define SAr_GetPciCommand(h, s)   SAr_PciCfgReadShort(h, PCI_COMMAND, s)
#define SAr_GetPciStatus(h, s)    SAr_PciCfgReadShort(h, PCI_STATUS, s)
```

As shown above, uHAL provides several aliases for the read routines to make register access more consistent and explicit.

3.0 PCI Bus Initialization

PCI specifies that each device must set its registers to appropriate values and clear the reset flag in the status register. The EBSA-285 needs to set the registers on the 21285 so that it appears in a useful state on the PCI bus.

The PCI setup routine is as follows:

- Disable all PCI interrupts.
- Remove PCI reset flag; if the 21285 is driving PCI reset, this releases other devices to start their initialization.
- Set the Base Address Registers (BARs) so that the PCI host can assign the EBSA-285 an appropriate amount of PCI address space. The 21285 can only support certain memory sizes, so there is an array of sizes, largest first, that is scanned to determine the correct mask for the Address Mask Register.
- If this board is configured as PCI host, set up the 21285 BARs and activate the memory, I/O, and master bits.
- Set the SETUP_COMPLETE flag to indicate to the PCI domain that the EBSA-285 is ready.

This is done with an ARM** Assembler macro called SETUP_PCI (in lib/ebsa285/target.s), which can be found in the Angel* and uHAL sources included in the EBSA-285 Evaluation Kit. In uHAL, there is an example "C" routine that behaves in exactly the same fashion. This routine is in lib/ebsa285/platform.c. If an application requires different PCI initialization code, it is suggested that the macro call is removed from lib/boot.s and a suitably modified version of SAir_PciInit() is called from targetInit().

Example 2. Initialization Code

```
void SAir_PciInit(U32 memSize, U32 memBase)
{
U8      *base = (U8 *)DC21285_ARMCSR_BASE ;
U32     *pciMemSize = SAiv_PciSizes ;
U32     *pciMemMask = SAiv_PciMasks ;
U32     temp ;

/* Disable PCI Outbound interrupts */
*(U32 *)(base + PCI_OUT_INT_MASK) = NO_OUTBOUND_INTS ;

/* Disable Doorbells */
*(U32 *)(base + DBELL_PCI_MASK) = 0 ;
*(U32 *)(base + DBELL_SA_MASK) = 0 ;

/* High PCI address bits all map to 0 */
*(U32 *)(base + PCI_ADDR_EXT) = 0 ;
/* Set Interrupt ID to 1 - fixes problems on systems which cannot
 * handle interrupt 0.
 */
*(U32 *)(base + PCI_INT_LINE) = SETUP_PCI_INT_ID ;

/* Remove PCI_reset; NOTE can’t assert reset if already done */
temp = *(U32 *)(base + SA_CONTROL) ;
temp |= PCI_NRESET ;
*(U32 *)(base + SA_CONTROL) = temp ;

/* Open up a window from PCI memory space into the EBSA-285
* SDRAM. If configured as an add-in card, this allows the
* host to allocate memory for the EBSA-285 during its POST.
*/
while (*pciMemSize && *pciMemSize > memSize) {
           pciMemSize++ ;
           pciMemMask++ ;
}
*(U32 *)(base + DRAM_BASE_ADDR_MASK) = *pciMemMask ;
*(U32 *)(base + DRAM_BASE_ADDR_OFF) = memBase ;

/* Only init PCI if central function & Standalone Bench Bit isn’t set */
if (SAr_PciHost()) {
           /* Don’t respond to any commands */
           *(U32 *)(base + PCI_COMMAND) = 0 ;
           /* Set Memory Base Address Register to our default */
           *(U32 *)(base + PCI_MEM_BAR) = 0x40000000 ;
           *(U32 *)(base + PCI_IO_BAR)   = 0xF000 ;
           *(U32 *)(base + PCI_DRAM_BAR) = 0 ;
           /* Done: respond to I/O space & Memory transactions.
            *         ALSO BE PCI MASTER
            */
           *(U32 *)(base + PCI_COMMAND) = PCI_CFN_INIT ;
}

