Technical Notes on using Analog Devices' DSP components and development tools
Phone: (800) ANALOG-D, FAX: (781) 461-3010, EMAIL: dsp.support@analog.com, FTP: ftp.analog.com, WEB: www.analog.com/dsp

Copyright 1999, Analog Devices, Inc. All rights reserved. Analog Devices assumes no responsibility for customer product design or the use or application of customers' products or for any infringements of patents or rights of others which may result from Analog Devices assistance. All trademarks and logos are property of their respective holders. Information furnished by Analog Devices Applications and Development Tools Engineers is believed to be accurate and reliable, however no responsibility is assumed by Analog Devices regarding the technical accuracy of the content provided in all Analog Devices' Engineer-to-Engineer Notes.

SHARC Bits

SHARC Link Port Booting

Last Modified: *8/21/97*

OVERVIEW:

This Engineer's Note will discuss the link boot process from both the perspective of the master device and the slave SHARC. One characteristic which makes link port booting different from EPROM and host booting is the handling of the link control register, LCTL, during the booting mode. Unlike the other booting processes where the control registers are initialized to meaningful values, the link port control register is initialized to zero but it is ignored during the initial booting process. This process results in complications which must be accounted for during the booting process. Details of the complications along with methods of handling them will be presented.

A link port boot example is provided with this engineers note. The example consists of code for the master SHARC which will transfer the boot code to a slave SHARC through the link ports. The link port boot kernel executed on the booting SHARC along with a simple test program is also provided.

The first section of the engineers note describes the general booting process of the SHARC. Section 2 discusses the details of booting through the link port. Section 3 discusses the hardware issues associated with link port booting and how to accommodate for these issues in software. Section 4 provides a step by step example of link port booting including documented code.

SHARC Booting Process

The term booting refers to the initial downloading of data and code into the SHARC processor so it may then begin instruction execution. There are three possible booting modes for the SHARC. The SHARC can be booted by reading data from an EPROM through its external port. The SHARC can be booted by a host processor writing data to the SHARC through its external port, or the SHARC can be booted by another processor writing data to the SHARC through its link ports. The SHARC can also be placed in a no boot mode after reset. In the no boot mode the SHARC begins executing instructions form external memory.

The booting mode of the SHARC is configured in hardware by three pins: EBOOT, LBOOT and BMS. Upon power up, the SHARC will enter the appropriate boot mode based on the setting of the EBOOT, LBOOT and BMS pins. Table 1 describes the pin configurations for each boot mode (EPROM boot, Host boot, Link boot and No boot):

EBOOT	LBOOT	BMS	Booting Mode
1	0	Output	EPROM
0	0	1	Host
0	1	1	Link Port
0	0	0	No Boot

Table 1. Boot mode pin configuration

On power up the SHARC will be configured for a 256 word Direct Memory Access (DMA) transfer through either the external port, for EPROM or Host boot, or through the Link Port for Link boot. (No boot begins executing code in external memory no DMA is required.) The registers associated with DMA Channel 6 are initialized to perform the 256 word DMA through the appropriate port to internal memory starting at location 0x20000 and ending at location 0x200FF.

ANALOG DEVICES

THE DMAC6 register is also initialized at power up based on the boot mode selected. Table 2 matches the DMAC6 initial setting to the corresponding boot mode:

Booting Mode	DMAC6
EPROM	0x02A1
Host	0x00A1
Link Port	0x00A0

Table 2. Control Register DMAC6 Initial value

All three booting modes initialize DMAC6 for instruction word transfers and 16-48 bit packing. (The packing mode is ignored during EPROM booting which performed 8-48 bit packing and link port booting which packs nibbles into 48 bit words.) In Host and EPROM boot mode DMAC6 is set to enable a DMA through the external port. In the EPROM boot mode DMAC6 is also configured for a master mode DMA (see chapter 6 of the SHARC User's Manual for an explanation of master mode DMA).

After reset the program counter (PC) will be pointing to location 0x20004 where it will be executing an idle instruction. Upon completion of the 256 word DMA the DMA channel 6 interrupt, EPB0I, will be latched and the PC will jump to location 0x20040. At this point the SHARC will begin executing instruction starting with the instruction at location 0x20040. If the instruction at location 0x20040 is an RTI, which it will be if the boot loader kernels provided are used, the PC will jump to location 0x20005 and begin executing code in a linear flow.

