Initialization and Asynchronous Programming of the PCILynx TSB12LV21A 1394 Device

APPLICATION REPORT: SLLA023

Danny Mitchell
BuS Solutions Software Group

Mixed Signal and Logic Products
BuS Solutions Group
15 December 1997

TEXAS

TRADEMARKS

CONTACT INFORMATION

US Product Information Center Hotline	(972) 644-5580
US Product Information Center Fax	(972) 480-7800
US Product Information Center Email	sc-infomaster@ti.com
US 1394 Web Site	http://www.ti.com/sc/1394

Contents

Figures

Tables

Initialization and Asynchronous Programming of the TSB12LV21A PCILynx 1394 Device

Abstract

This document discusses the PCILynx 1394 device, its theory of operation, and setup and use. It provides example software to illustrate the initial programming of the PCILynx TSB12LV21A 1394 link-layer controller and the TI TSBKPCI Evaluation Module (EVM) board. Extensive software listings are shown.

Product Support

Related Documentation

Further information and data sheets can be obtained via the Internet at http://www.ti.com/sc/1394.

The following documents are available via links from the TI 1394 external web page:

- Errata List for TSB12LV21, TSB12LV21A, TSB12C01A, TSB11C01, TSB11LV01, TSB21LV03, TSB21LV03A, and TSB14C01.
- Data sheets for all TI 1394 devices.
- Information on designer kits, including TSBKPCI, TSBKPCITST, TSKBGPLYNX, TSBKBACKPL, TSBKPRPHRL.

The IEEE 1394-1995 standard is available for purchase at http://standards.ieee.org/catalog/index.html

World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

The URL specifically for the TI 1394 external web site is http://www.ti.com/sc/1394. On this page, one can subscribe to 1394Times, which periodically updates subscribers on events, articles, products, and other news regarding 1394 developments.

Introduction

This application note will provide example software to illustrate the initial programming of the PCILynx TSB12LV21A 1394 link-layer controller and the TI TSBKPCI Evaluation Module (EVM) board.

This document and source code is divided into 7 sections. They are shown below.

Table 1. Content Description

Section Name	Description
PCILynx Theory of Operation	Discusses the theory of operation of the PCILynx chip.
PCILynx DMA Engine Theory of Operation	Discusses the theory of operation of the PCILynx DMA Engine.
PCILynx Hardware Definitions and Data Structures	This section contains definitions of the PCILynx hardware registers that are used in this sample software. This software is included in appendix A for readability.
PCILynx Initialization	Initialization software for the PCILynx EVM board.
Setting Up a DMA Channel for Receiving of 1394 Bus Events	Setup a list of Packet Control Lists to enable the reception/transmission of asynchronous packets.
Linking a PCL Chain to a DMA Channel	After creating a PCL linked list this routine describes how to link the PCL chain to a DMA channel for asynchronous reception.
Processing a 1394 Bus Reset	This routine can be used as an example of the processing of a 1394 bus reset. This software will extract the self-id stream from the PCILynx chipset.

PCILynx Theory of Operation

The PCILynx TSB12LV21A is a PCI based 1394 host controller chip. It performs the transfer of packets from the 1394 bus to host PCI memory and transfers PCI memory over the 1394 bus. The transfer of data is accomplished by programming a DMA engine. The DMA engine is programmed by using a data structure called Packet Control Lists (PCLs). Each of the PCLs performs a scatter/gather DMA that completely describes the packets that are either transferred to PCI memory from the 1394 bus or to the 1394 bus from PCI memory. The PCLs are constructed in a linked list architecture that allows either one or many packets to be transferred. Each of the packets to be transferred must be completely described by a PCL for it to be transferred. The only method of moving data with the PCILynx is to use the PCL descriptors. Each of the PCL lists must be “linked” to a DMA channel. For receive operations comparator registers are set to filter which type of 1394 packets each DMA should accept. The DMA will not transfer 1394 packets into PCI memory unless the comparators have been set to accept the packet.

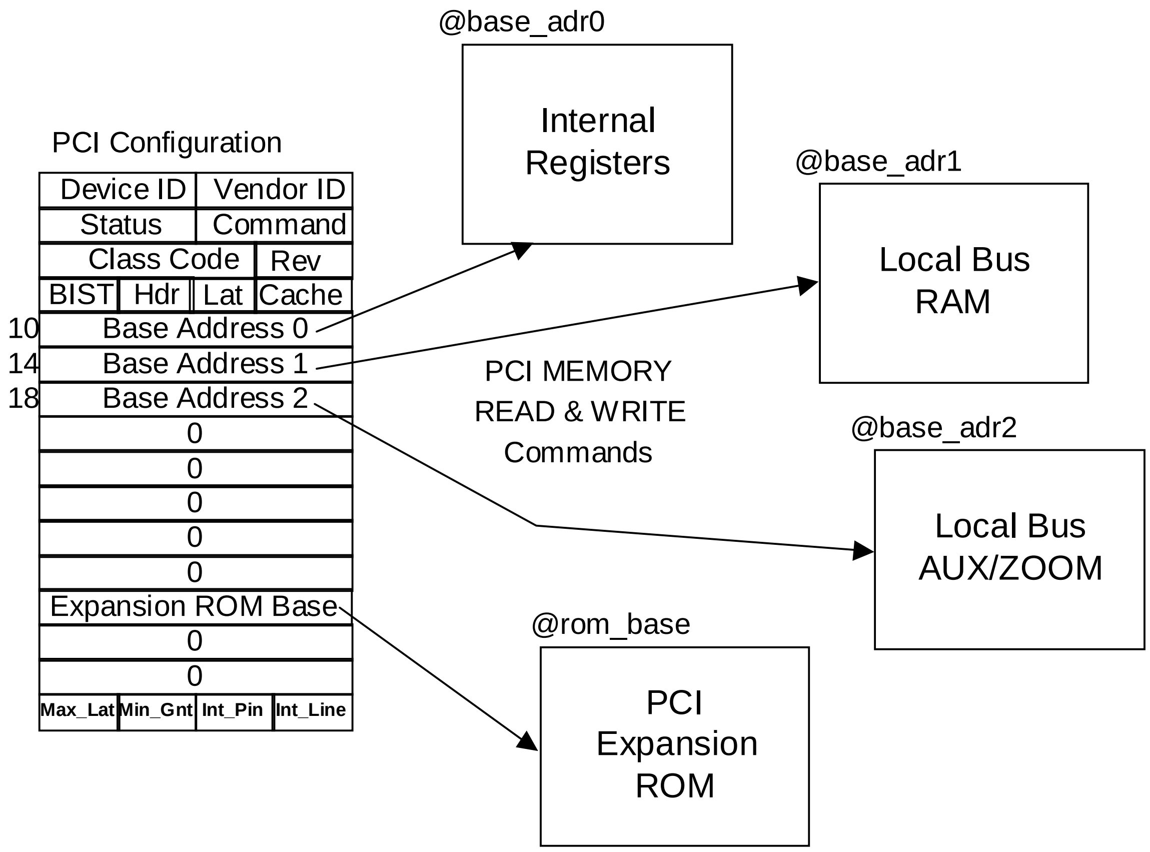
The PCL list and the buffer space that is transferred to/from MUST be in contiguous physical memory space. The DMA engines cannot handle virtual memory addresses when fetching PCL lists or transferring 1394 data to/from the serial bus.