/* Signal PCI_init_complete, won’t hurt if it’s already been done */
temp |= INIT_COMPLETE ;
*(U32 *)(base + SA_CONTROL) = temp ;
}
```

4.0 Configuring PCI Devices

In uHAL, it is assumed that the EBSA-285 can only configure other cards if it is the PCI host; this avoids problems with multiple devices attempting to configure the PCI memory space. uHAL scans the bus and builds a list of device nodes; then resources are assigned for each device found.

4.1 Procedure

A PCI card is configured as follows:

- To stop the card from accessing PCI, it is first disabled.
- Each Base Address Register (BAR) is reset by writing -1 to it.
- When the BAR is read, it returns the size and type of PCI space required. This size information must be stored in the devnode, as this register will soon hold the base address, rather than size.
- Assign the next base address for the appropriate PCI space. I/O space must be an address less than 1MB; memory can be any 32-bit address. Each address must be naturally-aligned with the requested size; thus, if a device asks for 2MB of memory, the address assigned to that BAR would be rounded to a 2MB boundary. The area of PCI memory space between the new base address and the top of the previous allocated space would not be assigned. This example code is straightforward and does not attempt to optimize address allocation.
- The Expansion ROM BAR is scanned to allow easy configuration when required. No PCI memory space is assigned at this point.
- Once all the BARs have been scanned and initialized for this device, accesses are re-enabled according to the types of BAR found.

Example 3. Assigning PCI Resources

```
void SAir_AssignResources(U32 handle, U32 *sizePtr)
{
U32     data, offset = PCI_MEM_BAR ;
S32     i, status, barFlag = 0 ;

        /* Disable PCI IO and memory accesses */
        SAr_PciCfgWriteShort(handle, PCI_COMMAND, 0, &status) ;

        /* Scan each of the BARS for this device */
        for (i = 0; i < PCI_MAX_BAR; i++) {
            data = SAir_ReadBARSize(handle, sizePtr, offset) ;
            if (data != 0) {
                /* Assign some PCI memory space to this BAR */
                barFlag |= SAir_WriteBARSize(handle, data,
                    *sizePtr, offset) ;
            }
            /* increment the BAR offset value */
            offset += 4 ;
            sizePtr++ ;
        }

        /* Check out the Expansion ROM, but don’t give it any PCI space */
        sizePtr++ ;
        offset = PCI_ROM_BAR ;
        SAir_ReadBARSize(handle, sizePtr, offset) ;

        /* Enable PCI IO and memory accesses as found by scan, with MASTER */
        barFlag |= PCI_MASTER_ENABLE ;
        SAr_PciCfgWriteShort(handle, PCI_COMMAND, barFlag, &status) ;
        SAr_PciCfgWriteByte(handle, PCI_LATENCY, 0xFF, &status) ;
}
```

4.2 Support Routines

4.2.1 Accessing Base Address Registers

To improve the readability of the code, some of the complexities of PCI initialization have been put into separate routines. One of these is reading the size of the memory blocks requested by a device.

Example 4. Reading the Memory Block Size

```
/*=============================================================================
 *
 * routine:        SAir_ReadBARSize(U32 handle, U32 offset)
 *
 * parameters:     handle = handle of device that we are assigning resources to
 *                 offset = offset into PCI config space for current BAR
 *
 * description:    this routine will attempt to glean resource requirements
 *                 from the device pointed to by handle
 *
 * calls:          pciCfgReadLong(), pciCfgWriteLong(),
 *
 * returns:        data - value read from BAR
 */

U32 SAir_ReadBARSize(U32 handle, U32 *sizePtr, U32 offset)
{
U32     data ;
S32     status ;

