Technical notes on using Analog Devices products and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

MDMA-Based Dual-SHARC+ Parallel Pipeline Audio Talkthrough

Contributed by Eric Gregori

Rev 1 – December 17, 2015

Introduction

The efficiency of a multi-core design is dependent on the quality of the Inter-Core Communication (ICC) system, as moving data between cores efficiently defines the overall system performance. This EE-note describes a method of Inter-Core Communication made possible by the unique DMA subsystem in the ADSP-SC58x heterogeneous multi-core SHARC+^{TM} processor (Figure 1).

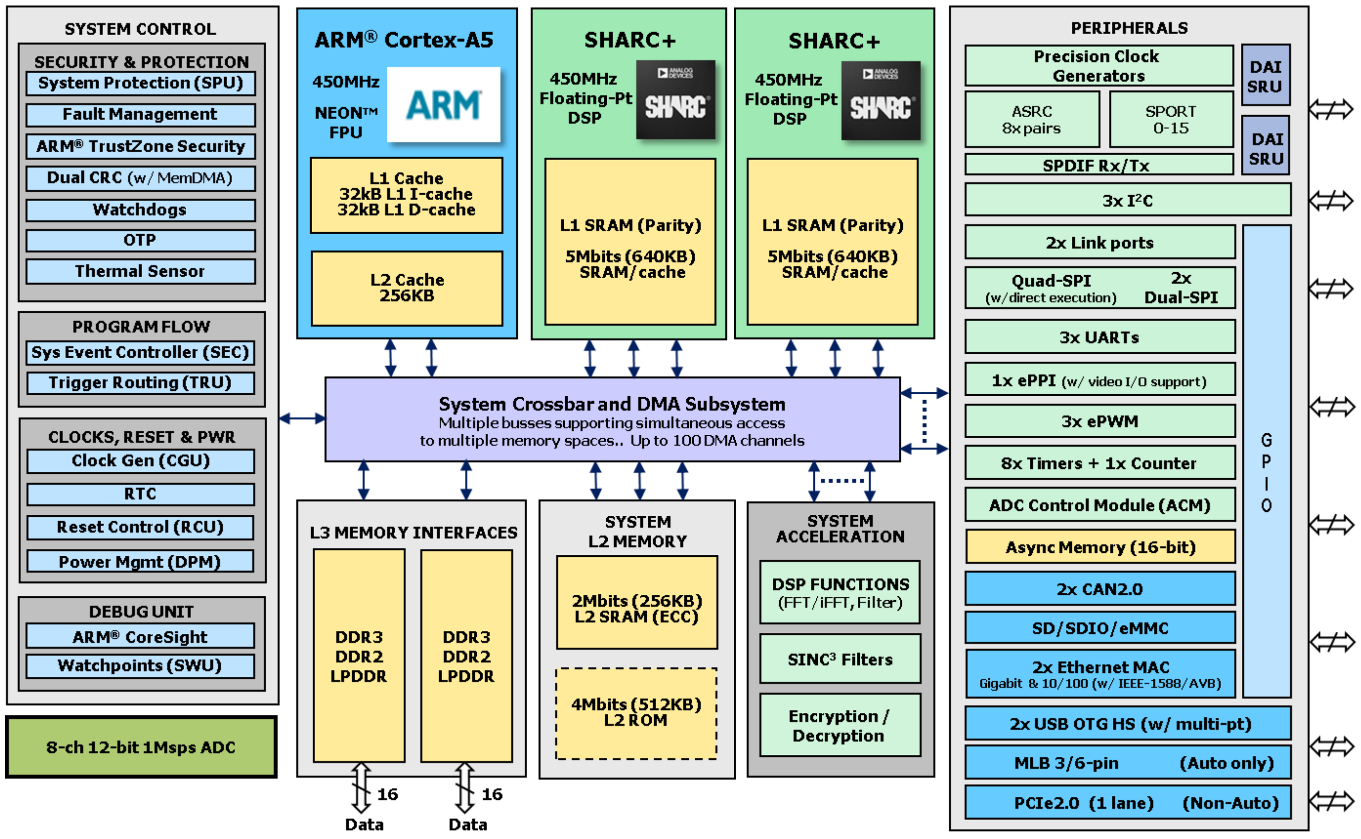

Figure 1. ADSP-SC58x SHARC+ Processor Block Diagram

The ADSP-SC58x processor contains one or two SHARC+ cores, SHARC0 (and optionally SHARC1), each with 640 KB of L1 SRAM (S0L1 and S1L1), an ARM Cortex-A5 processor, 256 KB of L2 SRAM, and a powerful Direct Memory Access (DMA) subsystem supporting Memory-to-Memory DMA (MDMA) operation with signaling.