The PCILynx hardware definition is contained in the complete source code in Appendix A. In that code all of the registers for the PCILynx are defined. Those register definitions will be used throughout this document as other code snippets are discussed. Along with the register descriptions are data structures to assist in accessing a PCL linked list.

PCI Slave Address Space

The TSB12LV21A PCILynx implements the PCI Configuration Space as required by the PCI Specification. These PCI memory slave address spaces are specified by the base address registers contained in the PCI Configuration space as follows:

Figure 1. PCI Memory Slave Address Spaces



The PCILynx DMA Engine Theory of Operation

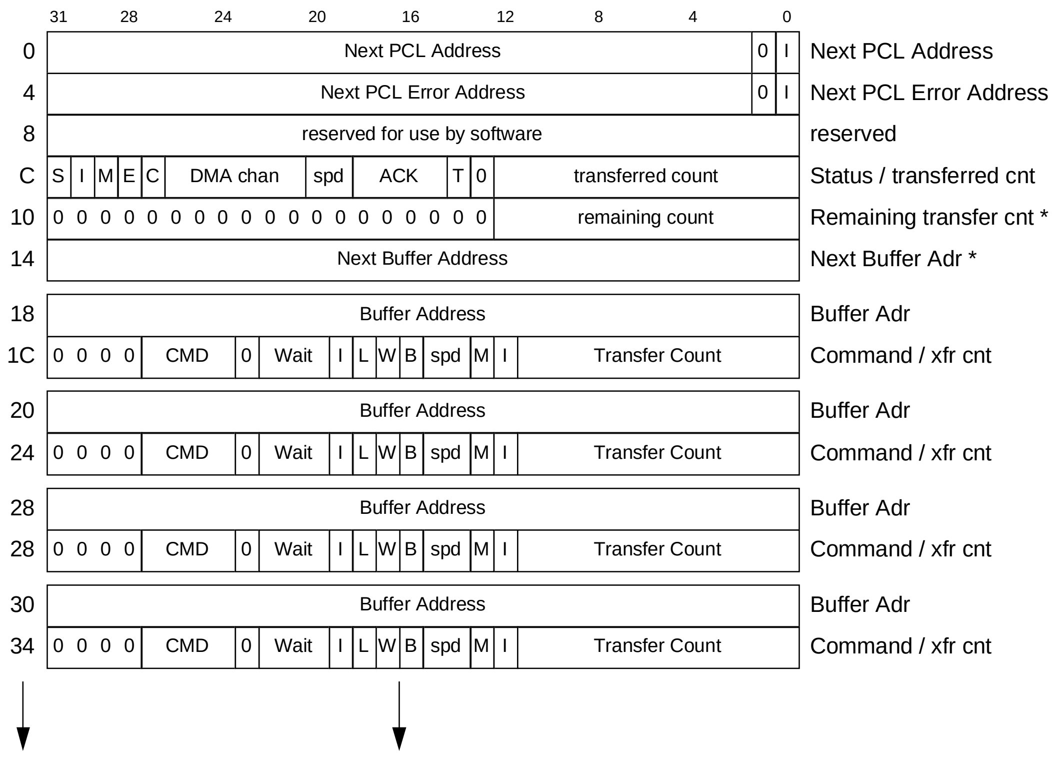
The DMA is controlled by data structures called Packet Control Lists or PCLs. The PCL contains command information, which the DMA fetches from memory as needed. These commands tell the DMA the sources and destinations for the data and how many bytes it is to transfer. Some commands move chunks of data between the 1394 Transmit FIFOs and the PCI and between the General Receive FIFO or GRF and the PCI. Another command moves data between the PCI and the AUX bus. Other commands are for secondary functions and are called auxiliary commands. These auxiliary commands allow the DMA to peek and poke quadlets of specified data to any PCI address and permit some conditional branching (described in the section defining PCL queues). The programming of the PCILynx DMA engines is described in detail in the PCILynx functional specification.

Below are two diagrams of a typical PCL data structure. Each of the fields is aligned on a 32 bit boundary. The first diagram shows fields on the 32 bit boundaries. The second diagram displays the individual bit fields of the PCL structure. A detailed explanation of each of the bit fields is beyond the scope of this document. For a detailed description of the usage of each bit of the data field of a PCL see the PCILynx functional specification.

Table 2. Typical PCL Data Structure

offset	PCL contents		DMA channel access performed
0x0	Next PCL address		read
0x4	Next PCL address after an Async transmit error		read
0x8	Reserved for use by software		ignored
0xC	PCL status and total transferred count		Updated by the DMA upon completion of PCL
0x10	Remaining Transfer count for the current scatter table entry		Updated by the DMA upon completion of PCL for RCV AND UPDATE and LBUS commands. Ignored by the DMA for other commands
0x14	Next Data Buffer address for the current scatter table entry		Updated by the DMA upon completion of PCL for RCV AND UPDATE commands. Ignored by the DMA for other commands
0x18	PCL command	Data buffer0 control and byte count	read
0x1C	Data buffer0 address pointer		read
0x20	Data buffer1 control and byte count		read
0x24	Data buffer1 address pointer		read
0x28	Data buffer2 control and byte count		read
0x2C	Data buffer2 address pointer		read
⋮	⋮		⋮
0x78	Data buffer12 control and byte count		read
0x7C	Data buffer12 address pointer		read