        /* write PCI_INVALID to BAR to get size */
        SAr_PciCfgWriteLong(handle, offset, PCI_INVALID, &status) ;
        /* check if this is an implemented BAR (size != 0) */
        data = SAr_PciCfgReadLong(handle, offset, &status) ;
        if (data != 0) {
            /* Keep a note of the size, because it isn't easy
             * to read - have to disable card, read address,
             * set to -1, read size, reload address & re-enable!
             */
            *sizePtr = (~(data & 0xFFFFFFF0)) + 1 ;
        }
            *sizePtr = 0 ;
        }

        return (data) ;
}
```

Example 5. Writing the Memory Block Base

```
/*============================================================================
 *
 * routine:        SAir_WriteBARSize(U32 handle, U32 data, U32 size, U32 offset)
 *
 * parameters:     handle = handle of device that we are assigning resources to
 *                 data   = raw data read from BAR
 *                 size   = size of PCI space required
 *                 offset = offset into PCI config space for current BAR
 *
 * description:    this routine writes the base address for the requested size
 *                 to the device pointed to by handle
 *
 * calls:          pciCfgWriteLong(),
 *                 SAir_AssignIoAddr(), SAir_AssignMemAddr()
 *
 * returns:        void
 */

S32 SAir_WriteBARSize(U32 handle, U32 data, U32 size, U32 offset)
{
S32     status, barFlag = 0 ;

        /* Do not assign PCI space to ROM Expansion BAR to IO space */
        if ((offset != PCI_ROM_BAR) && (data & PCI_IO_ENABLE)) {
            /* IO space BAR */
            SAr_PciCfgWriteLong(handle, offset,
                SAir_AssignIoAddr(size), &status) ;
            barFlag |= PCI_IO_ENABLE ;
        }
        else {
            /* memory space BAR */
            SAr_PciCfgWriteLong(handle, offset,
                SAir_AssignMemAddr(size), &status) ;
            barFlag |= PCI_MEM_ENABLE ;
        }

        return barFlag ;
}
```

When configuring a device, uHAL provides routines to find the next PCI memory, or PCI IO Base Address Register. If bar is zero, this indicates first bar. A return value of zero indicates no more BARs of that type.

Example 6. Quick Access to Base Address Registers

```
/*=============================================================================
 *
 * routine:       SAr_GetNextMemBAR(U32 handle, U32 *bar)
 *
 * parameters:    handle - handle of device being accessed
 *                *bar - offset into Base Address Registers to start from
 *
 * description:   this routine will scan the remaining Base Address Registers
 *                for the specified device. When a BAR is found which points to
 *                PCI memory space, the address in the BAR is returned.
 *
 * calls:         SAr_GetPciBAR()
 *
 * returns:       contents of BAR (0 if none found)
 *                *bar is modified to next possible BAR offset
 */
U32 SAr_GetNextMemBAR(U32 handle, U32 *bar)
{
U32     i, data ;
S32     status ;

        /* Scan remaining BARS for this device */
        for (i = *bar; i < PCI_MAX_BAR; i++) {
            /* Read back address assigned for this BAR */
            data = SAr_GetPciBAR(handle, i, &status) ;

            /* If not I/O, then it is memory */
            if ((data & PCI_IO_ENABLE) == 0) {
                /* Got one, set next available BAR & return address */
                *bar = i + 1 ;
                return (data & ~0x0F) ;
            }
        }

        /* Nothing found */
        *bar = 0 ;
        return 0 ;
}
```

```
/*============================================================================
 *
 * routine:       SAr_GetNextIOBAR(U32 handle, U32 *bar)
 *
 * parameters:    handle - handle of device being accessed
 *                *bar - offset into Base Address Registers to start from
 *
 * description:   this routine will scan the remaining Base Address Registers
 *                for the specified device. When a BAR is found which points to
 *                PCI I/O space, the address in the BAR is returned.
 *
 * calls:         SAr_GetPciBAR()
 *
 * returns:       contents of BAR (0 if none found)
 *                *bar is modified to next possible BAR offset
 */