This EE-note describes a dual-SHARC+ core quad-channel audio talkthrough example (in the *Associated ZIP File*[1]) which takes quad-channel audio from an ADC and filters two channels on each of the SHARC+ cores. The filtering is done in parallel to demonstrate the advantage of a multi-core system. After each SHARC+ core filters its channels, the filtered results are sent to a DAC for audio playback. Though the example is a simple audio talkthrough application, hooks are in place for readers to add their own audio processing algorithms.

Memory-to-Memory DMA (MDMA)

Each SHARC+ core in the ADSP-SC58x processor has 640 KB of single-cycle L1 memory, and the core executes at maximum efficiency when accessing data in its L1 memory. Distributing an algorithm across cores depends on an efficient method of copying data from the SHARC0 core's L1 space (S0L1) to the SHARC1 core's L1 space (S1L1). Using MDMA, data can be moved between these two spaces without core intervention (entirely in the background) at up to 1500 MB/s. The ADSP-SC58x processor contains four independent MDMA streams, coupled into source/destination (S/D) DMA channels, as shown in Figure 2.

MDMA Stream	Channel (S/D)	FIFO Depth	Speed	Performance S0L1 to S1L1
0	**8, 9**	**128, 64**	**Low**	**450 MB/s**
1	**18, 19**	**128, 64**	**Low**	**450 MB/s**
2	**39, 40**	**128, 64**	**Medium**	**900 MB/s**
3	**43, 44**	**128, 64**	**High**	**1500 MB/s**

Figure 2. ADSP-SC58x Processor Memory-to-Memory DMA Channel Description

In addition to copying data, an ICC system requires a signaling method. A unique feature of the ADSP-SC58x processor's MDMA engine is the ability to generate an interrupt on the processor receiving the data transfer. The SHARC0 core can initiate a MDMA transfer that raises an interrupt on the SHARC1 core. The interrupt acts as a signal between cores that a transfer has completed. As shown in Figure 3, the MDMA stream is comprised of two DMA channels, a source channel (SRC) and a destination channel (DST), each with a unique interrupt source ID (SID), and each core can request to be interrupted by any SID. In the example, the MDMA0 stream is configured to raise an interrupt on the SHARC0 core when the source DMA channel has completed the transfer from S0L1 to the FIFO, and another interrupt is raised on the SHARC1 core when the destination DMA channel has completed the transfer from the FIFO to the S1L1 space. The code executing on the SHARC1 core uses the interrupt for synchronization between cores.

The example uses two MDMA streams, each with a unique interrupt handler on the SHARC1 core. The MDMA0 stream is used to move raw audio from S0L1 to S1L1, and the MDMA1 stream is used to move filtered audio from S0L1 to S1L1.