Figure 2. Typical Program Control List (PCL)



31 28 24 20 16 12 8 4 0
0 | Next PCL Address | 0 | I | Next PCL Address
4 | Next PCL Error Address | 0 | I | Next PCL Error Address
8 | reserved for use by software | reserved
C | S | I | M | E | C | DMA chan | spd | ACK | T | 0 | transferred count | Status / transferred cnt
10 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 | remaining count | Remaining transfer cnt *
14 | Next Buffer Address | Next Buffer Adr *
18 | Buffer Address | Buffer Adr
1C | 0 0 0 0 | CMD | 0 | Wait | I | L | W | B | spd | M | I | Transfer Count | Command / xfr cnt
20 | Buffer Address | Buffer Adr
24 | 0 0 0 0 | CMD | 0 | Wait | I | L | W | B | spd | M | I | Transfer Count | Command / xfr cnt
28 | Buffer Address | Buffer Adr
28 | 0 0 0 0 | CMD | 0 | Wait | I | L | W | B | spd | M | I | Transfer Count | Command / xfr cnt
30 | Buffer Address | Buffer Adr
34 | 0 0 0 0 | CMD | 0 | Wait | I | L | W | B | spd | M | I | Transfer Count | Command / xfr cnt

* updated on Receive and Update Command only

PCILynx Initialization

This section describes initialization of the PCILynx EVM board. The sample software shown below gives an example of initializing the PCILynx chip and the EVM board.

```
InitializeBoard()
{
   //
   // Software reset.
   // NOTE:  WriteLynxQuad and ReadLynxQuad are routines that need
   // to be provided by the user based
   // on the method of accessing the PCI registers in their system.
   //
   WriteLynxQuad (MPCI_MISCTRL_REG, SOFTWARE_RESET);
   //
   // Set the GRF/ATF/ITF FIFO Sizes.  The General Receive FIFO
   // (GRF),  Asynchronous Transmit FIFO
   //  (ATF) and the Isochronous Transmit FIFO ( ITF).
   //
   WriteLynxQuad (FIFO_SIZES, DEFAULT_GRF_SIZE |
                             (DEFAULT_ATF_SIZE << 8) |
                             (DEFAULT_ITF_SIZE << 16));

   //
   // Set the transmit FIFO thresholds.  The thresholds signal the
   // DMA engines when to start processing
   // data.  The DMA starts processing data after the threshold in
   // the FIFO is passed.
   //
   WriteLynxQuad (FIFO_XMT_THRESHOLD,(ATF_THRESHOLD << 8) | ITF_THRESHOLD);
   //
   // Set the Overrun and Underrun counters
   //
   WriteLynxQuad (LLC_FIFO_OVRFL, 0x00FE0000L);
  //
  //  Read the revision of the PCILynx chip set.  There are some
  //   certain functions that need to be performed
  //   if the chipset is a rev A.
  //
  m_Rev = ReadLynxQuad (MPCI_REV_ID_REG);
   //
   // Only do this if below rev. A.
   //
   if ((m_Rev & 0xFF) < 0x2) {
      WriteLynxQuad (MPCI_MISCTRL_REG, ENA_POST_WR | ENA_SLV_BURST);
   }
   //
   // Set the Lynx Control Register.  Set the Lynx Control register
   // to enable isochronous transfers
   //
   WriteLynxQuad (LLC_CONTROL, TX_ISO_EN | RX_ISO_EN |
                               TX_ASYNC_EN | RX_ASYNC_EN |
                               CYCMASTER | CYCTIMEREN |
                               RCV_COMP_VALID);
   //
   // Set up the DMA Async receive buffers.  These must be set up
   // before you can receive ANY
   // asynchronous  packets.
   //
   if (!DMAReceiveBusEvents(PKT_RCV_AND_UPDATE,
                        (CHAN_NEEDS_ADDR|CHAN_RCV|CHAN_CIRCULAR|CHAN_SELFID)))
      // Print status or error message here for debug
      return FALSE;
   //
   // Enable Bus Reset, Over/Underflow, and PCL interrupts
   //
   WriteLynxQuad (LLC_INT_MASK, INTERRUPT_TRAP_MASK);
   WriteLynxQuad (MPCI_INTERRUPT_ENA, DMA_PCL_MSK | DMA_HLT_MSK | LLC_INT);
   //
   // Set the PCILynx automatic retry count to 5 and retry
   //   delay to 2 Iso Intervals
   WriteLynxQuad (DMA_BUSY_RETRY, 0x0205);
   //
   // A bus reset should be invoked here if it is desired to
   //   become the root on the bus.

   return TRUE;
}
```

Setting Up a DMA Channel for Receiving of 1394 Bus Events.