If the program and data to be booted into the SHARC is greater than 256 words, the remaining code and data must be loaded in by the SHARC and using the first 256 words. For each booting mode Analog Devices provides what is called a boot kernel. The boot kernel is 256 words in length and is prepended to the user code when the loader utility is run (see link port boot example for operation of the loader utility). The purpose of the 256 word loader kernel is to complete the booting process by loading in the user code and data then writing over itself. Each booting mode requires a unique loader kernel, however the purpose of each kernel is the same; load in the user data and code.

The booting kernels provided by Analog devices create a three phase booting process. the first phase is the loading in of the first 256 words as described above. The first 256 words will be the boot kernel. The second phase is the execution of the boot loader kernel which will perform single word DMA transfers through the appropriate port based on the boot mode. In the third and final phase the boot kernel performs a 256 word DMA that will load the code located at addresses 0x20000 through 0x200FF. This results in the loader kernel overwriting itself. The PC will then jump to location 0x20005 and begin executing code. WARNING: a residual instruction will be left at location 0x20004 (please refer to the boot kernel code for explanation).

Link Port Booting Process

The link port booting process has an additional feature unique to the other booting modes. The DMAC6 register is used to configure the external port and DMA channel 6. The link port requires an addition control register, LCTL, in order to configure link port 4 for a DMA transfer. Unlike DMAC6, LCTL is initialized to 0x0000 at power up. LCTL is ignored by the processor during the initial 256 word DMA.

The SHARC ignores the LCTL register setting until some time shortly after the first 256 word DMA is complete. A transition period elapses from the time the SHARC ignores the LCTL setting to the time the SHARC adheres to the LCTL setting. The LCTL is set to 0x0000, therefore link port 4 will be disabled. The disabling of link port 4 discontinues LACK from being driven high. LACK is internally pulled down by a 50 kΩ resistor thus resulting in an RC decay of LACK. The actual decay rate of LACK will depend on the connector type and length connecting the booting device to the link port of the SHARC. The decay time of LACK must be taken into account when transmitting the remaining data to complete the booting process.

Hardware Issues with Link Port Booting

The hardware dependent decay time for LACK must be taken into account when creating the code to boot the SHARC. LACK is used by the link port to hold off a transfer if the link port is not ready to receive data. The transmitting device must sample LACK to determine if the SHARC is ready to receive. If LACK is high the SHARC is

ready to receive, if LACK is low the transfer is held off. (See chapter 9 of the SHARC User's Manual for a detailed explanation of link port operation.) If LACK is sampled before it decays to a 0 logic level the transmitter may attempt to transmit data while the link port is disabled. This **will** result in a loss of data.

The actual RC decay time of LACK is hardware dependent. The decay time will depend on the length of the cable or trace connecting the link port with the transmitting device and the type of material used. Since the decay time will be variable across platforms, a constant time delay to wait for LACK to decay after the first 256 words are transferred is not always possible. The transmitter must dynamically determine what a sufficient decay time is for LACK.

Upon completion of the first 256 word DMA, the transmitter should drive LCLK low and continually sample LACK until it sees LACK is low. Once LACK is low, the transmitter can drive LCLK high and once again sample LACK. When LACK is now high the transmitter knows the receiver is ready to receive and can begin transmitting the remaining data.

A potential race condition is created by the polling of LACK by the transmitter. If LACK does not decay to a logic level of 0 before the receiver enables its link port, LACK will be driven high by the receiver and the transmitter will be stuck in an infinite loop. This is avoided by a polling (handshaking) process within the loader kernel as described below.

The linker loader kernel executed by the booting SHARC enables a handshaking process which can insure the decay of LACK will not result in a loss of data. Link port 4 will be initially disable when the SHARC begins executing the loader kernel. Before the loader kernel enables the link port to receive the remaining code it polls it's LSRQ register. Once it sees a receive data request in the LSRQ register for link port 4, it enables the link port to receive.

An LSRQ link port 4 receive request will occur when the link port 4 LCLK signal is driven high by another device. (See Chapter 9 of the SHARC User's Manual for a detailed description of LSRQ.) In order for the handshaking to work, the transmitter must drive LCLK high after it has sample LACK low.