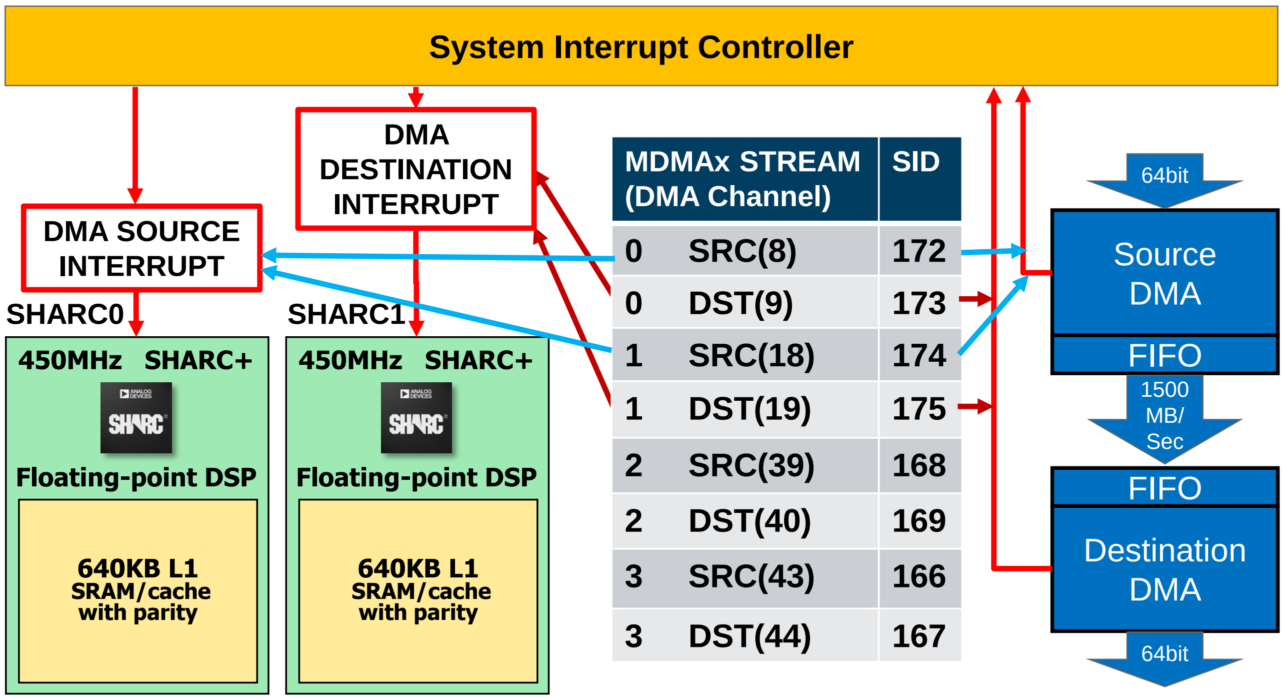

Figure 3. ICC MDMA Mapping and Signaling

Each SHARC+ core's L1 memory is broken up into four blocks, each having multiple private and public addresses depending on the access width (long word, normal word, short word, or byte). Figure 4 shows the local and global address ranges for byte accesses.

	Port	Block	Local Byte Address Space	Global Byte Address Space
SHARC0 L1	Slave 1	0	0x00240000 – 0x0026FFFF	0x28240000–0x2826FFFF
		1	0x002C0000 – 0x002EFFFF	0x282C0000–0x282EFFFF
		2	0x00300000 – 0x0031FFFF	0x28300000–0x2831FFFF
		3	0x00380000 – 0x0039FFFF	0x28380000–0x2839FFFF
	Slave 2	0	0x00240000 – 0x0026FFFF	0x28640000–0x2866FFFF
		1	0x002C0000 – 0x002EFFFF	0x286C0000–0x286EFFFF
		2	0x00300000 – 0x0031FFFF	0x28700000–0x2871FFFF
		3	0x00380000 – 0x0039FFFF	0x28780000–0x2879FFFF
SHARC1 L1	Slave 1	0	0x00240000 – 0x0026FFFF	0x28A40000–0x28A6FFFF
		1	0x002C0000 – 0x002EFFFF	0x28AC0000–0x28AEFFFF
		2	0x00300000 – 0x0031FFFF	0x28B00000–0x28B1FFFF
		3	0x00380000 – 0x0039FFFF	0x28B80000–0x28B9FFFF
	Slave 2	0	0x00240000 – 0x0026FFFF	0x28E40000–0x28E6FFFF
		1	0x002C0000 – 0x002EFFFF	0x28EC0000–0x28EEFFFF
		2	0x00300000 – 0x0031FFFF	0x28F00000–0x28F1FFFF
		3	0x00380000 – 0x0039FFFF	0x28F80000–0x28FBFFFF

Figure 4. ADSP-SC58x SHARC+ Processor L1 Memory Address Ranges for Byte Accesses

Each SHARC+ core uses the local address range to access its own L1 memory, while the Cortex-A5 ARM core and the DMA engine use the global address range. In the example, MDMA is used to transfer data from S0L1 to S1L1 block1. The SHARC0 core configures the MDMA destination address to be 0x28AC0000. After the transfer completes, the SHARC1 core can read the transferred data at address 0x002C0000.

Walkthrough of Audio Talkthrough Example

A block diagram of the audio talkthrough example is shown in Figure 5.