This section describes the routines to handle asynchronous bus events from the 1394 bus. This routine will hook a PCL to a DMA channel, setup the comparator registers and setup the PCLs to move that data into internal buffers.

```
///////////////////////////////////////////////////////////////////////////////
//
//  Function:    DMAReceiveBusEvents
//  Description: Implements DMA Receive Bus Events function by constructing
//               PCLs to receive async packets into a ring of MAX_PCLS
//               buffers, each of MAX_GNRLRECV_SIZE size.  After each
//               packet is received into a buffer, the PCL generates an
//               interrupt so that the ISR can cause an indication callback
//               to be performed.
//
//  Input: command    -   PKT_RCV
//         flags      -   Any specific flags
//
//  Output: TRUE == SUCCESS
////////////////////////////////////////////////////////////////////////////////

DMAReceiveBusEvents(ULONG command, ULONG flags)
{
   PCL_S *prevPCL ;  // Linked list last pointer to the previous PCL
   ULONG headPhys;          // Physical memory Head Pointer
   PCL_S *pPCL;        //  PCL pointer
   PCL_S *dummy;             // Dummy PCL linear memory pointer
   int   currentPCL;   // Currently working on PCL.
   ULONG startPCL;  // Starting PCL chain.
   ULONG physicalAddr; // A physical address
   ULONG dummyPhysAddr;     // Dummy PCL physical pointer
   PHYSICAL_ADDRESS maxPhysicalAddress;
   ULONG *pReceiveBuffers[MAX_GNRLRECV_PCLS];

   //
   //  Find a channel for DMA with the same flags.  The FLAGs is used to
   //  determine if a channel exists to
   //  do asynchronous, isochronous receive or transmit.   The FindDMAChannel
   //  is a local routine that can
   //  be used to keep track of the DMA channels that have not been used.  Any
   //  of the 5 DMA channels can
   //  be used for the asynchronous transfers.  If you want you can just place
   //  a static channel number here.
   //
   int channel = FindDMAChannel(flags, FALSE);
   if (channel == -1)
      //  Print an error or status message stating that there is no channel
      //  available
      return FALSE;
   //
   // Store off this channel in a global variable.  This is used by the
   // ISR to determine that an async packet has arrived (rather then ISO)
   //
   m_BusEventChannel = channel;
   /*-------------------------------------------------------------
```

```
 *  Allocate an array of MAX_GNRLRECV_PCLS buffers each of
 *  MAX_SELFID_PKT_BUFFER_SIZE in size from the heap.  These
 *  will form the ring of buffers used to receive all asynchronous
 *  packets.
 *-------------------------------------------------------------------
 */

//
// Buffer large enough for all Receive buffers,
// used for DMA - non-paged, cache aligned, contiguous
// The DMA of the PCILynx uses contiguous memory to transfer data.
//  The MmAllocateContiguous memory is a function call that will be
//  unique to your operating system.
//
maxPhysicalAddress.LowPart = 0xFFFFFF;
maxPhysicalAddress.HighPart = 0x0;
m_DMAMemory = (ULONG*) MmAllocateContiguousMemory (
                       MAX_SELFID_PKT_BUFFER_SIZE*MAX_GNRLRECV_PCLS,
                       maxPhysicalAddress);
if (m_DMAMemory == NULL)
   // Print error or status messages here
   return (FALSE);
//
// Make sure address is valid
//
if (!MmIsAddressValid (m_DMAMemory)) {
   // Print error and status messages here.
   // A routine that frees contiguous memory for the applications OS
   // should be used here.
   MmFreeContiguousMemory (m_DMAMemory);
   return (FALSE);
}
//  A routine that obtains contiguous memory for the applications OS
//  should be used here.
m_DMAPhysicalMemory = MmGetPhysicalAddress (m_DMAMemory);

//
// MDL for flushing DMA'ed data.  MDL is a Windows operating system
//  construct for referencing physical memory.
//
m_pDMAMdl = IoAllocateMdl (m_DMAMemory,
                           MAX_SELFID_PKT_BUFFER_SIZE*MAX_GNRLRECV_PCLS,
                           FALSE, FALSE, NULL);
MmBuildMdlForNonPagedPool (m_pDMAMdl);   // Use applications routine here.
//
// Now a PCL lists will be created.
//
m_PCLList = (PCL_LIST_S*) ExAllocatePool (NonPagedPool,
                          MAX_GNRLRECV_PCLS * sizeof (PCL_LIST_S));
if (m_PCLList == NULL) {
   //  Print error and status messages here
   MmFreeContiguousMemory  (m_DMAMemory);
   return (FALSE);
}

//
// Setup the indices into receive buffer
//
for (currentPCL=0; currentPCL < MAX_GNRLRECV_PCLS; currentPCL++) {
   pReceiveBuffers[currentPCL] = m_DMAMemory +
                  (MAX_SELFID_PKT_BUFFER_SIZE * currentPCL) / sizeof(ULONG);
```

```
}

//
// Map Receive Async PCLs to these buffers.  The user must decide what size
// Asynchronous packets they
// might be receiving and allocate enough memory to take care of the maximum
// size.  If the packets smaller then more can be mapped.
//
prevPCL = NULL;
headPhys = 0;
pPCL = NULL;
for (currentPCL = 0; currentPCL < MAX_GNRLRECV_PCLS; currentPCL++) {
   pPCL = (PCL_S*) AllocPCL(&physicalAddr, &m_PCLList[currentPCL].startPCL);
   if (pPCL == NULL) {
      // Print status messages here
      MmFreeContiguousMemory (m_DMAMemory);
      ExFreePool (m_PCLList); // This is a memory management local function.
      return (FALSE);
   }

   pPCL->transferCount = 0;
   pPCL->bufferAddress = 0;
   pPCL->software      = 0;

   if (prevPCL) {
      prevPCL->nextPcl = prevPCL->errorPcl = physicalAddr;
   }
   else {
      headPhys = physicalAddr;
   }
   prevPCL = pPCL;

   //
   // No need to scatter lock the buffer, in contiguous non-paged memory.
   // Now the PCL structures are being setup for the receive of the packets.
   // See the PCILynx functional specification for the definition of the
   // PCL bit definitions and the PCL structures.
  //
   pPCL->pairs[0].control.bits.rsvd28 = 0;
   pPCL->pairs[0].control.bits.rsvd23 = 0;
   pPCL->pairs[0].control.bits.transferCount = MAX_SELFID_PKT_BUFFER_SIZE;
   pPCL->pairs[0].control.bits.isochronousMode = 0;
   pPCL->pairs[0].control.bits.multipleIsochronous = FALSE;
   pPCL->pairs[0].control.bits.transmitSpeed = 0;
   pPCL->pairs[0].control.bits.bigEndian = 1;
   pPCL->pairs[0].control.bits.waitForStatus = 0;
   pPCL->pairs[0].control.bits.waitSelect = 0;
   pPCL->pairs[0].control.bits.command = 0;
   pPCL->pairs[0].data = (MmGetPhysicalAddress
                                  (pReceiveBuffers[currentPCL])).LowPart;
   pPCL->pairs[0].control.bits.doneInterrupt = 0;
   pPCL->pairs[0].control.bits.lastBuffer = 1;
   pPCL->status.quad = 0;

   //
   // Set PCL command (RECEIVE) and tell PCL to interrupt on each async
   // packet so that an indication callback can be made.
   //

   pPCL->pairs[0].control.bits.doneInterrupt = 1;
   pPCL->pairs[0].control.bits.command = command;
```

```
      //
      // Add it to the m_PCLList....so that we can clean up at a later time
      // and also so that the bus event handler can pass a linear pointer to
      // the packet up to the bus manager at indication time
     //
      m_PCLList[currentPCL].pPCL = pPCL;
      m_PCLList[currentPCL].pBuffer = pReceiveBuffers[currentPCL];
      }

     if (flags & CHAN_CIRCULAR) {
        pPCL->errorPcl = pPCL->nextPcl = headPhys;
     }
     else {
       pPCL->errorPcl = pPCL->nextPcl = 1;
     }

     //
     //  A dummy PCL is setup for initial linkage to the DMA channel.  See the
     //  PCILynx functional specification for the usage of the dummy PCL
     //  packet.
     dummy = (PCL_S*) AllocPCL(&dummyPhysAddr, &startPCL);
     dummy->nextPcl = dummy->errorPcl = headPhys;

     m_BusResetInfo.receiveSelfIdPCL = dummyPhysAddr;

     //
     //  “Link” the PCL chain that has been created to an available DMA channel
     //
     LinkPPcl (channel, command,
            (CHAN_NEEDS_ADDR|CHAN_RCV|CHAN_CIRCULAR|CHAN_SELFID),
             dummyPhysAddr, TRUE);
   return (TRUE);
}
```