Link Port Booting Example

The following example demonstrates a link port boot of one SHARC from another SHARC. The link boot loader kernel, the transmitter (master) SHARC code and the secondary (slave) SHARCs link booted routine are included and documented.

The master SHARC will boot the slave SHARC over the link ports by transferring a buffer of data to the booting SHARC. The data will be transferred by the core in two sections. The first section will transfer the first 256 48 bit words to the booting SHARC. After the first transfer the SHARC will disable its link port and wait for LACK to go low. Once LACK goes low the SHARC will re-enable its link port and transmit the remaining 48 bit words.

The code to be booted into the slave SHARC is stored in a buffer named ldata within the master SHARC. This data is stored in 48 bit memory, however, the data is actually 16 bit words. Loader data was created using the following call to the loader:

```
ldr21k lsrq -blink -fascii  -o lsrq.ldr
```

where lsrq is the name of the program to be executed by the slave SHARC, -blink indicates the program will be loaded via a link boot, -fascii indicates the format of the loader file is to be 16 bit ascii and -o forces the name of the output of the loader to be lsrq.ldr.

The 16 bit per word loader data is stored in 48 bit memory. Before transferring the data through the link port to the slave SHARC the data must be packed into 48 bit words. The following code in the master SHARC packs the 16 bit words into 48 bit words and transfers the 48 bit word through link port 4:

```
lcntr = 0x100, do link until lce;
        px1=dm(i0,m0);    /*Read 16 LSBs from
                            memory to PX1*/
        r0=dm(i0,m0);     /*Read middle word*/
        r1=dm(i0,m0);     /*Read 16 MSBs    */
        r0=r0 OR LSHIFT r1 by 16;
        px2=r0;           /*load 32 MSBs in PX*/
link:   pm(LBUF4)=px;     /*Transfer 48 bit word
                            over link port 4*/
```

The least significant bits are store in the lowest location in memory and the most significant 16 bits are stored in the highest location in memory. Three memory accesses are required to retrieve the entire 48 bit word. The PX register is used to pack the 3 16 bit words into one 48 bit word.

The first 256 48 bit words are transferred first. This transfer will load the loader kernel into the booting SHARC. Once the initial 256 words are transferred the transmitting SHARC will disable its link port and then wait for LACK to go low. The following code disables the link port after the link buffer is empty then polls LACK:

```
not_empty: r1=dm(LCOM);   /*wait until lbuf4 is empty */
           btst r1 by 9;
           if not sz jump not_empty;

           bit clr imask  LSRQI;   /*disable LSRQ int*/
           dm(LCTL)=r0;          /* disable lbuf */

           r5=0x1000; dm(LSRQ)=r5;    /*unmask L4TM */
poll:      r4=dm(LSRQ);                /* read LSRQ register */
           btst r4 by 28;        /* check for transmit request */
           if not sz jump poll; /* if not keep polling */

           dm(LSRQ)=r0;      /* mask L4TM */
           dm(LCTL)=r9;      /* enable LCTL again */
```

When LACK is low, the SHARC re-enables its link port and begins to transmit the remaining data to the receiving SHARC.

The loader kernel will now be executing in the receiving SHARC. The reveiving SHARC will poll its LSRQ to determine if the transmitting SHARC has enable its link port to transmit the remaining data. The following code found in the loader kernel is used to poll the LSRQ link port 4 receive request:

```
poll:      R4 =DM(LSRQ);    /* see if there is a */
           btst r4 by 29;   /* receive request */
           if sz jump poll; /* if not keep polling */
dm(LCTL) = R7;              /*Lbuf4 enabled,
                            recieve, 48bits */
```

Bit 29 (L4RRQ) in the LSRQ register will be set when the LCLK signal of the receiving SHARC is driven high by the transmitter. This indicates the transmitter is ready to send data. In our example this means the master SHARC has enabled its link port and is attempting to transmit the remaining data.

When L4RRQ of the LSRQ register is set, the kernel will enable link port 4 and configure it for receiving 48 bit words. The loader kernel will load the remaining data, store all data and code in the proper locations, write over itself then the SHARC will begin executing the program booted into the SHARC.