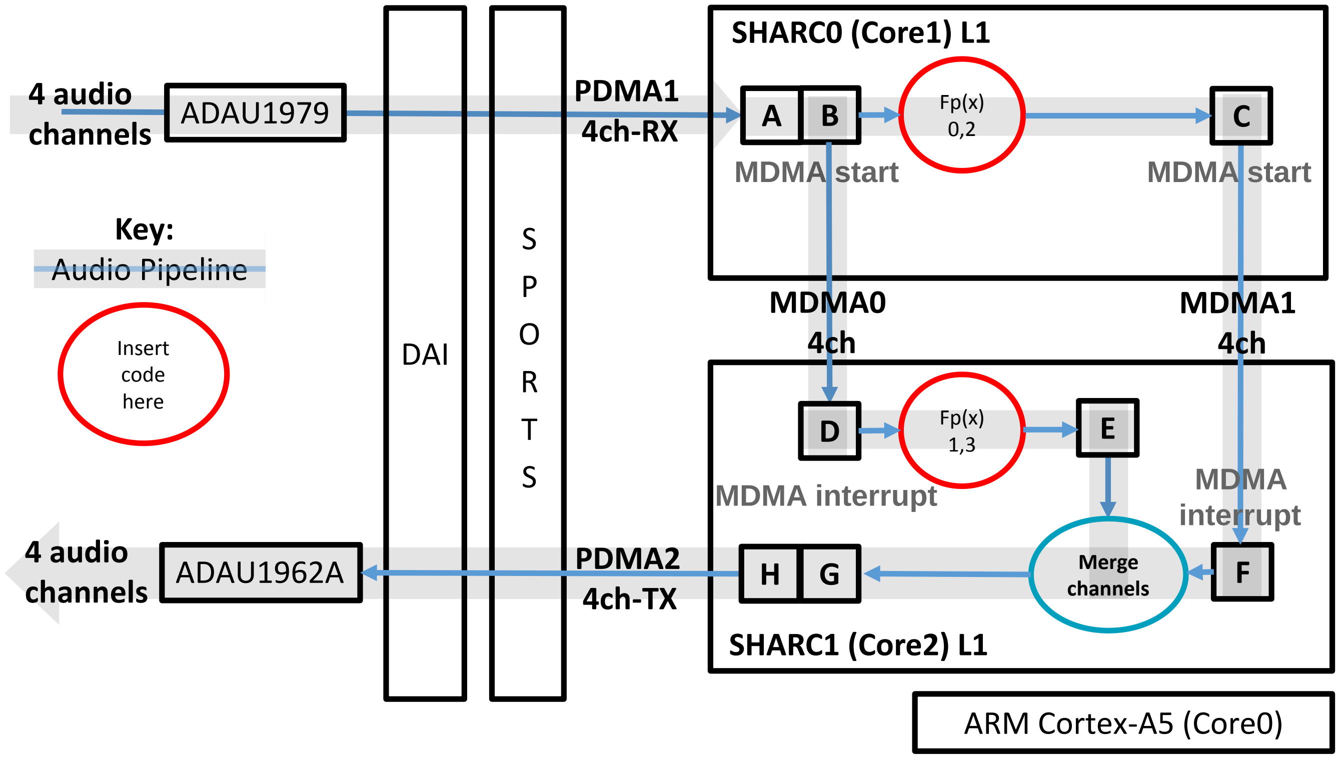

Figure 5. Block Diagram of Audio Talkthrough Example

In the CrossCore® Embedded Studio (CCES) Integrated Development Environment (IDE), the ARM Cortex-A5 core is Core0 on the ADSP-SC58x processor. The SHARC+ cores are Core1 and Core2; however, as the System Event Controller (SEC) does not handle ARM events, the SHARC+ cores themselves are enumerated as SHARC0 (SHARC+ core 0, which is Core1 in the IDE) and SHARC1 (The SHARC0 core, which is Core2 in the IDE).

Starting from the upper left of Figure 5, four audio channels enter the ADAU1979 ADC and are digitized into a serial stream. The serial stream enters the ADSP-SC58x processor via the Digital Audio Interface (DAI) and is transferred to S0L1 via the synchronous serial port (SPORT) receive (RX) Peripheral DMA (PDMA) channel. The SHARC0 core receives an interrupt after the audio frame is received into the SPORT RX PDMA destination buffer in S0L1 (ping-pong buffers A or B). The interrupt service routine (ISR) then executes, where the SHARC0 core starts the MDMA0 stream to transfer the raw audio from the source buffer (ping-pong buffer A or B) in S0L1 to the destination buffer D in S1L1. At that point, the two

SHARC+ cores start filtering the audio frame in parallel, with the SHARC0 core handling channels 0 and 2, storing the results to output buffer C in S0L1, while the SHARC1 core handles channels 1 and 3, storing the results to output buffer E in S1L1.