Linking a PCL Chain to a DMA Channel

This function will take a completed PCL chain and link it to one of the available DMA channels. The PCL chain must have been allocated in physical memory. The chain must have a dummy PCL packet that is used to point to the actual PCL chain that will be used to transfer the packets to memory.

```
/////////////////////////////////////////////////////////////////////////
//  Function: LinkPPcl
//  Description: Links a PCL chain to a given DMA channel
//
//  Input: channel    -   channel
//         command    -   PKT_XMT or PKT_LOCK
//         flags      -   Any specific flags
//         physAddr   -   Physical Address of PCL chain head
//         bSelfID    -
/////////////////////////////////////////////////////////////////////////

LinkPPcl(long channel, ULONG command,
         ULONG flags, ULONG physAddr, BOOLEAN bSelfID)
{
   // The DMA channel control register has only 2 writeable bits: DMA_LINK
   // and DMA_CHAN_ENA. The rest are read-only. Will NOT enable a disabled
   // channel UNLESS CHAN_NEEDS_ENABLE flag is set in the channel.
   // If the user wants to specify a DMA channel that is currently in use
   // update the current dmaCurrentPCL
   if (m_DMAChannels[channel].flags & CHAN_NEEDS_ADDR) {
      *(m_DMAChannels[channel].dmaCurrentPCL) = physAddr;
   }
   // The DMA engine will not begin transmitting unless the comparator
   // registers match on of the incoming fields of the 1394 packet.
   // If channel will not be used for a transmit operation, then
   // enable the receive comparators for the channel just before
   // starting the DMA engine processing of this DMA channel
   WriteLynxQuad (LLC_INT_MASK, INTERRUPT_TRAP_MASK);
   WriteLynxQuad (MPCI_INTERRUPT_ENA, DMA_PCL_MSK | DMA_HLT_MSK | LLC_INT);
   if (command != PKT_XMT) {
      if (bSelfID) {
          *(m_DMAChannels[channel].dmaWord0Value)=(m_BusResetInfo.nodeID) << 16
          *(m_DMAChannels[channel].dmaWord0Mask)  = CMP_W0_DEST_ID_MSK | 0xA0;
          *(m_DMAChannels[channel].dmaWord1Value) = 0;
          *(m_DMAChannels[channel].dmaWord1Mask)  = CMP_W1_SELF_ID_ENA    |
                                                    CMP_W1_ENA_CH_COMPARE |
                                                    WRITE_REQ_ACK_SEL     |
                                                    MATCH_BUS_AND_NODE    |
                                                    MATCH_3FF_AND_NODE    |
                                                    MATCH_BUS_AND_3F      |
                                                    MATCH_BROADCAST;
      }
      else {
          *(m_DMAChannels[channel].dmaWord1Mask)  = CMP_W1_ENA_CH_COMPARE;
      }
   }
   *(m_DMAChannels[channel].dmaControl) = DMA_CHAN_ENA | DMA_LINK;
   WriteLynxQuad (MPCI_INTERRUPT_ENA, DMA_PCL_MSK | DMA_HLT_MSK | LLC_INT);
}
```

Processing a 1394 Bus Reset

When a device is hot plugged into the 1394 bus a bus reset occurs. At this time the bus is automatically re-configured. Each device then arbitrates and the nodes can be re-numbered dynamically. This section of software handles the bus reset. It receives the self-id packets, parses them and checks them for correctness.

The PCILynx collects ALL of the small self-id packets (called runt packets in this source code) and places them into one asynchronous packet that will be transferred into the memory buffer that has been setup to receive asynchronous packets.

```
//////////////////////////////////////////////////////////////////////////
//
//  Function: ProcessBusReset ( ULONG *pktPtr, int length )
//  Description: Begins the bus reset processing
//
//  Input: pktPtr -  Pointer to the self-id packets.
//             length -  The length of the selfid packet.
//
//////////////////////////////////////////////////////////////////////////
void  ProcessBusReset(ULONG *pktPtr, int length)
{
   int offset;
   int count = 0;
   int foundCount = 0;
   m_BusResetInfo.busGeneration++;
   DMA_CHANNEL_S *pChannel;
   //
   // Copy runt packets out of self-id packet and place them in an array
   // for use by the bus manager (also store inverses)
   //
   for (offset=0; (offset<length/4) && (offset<MAX_SELF_ID_PACKETS);offset++) {
      m_BusResetInfo.runtPackets[offset] = *(pktPtr+offset);
   }
   if (length/8 > 0) {
      SetNodeId(&m_BusResetInfo, length/4);
   }
   else {
      m_BusResetInfo.numberNodes = 1;
      m_BusResetInfo.nodeID = OurBusID << 6;
      WriteLynxQuad (LLC_CONTROL, (TX_ISO_EN | RX_ISO_EN | TX_ASYNC_EN |
                             RX_ASYNC_EN | CYCMASTER |
                             CYCTIMEREN | RCV_COMP_VALID));
      WriteLynxQuad (LLC_NODE_ADDR, m_BusResetInfo.nodeID << 16);
   }
   pChannel = &m_DMAChannels[m_BusEventChannel];
   *(pChannel->dmaCurrentPCL) = 1;

   //
   // Add some delay to make sure the channel is not busy.    Add approximately
   //  4 Isochronous cycles  ( 1 ISO cycle is appr. 125 microsecs. )
   //
   ctDelay (4);   // Whatever delay your system supports should be placed here.

   *(pChannel->dmaControl) = 0;
   pChannel->busy = FALSE;
   WriteLynxQuad (FIFO_TEST, ATF_FLUSH);

   LinkPPcl(m_BusEventChannel, PKT_RCV,
           (CHAN_NEEDS_ADDR|CHAN_RCV|CHAN_CIRCULAR|CHAN_SELFID),
            m_BusResetInfo.receiveSelfIdPCL, TRUE);
}
```

Summary

This document is only a starting point for programming the PCILynx 1394 link-layer controller. This document attempts to assist the programmer in beginning to program the chip. The programmer will need to obtain the PCILynx functional specification as well as the 1394-1995 Serial Bus Specification to properly program this device to conform to the 1394 serial bus.