The code for the link port loader kernel, the transmitting SHARC and the booting SHARC are listed below.

Link Port Booting Loader Kernel

```
/*************************************************************
***************
060_link.asm
Link based boot loader of the ADSP21060 processor.
Copyright (c) 1997 Analog Devices Inc.  All rights reserved.
*************************************************************
***************/
/*  As of 8/11/97 this kernel works with blsrq3.asm   */
/* Warning: After booting process, EP0I bit in IMASK is set. */
/* Define the addresses of various IOP registers*/
#define SYSCON 0x00
#define WAIT   0x02
#define II6    0x40
#define IM6    0x41
#define C6     0x42
#define CP6    0x43
#define EC6    0x47
#define DMAC6  0x1c
#define LCOM       0xc7
#define LCTL       0xc6
#define LSRQ       0xc9
#define LBUF4      0xc4

.SEGMENT/PM     seg_ldr;

/* The loader begins with interrupts up to and including    */
/* the low priority timer interrupt.                        */

NOP;NOP;NOP;NOP;          /* Reserved interrupt*/

___lib_RSTI:        IDLE;     /* Implicit IDLE instruction */
                    JUMP Start_Loader (DB);/*Begin loader
          */
                    NOP;      /* Pad to next interrupt       */
                    NOP;      /* Pad to next interrupt       */

                    NOP;NOP;NOP;NOP;/* Reserved       */
/* Vector for status /loop stack overflow or PC stack full:  */
___lib_SOVFI:       RTI; RTI; RTI; RTI;
/* Vector for high priority timer interrupt:                 */
___lib_TMZHI:       RTI; RTI; RTI; RTI;
/* Vectors for external interrupts:                          */
___lib_VIRPTI:      RTI; RTI; RTI; RTI;
___lib_IRQ2I:       RTI; RTI; RTI; RTI;
___lib_IRQ1I:       RTI; RTI; RTI; RTI;
___lib_IRQ0I:       RTI; RTI; RTI; RTI;
                    NOP;NOP;NOP;NOP;/* Reserved       */
/* Vectors for Serial port DMA channels:                     */
___lib_SPR0I:       RTI; RTI; RTI; RTI;
___lib_SPR1I:       RTI; RTI; RTI; RTI;
___lib_SPT0I:       RTI; RTI; RTI; RTI;
___lib_SPT1I:       RTI; RTI; RTI; RTI;
/* Vectors for link port DMA channels:
          */
___lib_LP2I:        RTI; RTI; RTI; RTI;
___lib_LP3I:        RTI; RTI; RTI; RTI;
/* Vectors for External port DMA channels:                   */
___lib_EP0I:        RTI (DB);
```

```
                    nop;
                    bit clr imask 0x10000;
                    RTI;
Start_Loader:
/* After power up or reset, default value in SYSCON is 0x0000
0010 and in WAIT is 0x21AD6B5A. If for any reason these two
value should be  modified, do it here as following example.
Make sure this file does not exceed 256 words.    */
/*   DM(SYSCON)=0Xxxxxxxxx;    Modify SYSCON        */
/*   DM(WAIT)=0Xxxxxxxxx;      Modify WAIT
          */

                    L0=0;     /* Zero out L-registers so they*/
                    L4=0;     /*  can be used without wrap */
                    L8=0;
                    L12=0;
                    L15=0;
                    M5=0;     /* Setup M-registers to use  */
                    M6=1;     /*  for various memory transfers*/
                    M13=0;
                    M14=1;


          R11=DM(SYSCON);     /* Read current SYSCON   */
          R12=PASS R11;       /* Hold Initial SYSCON   */
          I12 = SYSCON;       /* Address of SYSCON     */
          I15 = 0x20004;
          R8 = 0x0;           /* 1x ( note: 0x10000 would be for
                                 lbuf4=2X) */
          dm(LCOM)=R8;
          R8 = 0x0;           dm(LSRQ)=R8;
          R8 = 0x2000;        dm(LSRQ) = R8;

          R8 = 0x10030000;  /* Lbuf4 and DMA enabled,
                               receive,   48bits */
          R7 = 0x10010000;    /* Lbuf4 enabled , recieve,
48bits */
          poll:     R4 =DM(LSRQ);    /* see if there is a */
                    btst r4 by 29;   /* receive request */
                    if sz jump poll; /* if not keep polling */
          dm(LCTL) = R7;             /*Lbuf4 enabled,
                                       recieve, 48bits */

read_boot_info:     CALL read_LINK_word;
                    R0=PASS R2;
                    CALL read_LINK_word;

load_memory:        R0=PASS R0;
                    IF NE JUMP (PC, test_dm16_zero);

/* After the IDLE completes, the following sequence of
instructions will be executed: (Remember these are in a loop)
          R0=R0-R0, DM(I4,M5)=R9, PM(I12,M13)=R11
This instructions sets the EQ flag to terminate the loop, writes
the original value to SYSCON, and writes a *new* instruction
overitself.  The new instruction is:
          PM(0, I8)=PX;
This instruction resets the DMA6 vector to whatever it should