U32 SAr_GetNextIOBAR(U32 handle, U32 *bar)
{
U32    i, data ;
S32    status ;

       /* Scan remaining BARS for this device */
       for (i = *bar; i < PCI_MAX_BAR; i++) {
           /* Read back address assigned for this BAR */
           data = SAr_GetPciBAR(handle, i, &status) ;

           /* Check if it is I/O */
           if (data & PCI_IO_ENABLE) {
               /* Got one, set next available BAR & return address */
               *bar = i + 1 ;
               return (data & ~0x0F) ;
           }
       }

       /* Nothing found */
       *bar = 0 ;
       return 0 ;
}
```

4.2.2 Assigning PCI Addresses

Each PCI device is mapped to one or more address ranges in PCI memory and/or I/O space. The least significant bit of the size read from the BAR determines whether the area is I/O or memory (see Example 5). The Expansion ROM BAR may indicate the amount of expansion ROM in the system, but the least significant bit is different to the other BARs. It is not used as a PCI Memory/ PCI I/O flag. On the 21285, the least significant bit of the Expansion ROM BAR will follow the PCI usage when bit 31 of the expansion ROM base address mask register (ERBAMR) is clear. If ERBAMR bit 31 is set, PCI access to the expansion ROM is disabled.

uHAL maintains a pair of variables that point to the next available PCI I/O address block and the next available PCI Memory address block.

Example 7. Reserving PCI Address Blocks

```
static U32      SAiv_PciMemAvailableAddress = SZ_1M ;
static U32      SAiv_PciIoAvailableAddress = 0x400 ;
/*============================================================================
 *
 * routine:        SAir_GetNextPciAddr(U32 *base, U32 size)
 *
 * parameters:     *base - pointer to next available PCI address
 *                 size - amount of memory to be assigned
 *
 * description:    this routine will return the next available aligned memory
 *                 address block. This address block is added to PCI_MEM to
 *                 obtain the memory location as seen by the StrongARM.
 *
 * calls:          SAir_AlignAddress()
 *
 * returns:        available aligned memory address
 */

U32 SAir_GetNextPciAddr(U32 *base, U32 size)
{
U32     address ;

        /* align the address to size (i.e., 4 meg size requires 4 meg alignment */
        address = SAir_AlignAddress(*base, size) ;

        /* generate next available address for next call */
        *base = address + size ;

#ifdef DEBUG
        printf("device requesting memory space %d bytes, assigned %08Xh\n",
            size, address) ;
#endif

        return(address) ;
}
```

```
/*=============================================================================
 *
 * routine:        SAir_AssignMemAddr(U32 size)
 *
 * parameters:     size - converted value read from BAR after writing PCI_INVALID
 *
 * description:    this routine will return the next available aligned memory
 *                 address block. This address block is added to PCI_MEM to
 *                 obtain the memory location as seen by the StrongARM.
 *
 * calls:          SAir_GetNextPciAddr()
 *
 * returns:        available aligned memory address
 */

U32 SAir_AssignMemAddr(U32 size)
{
       return(SAir_GetNextPciAddr(&SAiv_PciMemAvailableAddress, size)) ;
}

/*=============================================================================
 *
 * routine:        SAir_AssignIoAddr(U32 size)
 *
 * parameters:     size - converted value read from BAR after writing PCI_INVALID
 *
 * description:    this routine will return the next available aligned IO
 *                 address block (must be < 1M, according to PCI spec.). This
 *                 address is added to PCI_IO to obtain the memory location
 *                 as seen by the StrongARM.
 *
 * calls:          SAir_GetNextPciAddr()
 *
 * returns:        available aligned IO address
 */