APPENDIX A: Header File of Register Definitions

This appendix contains a header file of register definitions for the PCILynx 1394 chipset.

```
// PCI hardware registers
#define MPCI_VENDOR_REG         0       // 0x00/4
#define MPCI_COMMAND_REG        1       // 0x04/4
#define MPCI_REV_ID_REG         2       // 0x08/4
#define MPCI_CACHE_REG          3       // 0x0c/4
#define MPCI_BASE10_REG         4       // 0x10/4
#define MPCI_BASE14_REG         5       // 0x14/4
#define MPCI_BASE18_REG         6       // 0x18/4
#define MPCI_BASE1C_REG         7       // 0x1c/4
#define MPCI_BASE20_REG         8       // 0x20/4
#define MPCI_BASE24_REG         9       // 0x24/4
#define MPCI_RSVD_1_REG         0xa     // 0x28/4
#define MPCI_SUBSYSTEM_ID       0xb     // 0x2c/4
#define MPCI_ROM_ADR_REG0       0xc     // 0x30/4
#define MPCI_RSVD_2_REG         0xd     // 0x34/4
#define MPCI_RSVD_3_REG         0xe     // 0x38/4
#define MPCI_INTLINE_REG        0xf     // 0x3c/4
#define MPCI_MISCTRL_REG        0x10    // 0x40/4
#define MPCI_I2C_EEPROM         0x11    // 0x44/4
#define MPCI_INTERRUPT_STAT     0x12    // 0x48/4
#define MPCI_INTERRUPT_ENA      0x13    // 0x4c/4
#define MPCI_TEST               0x14    // 0x50/4
#define MPCI_LOCALBUS_CTRL      0x2c    // 0xb0/4
#define MPCI_LOCALBUS_ADDR      0x2d    // 0xb4/4
#define MPCI_GPIO_REGA          0x2e    // 0xb8/4
#define MPCI_GPIO_REGB          0x2f    // 0xbc/4
#define MPCI_GPIO_NOP           0x30    // 0xc0/4
#define MPCI_GPIO_0             0x31    // 0xc4/4
#define MPCI_GPIO_1             0x32    // 0xc8/4
#define MPCI_GPIO_10            0x33    // 0xcc/4
#define MPCI_GPIO_2             0x34    // 0xd0/4
#define MPCI_GPIO_20            0x35    // 0xd4/4
#define MPCI_GPIO_21            0x36    // 0xd8/4
#define MPCI_GPIO_210           0x37    // 0xdc/4
#define MPCI_GPIO_3             0x38    // 0xe0/4
#define MPCI_GPIO_30            0x39    // 0xe4/4
#define MPCI_GPIO_31            0x3a    // 0xe8/4
#define MPCI_GPIO_310           0x3b    // 0xec/4
#define MPCI_GPIO_32            0x3c    // 0xf0/4
#define MPCI_GPIO_320           0x3d    // 0xf4/4
#define MPCI_GPIO_321           0x3e    // 0xf8/4
#define MPCI_GPIO_3210          0x3f    // 0xfc/4
#define DMA_CHAN_0_PREV_PCL     0x40    // 0x100/4
#define DMA_CHAN_0_CURR_PCL     0x41    // 0x104/4
#define DMA_CHAN_0_CURR_DAT     0x42    // 0x108/4
#define DMA_CHAN_0_STATUS       0x43    // 0x10c/4
#define DMA_CHAN_0_CTRL         0x44    // 0x110/4
#define DMA_CHAN_0_READY        0x45    // 0x114/4
#define DMA_CHAN_0_STATE        0x46    // 0x118/4
//                              0x47    // 0x11c/4 rsvd
#define DMA_CHAN_1_PREV_PCL     0x48    // 0x120/4
#define DMA_CHAN_1_CURR_PCL     0x49    // 0x124/4
#define DMA_CHAN_1_CURR_DAT     0x4a    // 0x128/4
#define DMA_CHAN_1_STATUS       0x4b    // 0x12c/4
```

```
#define DMA_CHAN_1_CTRL          0x4c    // 0x130/4
#define DMA_CHAN_1_READY         0x4d    // 0x134/4
#define DMA_CHAN_1_STATE         0x4e    // 0x138/4
//                               0x4f    // 0x13c/4 rsvd
#define DMA_CHAN_2_PREV_PCL      0x50    // 0x140/4
#define DMA_CHAN_2_CURR_PCL      0x51    // 0x144/4
#define DMA_CHAN_2_CURR_DAT      0x52    // 0x148/4
#define DMA_CHAN_2_STATUS        0x53    // 0x14c/4
#define DMA_CHAN_2_CTRL          0x54    // 0x150/4
#define DMA_CHAN_2_READY         0x55    // 0x154/4
#define DMA_CHAN_2_STATE         0x56    // 0x158/4
//                               0x57    // 0x15c/4 rsvd
#define DMA_CHAN_3_PREV_PCL      0x58    // 0x160/4
#define DMA_CHAN_3_CURR_PCL      0x59    // 0x164/4
#define DMA_CHAN_3_CURR_DAT      0x5a    // 0x168/4
#define DMA_CHAN_3_STATUS        0x5b    // 0x16c/4
#define DMA_CHAN_3_CTRL          0x5c    // 0x170/4
#define DMA_CHAN_3_READY         0x5d    // 0x174/4
#define DMA_CHAN_3_STATE         0x5e    // 0x178/4
//                               0x5f    // 0x17c/4 rsvd
#define DMA_CHAN_4_PREV_PCL      0x60    // 0x180/4
#define DMA_CHAN_4_CURR_PCL      0x61    // 0x184/4
#define DMA_CHAN_4_CURR_DAT      0x62    // 0x188/4
#define DMA_CHAN_4_STATUS        0x63    // 0x18c/4
#define DMA_CHAN_4_CTRL          0x64    // 0x190/4
#define DMA_CHAN_4_READY         0x65    // 0x194/4
#define DMA_CHAN_4_STATE         0x66    // 0x198/4
//                               0x67    // 0x19c/4 rsvd
//                               0x68    // 0x1a0/4 rsvd
//                               ...
//                               0x23f   // 0x8e0/4 rsvd
#define DMA_DIAG_TEST            0x240   // 0x900/4
#define DMA_RCV_PKT_LEFT         0x241   // 0x904/4
#define DMA_GLOBAL_FLAGS         0x242   // 0x908/4