be.
The loop will terminate, because of the previous set EQ.
Instructio flow will continue with 0x20005, just like nothing
happened!                                                  */

final_init: R9=0xb1db0000;          /* Load instruction
                                                  PM(0,I8)=PX;
*/
          R11=BSET R11 BY 8;        /* Set IMDW to 1 for
                                       inst write  */
          DM(SYSCON)=R11;           /* Setup to read
                                       PROM        */

          R1=0x020000;        /* Point to destination       */
          R2=0x100;                 /* Load length of last
init     */
          DM(IM6)=M14;      /* Set to increment internal ptr
*/

          I4=0x020004;        /* Point to 0x020004 for patch
*/
          I8=0x020040;        /* Point  DMA6 vector to patch
*/
          R4=PASS R4, R11=R12;   /* Clear AZ, hold initial
                                                  SYSCON */
          DO ___lib_RSTI UNTIL EQ;/* Setup dummy loop  */
          Flush Cache;
          R0=0x20004;         /* replace with new       */
          PCSTK=R0;           /*  top-of-loop value         */
          R0=0x0;
          DM(II6)=R1;         /* Setup DMA to load over ldr */
          DM(C6)=R2;          /* Load internal count        */
          DM(CP6)=R0;         /* Set CP to 0                */
          DM(LCTL)=R8;        /* Start DMA transfer         */
          bit clr irptl 0x10000;  /*clear pending EP0 interrupts
*/
          bit set imask 0x10000;  /* enable  EP0 interrupts
*/
          JUMP 0x20004 (DB);      /* Jump to start
          */
          bit set mode1 0x1800;   /* allow global interrupts */
          IDLE;               /* After IDLE, patch then start*/

test_dm16_zero:   R0=R0-1;
                  IF EQ JUMP (PC, dm16_zero);
                  R0=R0-1;
                  IF NE JUMP (PC, test_dm40_zero);

dm32_zero:
dm16_zero:        R0=R0-R0, I0=R3;
                  LCNTR=R2, DO dm16_zero_loop UNTIL
LCE;
dm16_zero_loop:   DM(I0,M6)=R0;
                  JUMP read_boot_info;

test_dm40_zero:   R0=R0-1;
                  IF NE JUMP (PC, test_dm16_init);

dm40_zero:        PX1=0;
                  PX2=0;
                  R0=R0-R0, I0=R3;
                  LCNTR=R2, DO dm40_zero_loop UNTIL
LCE;

dm40_zero_loop:   DM(I0,M6)=PX;
                  JUMP read_boot_info;

test_dm16_init:   R0=R0-1;
                  IF NE JUMP (PC, test_dm32_init);

dm16_init:        I0=R3;
                  LCNTR=R2, DO dm16_init_loop UNTIL
LCE;
                  CALL read_LINK_word (DB);
```

```
                NOP;            /* NOP's is required by */
                NOP;            /* loop restriction     */

dm16_init_loop: DM(I0,M6)=R3;
                JUMP read_boot_info;

test_dm32_init: R0=R0-1;
                IF NE JUMP (PC, test_dm40_init);

dm32_init:      I0=R3;
                LCNTR=R2, DO dm32_init_loop UNTIL
LCE;
                CALL read_LINK_word (DB);
                NOP;
                NOP;

dm32_init_loop: DM(I0,M6)=R3;
                JUMP read_boot_info;

test_dm40_init: R0=R0-1;
                IF NE JUMP (PC, test_pm16_zero);

dm40_init:      I0=R3;
                LCNTR=R2, DO dm40_init_loop UNTIL
LCE;
                CALL read_LINK_word (DB);
                NOP;
                NOP;

dm40_init_loop: DM(I0,M6)=PX;
                JUMP read_boot_info;

test_pm16_zero: R0=R0-1;
                IF EQ JUMP (PC, dm16_zero);
                R0=R0-1;
                IF EQ JUMP (PC, dm32_zero);

test_pm40_zero: R0=R0-1;
                IF EQ JUMP (PC, dm40_zero);
                R0=R0-1;
                IF NE JUMP (PC, test_pm16_init);

pm48_zero:      PX1=0;
                PX2=0;
                R0=R0-R0, I8=R3;
               LCNTR=R2, DO pm40_zero_loop UNTIL
LCE;
pm40_zero_loop: PM(I8,M14)=PX;
                JUMP read_boot_info;

test_pm16_init: R0=R0-1;
                IF EQ JUMP (PC, dm16_init);

test_pm32_init: R0=R0-1;
                IF EQ JUMP (PC, dm32_init);

test_pm40_init: R0=R0-1;
                IF EQ JUMP (PC, dm40_init);

test_pm48_init: R0=R0-1;
                IF NE JUMP read_boot_info;

pm48_init:      I8=R3;
                LCNTR=R2, DO pm48_init_loop UNTIL
LCE;
                CALL read_LINK_word (DB);
                NOP;
                NOP;

pm48_init_loop: PM(I8,M14)=PX;
                JUMP read_boot_info;

read_LINK_word:
                PX=PM(LBUF4);/* Read word          */
                RTS (DB);
                R2=PX1; /* Copy PX values into DREG
          */
                R2=PASS R2, R3=PX2;/* Test  length  */
.ENDSEG;
```