U32 SAir_AssignIoAddr(U32 size)
{
       return(SAir_GetNextPciAddr(&SAiv_PciIoAvailableAddress, size)) ;
}
```

4.2.3 Accessing PCI Address Space

Once a device is mapped into PCI space, it can simply be accessed as an area (or areas) of memory by the StrongARM.* The base address assigned to each register is stored in the devnode for each device. To convert the raw PCI address into a StrongARM address, two simple macros are used:

```
#define _MapIOAddress(a)            ((U32)PCI_IO + (U32)(a))

#define _MapMemAddress(a)           ((U32)PCI_MEMORY + (U32)(a))
```

These convert the address read from a BAR of a PCI device to the equivalent address in the StrongARM memory map.

To convert back from the StrongARM address to the PCI memory address, uHAL provides the following routine.

Example 8. StrongARM to PCI Address Conversion

```
/*=============================================================================
 *
 * routine:         SAr_Mem2Pci(void *memAddress, U32 *status)
 *
 * parameters:      memAddress - StrongARM address to be converted to PCI space
 *                  *status - flag for good/bad address.
 *
 * description:     this routine will convert the given StrongARM address to PCI
 *                  space. First, need to look up the PCI base address and amount
 *                  of memory allocated to PCI - don’t use constants because the
 *                  devNode init code just slots the EBSA-285 into the next
 *                  available PCI space.
 *
 * calls:           SAr_FindHostNode() - returns devNode for this device
 *
 * returns:         address in PCI memory space
 *                  *status OK if address is in PCI space, else !OK
 */

void *SAr_Mem2Pci(void *memAddress, U32 *status)
{
devNodeType         *node ;
U32                 memPtr, pciBase, pciSize, bar ;
```

```
        memPtr = (U32)memAddress ;

        /* Simple mapping of StrongARM high memory -> PCI space */
            if (memPtr >= PCI_MEM) {
            *status = OK ;
            /* FIXME - target specific! */
            return (void *)(memPtr - PCI_MEM) ;
        }

        /* Else find our memory in PCI space:
         * This target may have no memory mapped to PCI space.
         */
        pciSize = PCI_DRAMSIZE ;
        /* Find our node */
        node = SAr_FindHostNode() ;

        if ((pciSize != 0) && (node != NULL)) {
            pciSize = node->memSize[HOST_MEMBAR] ;
            bar = HOST_MEMBAR ;
            pciBase = SAr_GetNextMemBAR(node->handle, &bar) ;

            /* Check the memory address is in range */
                if ((memPtr > PCI_DRAMBASE) &&
                    (memPtr <= (PCI_DRAMBASE + pciSize))) {
                    *status = OK ;
                    return (void *)(pciBase + memPtr - (U32)PCI_DRAMBASE) ;
            }
        }

        *status = ~OK ;
        return (void *)0 ;
    }
```

4.2.4 Finding PCI Devices

At a higher level, it is often useful to scan for a PCI device of a particular class or by using known Vendor IDs and or Device IDs. Again, the uHAL library provides routines for this. Each routine is called with an inst value, which can be used to find a PCI device when more than one matches the given criteria. That is, if inst is non-zero, each routine looks for the inst+1'th device. Being passed a pointer, the find routines modify the inst value so the calling routine can track which devices have been scanned. The find routines return the handle of the matching device.

Example 9. Finding PCI Devices by Class

```
/*==============================================================================
 *
 * routine:        SAr_FindPCIDevClass(U32 devClass, U32 *inst)
 *
 * parameters:     devClass - device class to scan for
 *                 *inst - pointer to instance of card to find.
 *
 * description:    this routine will scan the devNode tree looking for devices
 *                 which match the class code required. When a device matches
 *                 the class code, the instance is checked - this enables
 *                 software to uniquely address a device, even when more than
 *                 one may be fitted.
 *
 * calls:          SAr_GetPciClassID()
 *
 * returns:        handle to required device (0 if not found)
 *                 *inst is modified to next possible instance
 */