After the SHARC0 core fills buffer C, it starts the MDMA1 stream to transfer the filtered audio from source buffer C in S0L1 to destination buffer F in S1L1. Upon completion, the SHARC1 core gets an interrupt and interleaves the data from channels 0 and 2 from buffer F (from the SHARC0 core's processing) with the data from channels 1 and 3 in buffer E (from its own processing), storing the merged results to the SPORT transmit (TX) DMA ping-pong buffer (G or H). At this point, the 4-channel audio frame is sent via SPORT TX PDMA through the DAI to the ADAU1962A DAC to be output.

Audio Data Path vs Time

Figure 6 shows the path of an audio frame through the system relative to time. Notice how the SHARC0 and SHARC1 cores run their signal processing algorithms in parallel. The MDMA0 stream transfer occurs while the SHARC1 core is merging the audio channels from the previous frame. Also note that the MDMA0 stream's interrupt is used to synchronize the ADAU1979 ADC and the ADAU1962 DAC.

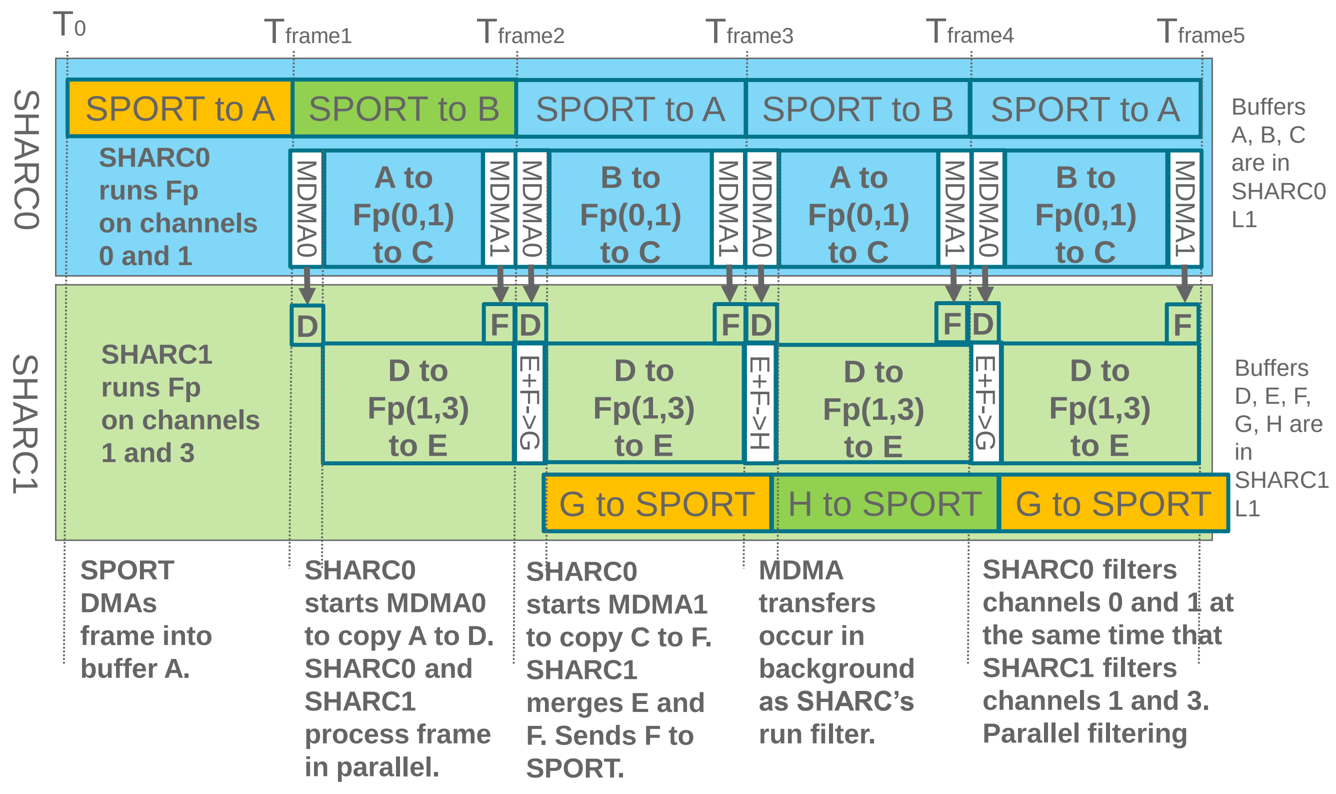
T0
Tframe1
Tframe2
Tframe3
Tframe4
Tframe5
SHARC0
SPORT to A
SPORT to B
SPORT to A
SPORT to B
SPORT to A
SHARC0 runs Fp on channels 0 and 1
MDMA0
A to Fp(0,1) to C
MDMA1
MDMA0
B to Fp(0,1) to C
MDMA1
MDMA0
A to Fp(0,1) to C
MDMA1
MDMA0
B to Fp(0,1) to C
MDMA1
Buffers A, B, C are in SHARC0 L1
SHARC1
SHARC1 runs Fp on channels 1 and 3
D
F D
F D
F D
F
D to Fp(1,3) to E
E+F->G
D to Fp(1,3) to E
E+F->H
D to Fp(1,3) to E
E+F->G
D to Fp(1,3) to E
G to SPORT
H to SPORT
G to SPORT
Buffers D, E, F, G, H are in SHARC1 L1
SPORT DMAs frame into buffer A.
SHARC0 starts MDMA0 to copy A to D. SHARC0 and SHARC1 process frame in parallel.
SHARC0 starts MDMA1 to copy C to F. SHARC1 merges E and F. Sends F to SPORT.
MDMA transfers occur in background as SHARC's run filter.
SHARC0 filters channels 0 and 1 at the same time that SHARC1 filters channels 1 and 3. Parallel filtering