Transmitting SHARC Code

```
/*LOADER EXAMPLE - TRANSMITTER (MASTER)*/
/*  NOTE: 1 thing needs to be changed each time you change the file
that you want to load.  The #define size should be changed to match the
number of lines in the *.ldr file that you are sending over the links.
In this case lsrq.ldr.    */
/*This file is intended for Rev 1.0 and later   ONLY                  */

#include "def21060.h"

/* CHANGE SIZE ONLY = # of lines in the  .ldr file that you include as
a variable in this program */

#define size 1542
#define loopsize ((size/3)-256)   /* automatically calculates, how many
                                     words to send after first 256 words */
.SEGMENT/PM     irqvects;

/* The loader begins with the interrupts up to and including the low
priority timer interrupt.                                          */
                NOP;NOP;NOP;NOP;        /* Reserved        */
___lib_RSTI:    IDLE;                   /* Implicit IDLE instruction       */
                JUMP start (DB); /* Begin loader           */
                NOP;                    /* Pad to next interrupt   */
                NOP;                    /* Pad to next interrupt   */
                NOP;NOP;NOP;NOP;        /* Reserved        */
/* Vector for status stack/loop stack overflow or PC stack full:      */
___lib_SOVFI:   RTI; RTI; RTI; RTI;
/* Vector for high priority timer interrupt:                          */
___lib_TMZHI:   RTI; RTI; RTI; RTI;
/* Vectors for external interrupts:                                   */
___lib_VIRPTI:  RTI; RTI; RTI; RTI;
___lib_IRQ2I:   RTI; RTI; RTI; RTI;
___lib_IRQ1I:   RTI; RTI; RTI; RTI;
___lib_IRQ0I:   RTI; RTI; RTI; RTI;
                NOP;NOP;NOP;NOP;  /* Reserved              */
/* Vectors for Serial port DMA channels:                              */
___lib_SPR0I:   RTI; RTI; RTI; RTI;
___lib_SPR1I:   RTI; RTI; RTI; RTI;
___lib_SPT0I:   RTI; RTI; RTI; RTI;
___lib_SPT1I:   RTI; RTI; RTI; RTI;
/* Vectors for link port DMA channels:                                */
___lib_LP2I:    RTI; RTI; RTI; RTI;
___lib_LP3I:    RTI; RTI; RTI; RTI;
/* Vectors for External port DMA channels:                            */
___lib_EP0I:    RTI; RTI; RTI; RTI;
___lib_EP1I:    RTI; RTI; RTI; RTI;
___lib_EP2I:    RTI; RTI; RTI; RTI;
___lib_EP3I:    RTI; RTI; RTI; RTI;
/* Vector for Link service request                                    */
___lib_LSRQ:            call lsrqf (db);
```

```
                    r4=dm(LSRQ);
                    dm(LSRQ)=r6;                    /*turn off*/
                    rti;
/* Vector for DAG1 buffer 7 circular buffer overflow                */
___lib_CB7I:    RTI; RTI; RTI; RTI;
/* Vector for DAG2 buffer 15 circular buffer overflow               */
___lib_CB15I:   RTI; RTI; RTI; RTI;
* Vector for lower priority timer interrupt                        */
___lib_TMZLI:   RTI; RTI; RTI; RTI;

.ENDSEG;

/*  The size of ldata should be changed to match file to send */
.SEGMENT/pm     pm48_2b0;
.VAR    ldata[size] = "lsrq.ldr";
.ENDSEG;

.SEGMENT/pm     pm48_1b0;
start:
                    ustat1=dm(SYSCON);
                    bit set ustat1 IMDW0X | IMDW1X;
                    dm(SYSCON)=ustat1;
                    bit set mode2 FLG0O;/* set FLG0 as an output */
                    call lload;
                    r2=0x11111111;

          test:     r0=dm(LCOM); /* When lbuf4 empty, disable  */
                    btst r0 by 9;
                    if not sz jump test;