#define FIFO_SIZES               0x280   // 0xa00/4
#define FIFO_PCI_PTR             0x281   // 0xa04/4
#define FIFO_LINK_PTR            0x282   // 0xa08/4
#define FIFO_TOKEN               0x283   // 0xa0c/4
#define FIFO_TEST                0x284   // 0xa10/4
#define FIFO_XMT_THRESHOLD       0x285   // 0xa14/4
//                               0x286   // 0xa18/4
//                               0x287   // 0xa1c/4
// These regs should not be read except in special cases to
//  prevent side-effects.
#define FIFO_GRF_PUSH_POP_0      0x288   // 0xa20/4
#define FIFO_GRF_PUSH_POP_1      0x289   // 0xa24/4
//                               0x28a   // 0xa28/4
//                               0x28b   // 0xa2c/4
#define FIFO_ATF_PUSH_POP_0      0x28c   // 0xa30/4
#define FIFO_ATF_PUSH_POP_1      0x28d   // 0xa34/4
//                               0x28e   // 0xa38/4
//                               0x28f   // 0xa3c/4
#define FIFO_ITF_PUSH_POP_0      0x290   // 0xa40/4
#define FIFO_ITF_PUSH_POP_1      0x291   // 0xa44/4
//                               0x292   // 0xa48/4
//                               0x293   // 0xa4c/4
#define DMA_CHAN_0_WORD0_VALU    0x2c0   // 0xb00/4
#define DMA_CHAN_0_WORD0_MASK    0x2c1   // 0xb04/4
#define DMA_CHAN_0_WORD1_VALU    0x2c2   // 0xb08/4
#define DMA_CHAN_0_WORD1_MASK    0x2c3   // 0xb0c/4
#define DMA_CHAN_1_WORD0_VALU    0x2c4   // 0xb10/4
```

```
#define DMA_CHAN_1_WORD0_MASK   0x2c5   // 0xb14/4
#define DMA_CHAN_1_WORD1_VALU   0x2c6   // 0xb18/4
#define DMA_CHAN_1_WORD1_MASK   0x2c7   // 0xb1c/4
#define DMA_CHAN_2_WORD0_VALU   0x2c8   // 0xb20/4
#define DMA_CHAN_2_WORD0_MASK   0x2c9   // 0xb24/4
#define DMA_CHAN_2_WORD1_VALU   0x2ca   // 0xb28/4
#define DMA_CHAN_2_WORD1_MASK   0x2cb   // 0xb2c/4
#define DMA_CHAN_3_WORD0_VALU   0x2cc   // 0xb30/4
#define DMA_CHAN_3_WORD0_MASK   0x2cd   // 0xb34/4
#define DMA_CHAN_3_WORD1_VALU   0x2ce   // 0xb38/4
#define DMA_CHAN_3_WORD1_MASK   0x2cf   // 0xb3c/4
#define LLC_NODE_ADDR           0x3c0   // 0xf00/4
#define LLC_CONTROL             0x3c1   // 0xf04/4
#define LLC_CYC_TIMER           0x3c2   // 0xf08/4
#define LLC_PHY_REGS            0x3c3   // 0xf0c/4
#define LLC_DIAG_TEST           0x3c4   // 0xf10/4
#define LLC_INTERRUPT           0x3c5   // 0xf14/4
#define LLC_INT_MASK            0x3c6   // 0xf18/4
#define DMA_BUSY_RETRY          0x3c7   // 0xf1c/4
#define LLC_STATE_VECTOR        0x3c8   // 0xf20/4
#define LLC_FIFO_OVRFL          0x3c9   // 0xf24/4
// END OF MEMORY REGS --------------------------------------



#define SOFTWARE_RESET          0x00000001

#define TOTAL_FIFO_SIZE         256
#define DEFAULT_ATF_SIZE        TOTAL_FIFO_SIZE/4
#define DEFAULT_ITF_SIZE        TOTAL_FIFO_SIZE/4
#define DEFAULT_GRF_SIZE        TOTAL_FIFO_SIZE/2       // twice as big GRF
#define ATF_THRESHOLD           32
#define ITF_THRESHOLD           32

#define RX_ASYNC_EN             0x00800000L
#define TX_ASYNC_EN             0x01000000L
#define RX_ISO_EN               0x02000000L
#define TX_ISO_EN               0x04000000L
#define RCV_COMP_VALID          0x00000080L
#define CYCTIMEREN              0x00000200L
#define CYCMASTER               0x00000800L

#define CONFIG_ROM_SIZE             256     // Size of Lynx EEPROM in Bytes
#define BUS_INFO_BLOCK_SIZE         20      // Size of BusInfoBlock CSR Section
#define SERIAL_CONFIG_ROM_OFFSET    0x10

#define DMA_CHAN_ENA            0x80000000
#define DMA_LINK                0x20000000
#define CMP_W1_ENA_CH_COMPARE   0x00000100

#define LLC_INT                0x00010000L
#define DMA4_PCL               0x00000200L
#define DMA4_HLT               0x00000100L
#define DMA3_PCL               0x00000080L
#define DMA3_HLT               0x00000040L
#define DMA2_PCL               0x00000020L
#define DMA2_HLT               0x00000010L
#define DMA1_PCL               0x00000008L
#define DMA1_HLT               0x00000004L
#define DMA0_PCL               0x00000002L
#define DMA0_HLT               0x00000001L
```

```
#define DMA_PCL_MSK            (DMA4_PCL | DMA3_PCL | DMA2_PCL |\
                                DMA1_PCL | DMA0_PCL)
#define DMA_HLT_MSK            (DMA4_HLT | DMA3_HLT | DMA2_HLT |\
                                DMA1_HLT | DMA0_HLT)
#define ATF_FLUSH              0x00000004

#define MAX_SELF_ID_PACKETS         256
    // Max Self ID Packets (including inverses)
#define MAX_GNRLRECV_PCLS           100
    //# of General Receive PCLS to be built
#define MAX_SELFID_PKT_BUFFER_SIZE  2048
#define MAX_CONNECTIONS             8
    // Number of callbacks stored

#define INIT_CHANNELS_AVAIL    0xffffffff