Figure 6. Audio Frame Data Path Relative to Time

Initializing MDMA for ICC

As MDMA configuration is only done during initialization, the penalty for accessing L2 has no effect on algorithm run-time. Configuring the MDMA for ICC requires a specific sequence of events. Sequencing these events between cores requires shared memory, and the example uses a 32-bit variable stored in L2 to facilitate this.

Figure 7 shows the events required to initialize the MDMA for ICC. T0 through T6 comprise the initialization sequence. The SHARC0 core is the master, and the SHARC1 core is the slave. Out of reset, all three cores start executing. The SHARC+ cores' pre-initialization code pends on a signal from the ARM. The ARM initializes power, general-purpose input/output (GPIO), and the System protection Unit (SPU) before sending a signal to the SHARC+ cores to release and start running their `main()` functions (T0).

Time	ARM Cortex-A5 (Core0)	SHARC0 (SHARC+ Core1)	SHARC1 (SHARC+ Core2)
T0	DAI, Power, GPIO, SPU init	Waiting	Waiting
T1	Waiting	Initialize ADAU1979	Initialize ADAU1962A
T2	Waiting	MDMA Init	Waiting
T3	Waiting	Waiting	Install Interrupt Handlers
T4	Waiting	ADAU1979 Enable	Waiting
T5	Waiting	Frame from ADAU1979 Send frame using MDMA0	Waiting
T6	Waiting	Filter channels 0,2	MDMA0 Interrupt – Enable ADAU1962A
T(n)	Waiting	Filter channels 0,2	Filter channels 1,3
T(n+1)	Waiting	Send filtered audio frame using MDMA1	Filter channels 1,3
T(n+2)	Waiting	Frame from ADAU1979 Send frame using MDMA0	MDMA1 Interrupt Merge channels 0,1,2,3 Send frame to ADAU1962
T(n+3)	Waiting	Filter channels 0,2	MDMA0 Interrupt