                    r8= 0x0;
                    dm(LCTL)=r8;          /* disable lbuf */

                    r5=0x1000; dm(LSRQ)=r5; /* unmask L4TM */
                    r6=0x0; /* for later masking */

                    bit set imask LSRQI; /* enable LSRQ and ints*/
                    bit set mode1 IRPTEN;

                    idle;
                    jump (pc,-1);

/* ----------------------link boot routine here---------------------------------*/
lload:              r0 = 0x0;   /* 1x; 0x10000 = 2x for lbuf4 */
                    dm(LCOM)=r0;
                    /*LBUFx<=>LPORTx */
                    r0=0x2c688; dm(LAR)=r0;

                    r9=0x10090000;dm(LCTL)=r9;  /*trans,lbuf en*/
                    b0=ldata; l0=0;  m0=1;

                    /* ----------------------PART 1 ----------------------*/
                    /*  Send first 256 words (link boot routine) */
                    lcntr = 0x100, do link until lce;
                    px1=dm(i0,m0);
                    r0=dm(i0,m0);
                    r1=dm(i0,m0);
                    r0=r0 OR LSHIFT r1 by 16;
                    px2=r0;
          link:     pm(LBUF4)=px;

                    /*----------------------PART 2 ----------------------*/
                    /* Wait until chip is ready for rest  of program */

                    /* (wait for booting chip to try to read more
          data,you don't want to send data right away
          because it will be flushed when the booting chip
          finishes booting its first 256 words and begins
          running its loader routine)  */
                    r0=0x0;
      not_empty: r1=dm(LCOM);/* wait until lbuf4 is empty */
                    btst r1 by 9;
                    if not sz jump not_empty;

                    bit clr imask  LSRQI|EP0I;/* disable LSRQ */
                    dm(LCTL)=r0;          /* disable lbuf */

                    r5=0x1000; dm(LSRQ)=r5;/* unmask L4TM */

          poll:     r4=dm(LSRQ); /* read LSRQ register */
                    btst r4 by 28;  /* check for a transmit request */
                    if not sz jump poll;     /* if not keep polling */

                    dm(LSRQ)=r0;          /* mask L4TM */
                    dm(LCTL)=r9;          /* enable LCTL again */

                    /*----------------------PART 3 ----------------------*/
                    /* Send rest of the program */

                    lcntr = loopsize, do link2 until lce;
                    px1=dm(i0,m0);
                    r0=dm(i0,m0);
                    r1=dm(i0,m0);
                    r0=r0 OR LSHIFT r1 by 16;
                    px2=r0;
          link2:    pm(LBUF4)=px;
                    nop;
                    rts (db);     /* return to main program */
                    nop;
                    nop;

/* ------------------------------end of link boot routine -------------------------*/
lsrqf:              btst r4 by 28;
                    if sz jump no_blink;

                    /* if L4TM, tgl LED and write word out */
                    bit tgl astat FLG0;
                    dm(LCTL)=r9;          /* enable, transmit */
                    dm(LBUF4)=r2;         /* service */

          test2:    r0=dm(LCOM);
                    btst r0 by 9;
                    if not sz jump test2;

                    dm(LCTL)=r8;          /* disable */
          no_blink: dm(LSRQ)=r5;          /* turn back on */
                    rts;
.ENDSEG;
```