U32 SAr_FindPCIDevClass(U32 devClass, U32 *inst)
{
extern devNodeType        *devNodes ;
devNodeType               *current ;
U32                       newClass, count = 0 ;
S32                       status ;
```

```
        for (current = devNodes ; current != NULL; current = current->next) {
            /* check if device is PCI, if so check the device class */
            if ((current->handle) & PCI_TYPE) {
                newClass = SAr_GetPciClassID(current->handle, &status) ;

                if (newClass == devClass) {
                    if (count++ >= *inst) {
                        *inst = count ;
                        return (current->handle) ;
                    }
                }
            }
        }

        /* Nothing found */
        *inst = 0 ;
        return 0;
}
```

As well as looking for a given class of PCI device, the specific Vendor ID and/or Device ID can be used. SAr_FindPCIDevice() can search for devices using either or both IDs, so it is also useful to find all devices from a particular vendor.

Example 10. Finding PCI Devices by ID

```
/*=============================================================================
 *
 * routine:        SAr_FindPCIDevice(U16 vendor, U16 device, U32 *inst)
 *
 * parameters:     vendor - vendor code to scan for
 *                 device - device code to scan for
 *                 *inst - pointer to instance of card to find.
 *
 * description:    this routine will scan the devNode tree looking for devices
 *                 which match the vendor & device code required. When a device
 *                 matches both vendor and device codes, the instance is checked
 *                 - this enables software to uniquely address a device, even
 *                 when more than one may be fitted. If only a specific vendor
 *                 or device code is required (not both), the other code can
 *                 be ignored by specifying PCI_NOTFITTED.
 *
 * calls:          SAr_GetPciVendorID(), SAr_GetPciDeviceID()
 *
 * returns:        handle to required device (0 if not found)
 *                 *inst is modified to next possible instance
 */

U32 SAr_FindPCIDevice(U16 vendor, U16 device, U32 *inst)
{
extern devNodeType            *devNodes ;
devNodeType                   *current ;
U32                           value, count = 0 ;
S32                           status ;

        for (current = devNodes ; current != NULL; current = current->next) {
            /* check if device is PCI, if so check the device class */
            if ((current->handle) & PCI_TYPE) {
                value = SAr_GetPciVendorID(current->handle, &status) ;
                /* Allow don't care vendor code or match */
                if ((vendor == PCI_NOTFITTED) || (value == vendor)) {
                    value = SAr_GetPciDeviceID(current->handle, &status) ;
                    /* Allow don't care device code or match */
                    if ((device == PCI_NOTFITTED) || (value == device)) {
                        if (count++ >= *inst) {
                            *inst = count ;
                            return (current->handle) ;
                        }
                    }
                }
            }
        }

        /* Nothing found */
        *inst = 0 ;
        return 0;
}
```

5.0 Conclusion

PCI provides a high-speed local bus that enables multiple devices to be configured and used in a straightforward manner. PCI cannot remove any of the internal complexities of a given device, but it does provide a uniform way to identify, initialize, and access devices. Also, as device intelligence increases, the processor allows more work to be done by external devices, with data transferred via shared memory rather than serial protocols. The uHAL library provides a set of basic routines to further simplify and speed the development cycle.

Support, Products, and Documentation

If you need technical support, a *Product Catalog*, or help deciding which documentation best meets your needs, visit the Intel World Wide Web Internet site:

http://www.intel.com

Copies of documents that have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling **1-800-332-2717** or by visiting Intel's website for developers at:

http://developer.intel.com

You can also contact the Intel Massachusetts Information Line or the Intel Massachusetts Customer Technology Center. Please use the following information lines for support:

For documentation and general information:	
Intel Massachusetts Information Line	
United States:	1–800–332–2717
Outside United States:	1–303-675-2148
Electronic mail address:	techdoc@intel.com

For technical support:	
Intel Massachusetts Customer Technology Center	
Phone (U.S. and international):	1–978–568–7474
Fax:	1–978–568–6698
Electronic mail address:	techsup@intel.com