Figure 7. Processor Initialization Sequence at Startup

After the SHARC0 core initializes the ADAU1979 ADC (T1), it configures the MDMA channels for ICC operation (T2). Meanwhile, the SHARC1 core initializes the ADAU1962A DAC (T1), then waits for the SHARC0 core to complete the MDMA initialization (T2). When the SHARC0 core completes the MDMA initialization, it signals the SHARC1 core using the 32-bit variable in L2 and waits for an acknowledge (T3). After getting the signal, the SHARC1 core installs the interrupt handlers for the MDMA channels and sends a signal back to the SHARC0 core using the 32-bit L2 variable (T3). The SHARC1 core then waits for an audio frame from the SHARC0 core (T4-T5). Upon receiving the signal from the SHARC1 core, the SHARC0 core enables the ADAU1979 ADC (T4) and waits for an audio frame to arrive via the SPORT (T5). At this point during the initialization, the SHARC1 core is waiting for a raw audio frame (MDMA0) from the SHARC0 core (T5). When the SHARC0 core gets the audio frame from the ADAU1979 ADC, it immediately sends the frame to the SHARC1 core using MDMA0 (T5) and starts filtering channels 0 and 2 (T6). When the SHARC1 core gets the first MDMA0 transfer complete interrupt, it enables the ADAU1962A DAC (T6). From that point on, the application continues receiving audio frames and processing them using this ICC implementation, as shown in T(n) through T(n+3).

Configuring the MDMA engine for ICC is easy using the MDMA device driver furnished with CCES. Listing 1 is a code snippet from the accompanying example code that shows how to use the device driver to open a MDMA stream and configure it for ICC. MDMA_STREAM_ID_RAW is the DMA stream ID. The driver translates the stream ID to the required pair of source/destination DMA channel IDs. As the raw data is sent using the MDMA0, the source DMA channel 8 (SID 172) and destination DMA channel 9 (SID 173) are used, as previously described in Figure 3.

```
//*************************************************************************
// Open MDMA streams
//*************************************************************************
//
// RAW stream
//
DEBUGMSG(stdout, "Core1: Opening MDMA RAW stream\n" );
eResult = adi_mdma_Open (MDMA_STREAM_ID_RAW,
                         &MemDmaStreamMem_raw[0],
                         &hMemDmaStream_raw,
                         &hSrcDmaChannel_raw,
                         &hDestDmaChannel_raw,
                         NULL,
                         NULL);
if (eResult != ADI_DMA_SUCCESS)
{
    DEBUGMSG(stdout,"Failed to open MDMA RAW stream, Error Code: 0x%08X\n", eResult);
    return SHARC_LINK_ERROR;
}

//*************************************************************************
// Configure MDMA streams
//*************************************************************************
//
// RAW stream
//

// Disable the MDMA destination transfer complete interrupt
adi_mdma_EnableChannelInterrupt(hDestDmaChannel_raw,false,false);
// Get the channel SID for the MDMA destination complete interrupt
adi_mdma_GetChannelSID(hDestDmaChannel_raw,&nSid);
// Set interrupt to occur on Core 2 (unfortunate enumeration name in driver, see note)
adi_sec_SetCoreID(nSid, ADI_SEC_CORE_1);
// Enable and register the MDMA source transfer complete interrupt
adi_mdma_EnableChannelInterrupt(hSrcDmaChannel_raw,true,true);
eResult = adi_dma_UpdateCallback ( hSrcDmaChannel_raw,
                                   RawMemDmaCallback,
                                   hMemDmaStream_raw);
if (eResult != ADI_DMA_SUCCESS)
{
    DEBUGMSG("Failed to set DMA RAW stream callback, Error Code: 0x%08X\n", eResult);
    return SHARC_LINK_ERROR;
}
```

Listing 1. Initializing MDMA Using the MDMA Device Driver

As described in the previous note, the enumeration scheme leveraged by the CCES drivers and services uses the _0 suffix for the SHARC0 core (which is Core1 in the IDE) and the _1 suffix for the SHARC1 core (which is Core2 in the IDE).