Code Executed in SHARC After Booting Process

```
/* LINK BOOTED ROUTINE(secondary processor)  */
/*This file uses a timer to cause a LSRQ interupt to occur on
another processor through lport4.  This file is intended for Rev
1.0 and later                                                    */

#include "def21060.h"

.SEGMENT/PM     irqvects;
/* The loader begins with the interrupts */
                    NOP;NOP;NOP;NOP;       /* Reserved     */
___lib_RSTI:        IDLE; /* Implicit IDLE instruction    */
                    JUMP start (DB); /* Begin loader
*/
```

```
                        nop;                /* Pad to next interrupt
*/
                        NOP;                /* Pad to next interrupt
*/
                        NOP;NOP;NOP;NOP;        /* Reserved      */

/* Vector for status /loop stack overflow or PC stack full:       */
___lib_SOVFI:   RTI; RTI; RTI; RTI;
/* Vector for high priority timer interrupt:                      */
___lib_TMZHI:       r0=dm(LBUF4);
                        RTI; RTI; RTI;
/* Vectors for external interrupts:                               */
___lib_VIRPTI:      RTI; RTI; RTI; RTI;
___lib_IRQ2I:       RTI; RTI; RTI; RTI;
___lib_IRQ1I:       RTI; RTI; RTI; RTI;
___lib_IRQ0I:       RTI; RTI; RTI; RTI;
                        NOP;NOP;NOP;NOP;  /* Reserved      */
/* Vectors for Serial port DMA channels:
            */
___lib_SPR0I:       RTI; RTI; RTI; RTI;
___lib_SPR1I:       RTI; RTI; RTI; RTI;
___lib_SPT0I:       RTI; RTI; RTI; RTI;
___lib_SPT1I:       RTI; RTI; RTI; RTI;
/* Vectors for link port DMA channels:
            */
___lib_LP2I:        RTI; RTI; RTI; RTI;
___lib_LP3I:        RTI; RTI; RTI; RTI;
/* Vectors for External port DMA channels:                        */
___lib_EP0I:        RTI; RTI; RTI; RTI;
___lib_EP1I:        RTI; RTI; RTI; RTI;
___lib_EP2I:        RTI; RTI; RTI; RTI;
___lib_EP3I:        RTI; RTI; RTI; RTI;
/* Vector for Link service request                                */
___lib_LSRQ:        RTI; RTI; RTI; RTI;
/* Vector for DAG1 buffer 7 circular buffer overflow              */
___lib_CB7I:        RTI; RTI; RTI; RTI;
/* Vector for DAG2 buffer 15 circular buffer overflow             */
___lib_CB15I:       RTI; RTI; RTI; RTI;
/* Vector for lower priority timer interrupt                      */
___lib_TMZLI:       RTI; RTI; RTI; RTI;
.ENDSEG;

.SEGMENT/pm     pm48_1b0;
start:              r1=0x2c688; /* map lbufx <--> lportx */
                        dm(LAR)=r1;

                        /* 1x on lbuf4 */
                        r1=0x0;   /* note: 0x10000=2x on lbuf4 */;
                        dm(LCOM)=r1;

                        /* enable lbuf4,enabled, receiver, 48bits */
                        r1=0x10010000; dm(LCTL)=r1;

                        tperiod = 0x1fc1e20;
                        tcount = tperiod;
                        bit set imask TMZHI;
                        bit set mode2 TIMEN;
                        bit set mode1 IRPTEN;

                        idle;
                        jump (pc,-1);
.ENDSEG;
```