#define INIT_BANDWIDTH         4915
#define INIT_BM_ID             0x3f

#define CHAN_XMT               PKT_XMT
#define CHAN_RCV               PKT_RCV
#define CHAN_ISO               0x10
#define CHAN_MULTI_ISO         0x20
#define CHAN_CIRCULAR          0x40
#define CHAN_SELFID            0x80
#define CHAN_NEEDS_ADDR        0x4000
#define CHAN_NEEDS_ENABLE      0x8000

#define WRPHY                  0x40000000

#define INTERRUPT_TRAP_MASK   (LLCI_PHY_BUSRESET | LLCI_GRF_OFLOW |\
                               LLCI_ITF_UFLOW    | LLCI_ATF_UFLOW)

#define BusReset() \
        WriteLynxQuad(LLC_PHY_REGS, WRPHY | 0x01ff0000); // Bus Reset


// Control bits
typedef union {
   struct {
      //LSB...MSB order below. If not marked "*ALL pairs" then only for pair[0]
      unsigned transferCount       : 12; // xfer_cnt  *ALL pairs
      unsigned isochronousMode     :  1; // iso_mode
      unsigned multipleIsochronous :  1; // multi_iso
      unsigned transmitSpeed       :  2; // xmit_speed 00=100mbps, 01=200mbps
      unsigned bigEndian           :  1; // big_endian *ALL pairs
      unsigned waitForStatus       :  1; // wait_for
      unsigned lastBuffer          :  1; // last_buf *ALL pairs
      unsigned doneInterrupt       :  1; // done_int
      unsigned waitSelect          :  3; // wait_sel
      unsigned rsvd23              :  1;
      unsigned command             :  4; // cmd
      unsigned rsvd28              :  4;
   } bits;
   ULONG quad;
} PCL_CONTROL_U;

// Status bits
typedef union {
   struct {
      //LSB...MSB order below. If not marked "*ALL pairs" then only for pair[0]
      unsigned transferCount       : 13;
      unsigned rsvd13              :  1;
      unsigned acknowledgeType     :  1;
      unsigned packetAcknowledge   :  4;
      unsigned receiveSpeed        :  2;
      unsigned dmaChannel          :  6;
      unsigned packetComplete      :  1;
      unsigned packetError         :  1;
      unsigned masterError         :  1;
      unsigned isochronousMode     :  1;
      unsigned selfIDPacket        :  1;
   } bits;
   ULONG quad;
} PCL_STATUS_U;

// The MAIN PCL structure for LYNX. This data structure MUST exist in PHYSICAL
// memory (i.e. must be mapped so that the PHYSICAL address can be obtained )
typedef struct {
   ULONG  nextPcl;       // next_pcl
   ULONG  errorPcl;      // error_pcl
   ULONG  software;      // software
   PCL_STATUS_U status;
   ULONG  transferCount; // rcvau_cnt
   ULONG  bufferAddress; // rcvau_buf
   struct {
      PCL_CONTROL_U control;  // ctrl
      ULONG data;        // XFER PCL: physical DMA address of data
   } pairs[3];
     // Assumes that the buffer will only occupy 2 physical addresses
} PCL_S;

// PCL Linked List
typedef struct {
   PCL_S *pPCL;
   ULONG startPCL;
   ULONG *pBuffer;
} PCL_LIST_S, PPCL_LIST_S;


typedef ULONG RUNT_PACKET_T;
typedef ULONG LYNX_NODE_ID_T;
typedef ULONG LYNX_CYCLE_TIME_T;

typedef struct {
   ULONG busGeneration;
   LYNX_NODE_ID_T nodeID;
   LYNX_CYCLE_TIME_T currentTime;
   ULONG numberNodes;
   RUNT_PACKET_T runtPackets[MAX_SELF_ID_PACKETS];
   ULONG receiveSelfIdPCL;
   HANDLE  BusResetCompleteEvent;
   BOOLEAN BusResetComplete;
} BUS_RESET_INFORMATION, *PBUS_RESET_INFORMATION;


typedef struct {
   char name[MAX_DMA_NAME_LENGTH]; // descriptive channel name
   int  flags; // current channel descriptive flags. See CHAN_ISO, etc above.

   /*-----------------------------------------
    * Direct access to dma register locations
    *-----------------------------------------
    */

   // Control Registers
   ULONG *dmaPreviousPCL;
   ULONG *dmaCurrentPCL;
   ULONG *dmaCurrentBuffer;
   ULONG *dmaStatus;
   ULONG *dmaControl;
   ULONG *dmaReady;
   ULONG *dmaState;

   // Compare and mask registers
   ULONG *dmaWord0Value;
   ULONG *dmaWord0Mask;
   ULONG *dmaWord1Value;
   ULONG *dmaWord1Mask;

   ULONG dmaWord0ValuePhysical;
   ULONG dmaWord0MaskPhysical;

   BOOLEAN busy;
} DMA_CHANNEL_S;


typedef struct {
   HANDLE  hEvent; // Event to set
   int     entry;

   // Bus Reset callback info
   BOOLEAN fBusReset;
   BOOLEAN fWantBusReset;
   BUS_RESET_INFORMATION busResetInfo;

   // Receive notification info
   BOOLEAN fIndication;
   int     nextWriteLocation;
   int     nextReadLocation;
   UCHAR   buffer[CALLBACK_BUFFER_SIZE];
} CALLBACKSTRUCT, *PCALLBACKSTRUCT;

typedef struct _BUS_ENUMERATION_INFORMATION {
   ULONG speed;
   ULONG physicalID;
   ULONG contender;
   ULONG portOne;
   ULONG portTwo;
   ULONG portThree;
   ULONG IDidIt;
} BUS_ENUMERATION_INFORMATION, *PBUS_ENUMERATION_INFORMATION;


BUS_RESET_INFORMATION m_BusResetInfo;
```