After the SHARC0 core configures the ADAU1979 ADC, a callback function is invoked each time the SPORT fills one of the ping-pong buffers with an audio frame. This function is shown in Listing 2. All the SHARC0 core run-time processing is done in the callback function under the context of the SPORT ISR, thus leaving the non-ISR cycles (main thread) available for idle tasks. The callback sends the raw audio frame to the SHARC1 core using the MDMA0 stream, filters channels 0 and 2, and sends the filtered results to the SHARC1 core using the MDMA1 stream.

```
//*************************************************************************
// ADC callback - Called after each audio frame is filled
// Audio Format (32 bits/sample) - interleaved (CH0, CH1, CH2, Ch3, ...)
//*************************************************************************
void AdcCallback(void *pCBParam, uint32_t nEvent, void *pArg)
{
    switch(nEvent)
    {
        case ADI_SPORT_EVENT_RX_BUFFER_PROCESSED:
             //*************************************************************
             // Send RAW audio to slave SHARC
             //*************************************************************
             if( SHARC_linkSend( MDMA_STREAM_ID_RAW,             // MDMA A0
                                 (void *)pArg,
                                 DMASlaveDestinationAddress,
                                 1, AUDIO_BUFFER_SIZE ) != 0 )
             {
                   // If we get here, there is an error
             }

             // Filter the data
             SHARC0Filter( pArg, FilteredData, AUDIO_BUFFER_SIZE );

             //*************************************************************
             // Send RAW audio to slave SHARC
             //*************************************************************
             if( SHARC_linkSend( MDMA_STREAM_ID_FILTERED,        // MDMA A1
                                 (void *)FilteredData,
                                 (DMASlaveDestinationAddress+AUDIO_BUFFER_SIZE),
                                 1, AUDIO_BUFFER_SIZE ) != 0 )
             {
                   // If we get here, there is an error
             }

             *sharc_flag_in_L2 = *sharc_flag_in_L2 + 1;

             //*************************************************************
             // Return buffer to pool
             //*************************************************************
             Adau1979DoneWithBuffer( pArg );
             AdcCount++;
             break;
        default:
            break;
    }
}
```

Listing 2. SPORT RX DMA ISR Callback Function

The SHARC1 core also does all its processing under the context of callback functions. Audio channels 1 and 3 are filtered under the context of the raw (MDMA0 stream) transfer complete callback function. Filtered audio channels 1 and 3 from the SHARC1 core are merged with filtered audio channels 0 and 2 from the SHARC0 core under the context of the filtered (MDMA1 stream) transfer complete callback function. After the filter channels are merged, the resulting audio frame is copied into a SPORT TX DMA ping-pong buffer for output to the ADAU1962A DAC. The MDMA0 and MDMA1 stream callback functions are shown in Listing 3.

```
//*********************************************************************
// Interrupt handler for MDMA RAW data transfer complete
// Runs on SHARC1 (Core2)
//*********************************************************************
static void RawDataTransferFromMasterComplete(uint32_t SID, void *pCBParam)
{
    SHARC1Filter(   (int8_t *)MDMA_LOCAL_ADDR,
                    FilteredData, AUDIO_BUFFER_SIZE );
    ++RAWBuffersReceived;
}

//*********************************************************************
// Interrupt handler for MDMA FILTERED data transfer complete
// Runs on SHARC1 (Core2)
//*********************************************************************
static void FilteredDataTransferFromMasterComplete(uint32_t SID, void *pCBParam)
{
    ++FILTEREDBuffersReceived;
    // Merge the FilteredData buffer with the audio frame just received from the
    // Master SHARC.
    if( pSportOutputBuffer != 0 )
    {
        MergeAudioChannels( (void *)(MDMA_LOCAL_ADDR+AUDIO_BUFFER_SIZE),
                            FilteredData,
                            pSportOutputBuffer,
                            AUDIO_BUFFER_SIZE );
    }
}
```

Listing 3. MDMA Complete Interrupt Callback Functions

Conclusion

In a multi-core design, the efficiency of the ICC is paramount. Filters running on the SHARC+ cores execute at peak efficiency when acting on data in their on-chip L1 memory space. The ADSP-SC58x contains a powerful DMA subsystem supporting MDMA with signaling, providing the capability to copy data from one SHARC's L1 to the other SHARC's L1 at up to 1500MB/s. In addition, the MDMA engine can do ICC signaling by creating a transfer complete interrupt on both the source and destination SHARC+ cores. The example explained in this EE-note demonstrates how to use the MDMA engine for ICC in a quad-channel parallel pipeline audio talkthrough. The talkthrough splits the filtering load across both SHARC+ cores with the SHARC0 core filtering channels 0 and 2 while the SHARC1 core filters channels 1 and 3 in parallel.

References

[1] *Associated ZIP File (EE383v01.zip) for MDMA-Based Dual-SHARC+ Parallel Pipeline Audio Talkthrough (EE-383).* December 2015. Analog Devices, Inc.

Document History

Revision	**Description**
Rev 1 – December 17, 2015 by Eric Gregori	Initial Release