a **Engineer To Engineer Note EE-143**

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

Understanding DMA on the ADSP-TS101

Last modified: 25/10/01

Contributed By: A.C.

The following Engineer-to-Engineer note will discuss various forms of Direct Memory Accesses on the TigerSHARC family of processors. Assembly code examples are provided with this note and explained.

The ADSP-TS101 is the first member of the TigerSHARC® DSP family - ADI's new family of ultra high-performance DSPs optimized for telecommunications infrastructure as well as 16-bit fixed point and 32-bit floating point general purpose applications. The ADSP-TS101 features a static superscalar architecture that combines RISC, VLIW and standard DSP functionality.

Introduction

Direct Memory Access is a mechanism for transferring data without the core being involved. The TigerSHARC on-chip DMA controller allows these transactions of data to take place as a background task freeing up the processor core for signal processing operations.
The TigerSHARC includes 14 DMA channels. Four channels are dedicated to external memory devices, eight to the link ports and two to the Auto DMA registers. These channels are shown in **figure 1** in order of priority. The DMA controller is capable of performing several types of data transfers.

- Internal memory → External memory and memory-mapped peripherals
- Internal memory → Internal memory of other TigerSHARCs residing on the cluster bus
- Internal memory → Host processor
- Internal memory → Link port IO
- External memory → External peripherals
- External memory → Link port IO
- Link port IO → Internal memory
- Link port IO → External memory
- Link port IO → Link port IO

Most of the types listed above are demonstrated in the examples described in this document.

Highest priority

Auto DMA Registers
Channel 13
Channel 12
Receiving Links
Channel 11 (Link 3)
Channel 10 (Link 2)
Channel 9 (Link 1)
Channel 8 (Link 0)
Transmitting Links
Channel 7 (Link 3)
Channel 6 (Link 2)
Channel 5 (Link 1)
Channel 4 (Link 0)
External Ports (EPs)
Channel 3 (DMAR3)
Channel 2 (DMAR2)
Channel 1 (DMAR1)
Channel 0 (DMAR0)

Lowest priority

Figure 1: DMA channels on TigerSHARC

Copyright 2001, 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.

The afore mentioned data transfer types can be broken down into the following main categories:

- Internal memory ←→ Cluster bus. This transfer can be done in both directions and requires the programming of two Transfer Control Blocks, otherwise referred as a TCB. One is a transmitting TCB and the other a receiving TCB.

- Auto DMA register → Internal memory. This requires one receiver TCB.

- Internal/external memory → Link ports. Requires one transmitter TCB.

- Link ports → Internal/external memory. Requires one receiver TCB.

- Link port → Link port. Requires one receiver TCB.

The Transfer Control Block is a quad word (128-bit) that contains the vital information required to perform a DMA. In the case of a transmitter TCB the four words contain the address of the source data, the number of words to be transferred, the address increment and the control bits.

In the case of a receiver TCB these four words contain destination address, the number of words to be received, the address increment and the control bits.

The TCB register is a 128-bit register, consisting of four 32-bit registers as shown in **figure 2**.

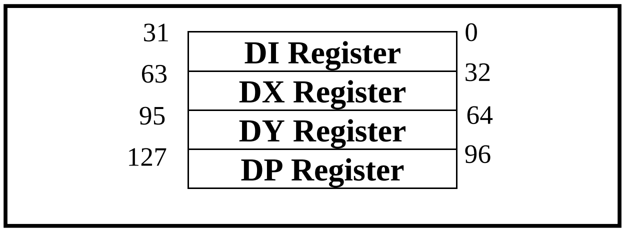


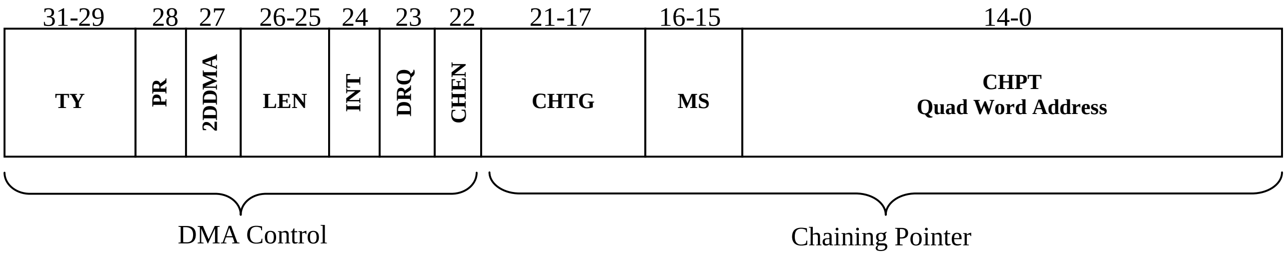
Figure 2: DMA TCB Register

The DI register is the 32-bit index register for the DMA. This contains the source or destination of the data to be transmitted or received and can point to internal, external memory or the link ports.

The DX register contains a 16-bit count value and a 16-bit modify value. The count value is stored in the upper 16 bits (16-31) and the modify in the lower (0-15). If a two-dimensional DMA is enabled then this register contains the modify and count values for the X dimension only. The value of X count must always be the number of normal (32-bit) words to be transferred. Likewise, the modify value is the number of normal words to modify the count by. For example if we wanted to transmit four quad words (16-normal words), then the count value would be 0x10 and the modify value 0x4 if the operand length in the DP register is set to Quad word. If the operand length was set to Long word then the modify value would be 0x2. These values of course are application specific however and would result in the entire data being transferred in order, by altering the modify values data can be transferred in any order required.

The DY register is used in conjunction with the DX. This register contains the 16-bit modify and 16-bit count values for the DMA in the Y dimension. If two-dimensional DMA is not selected then this register is not used and the contents are irrelevant.

The DP register contains all the control information for the DMA. This register is split into two main fields. The first contains all the control information and the second the chaining information. The breakdown of this register is shown in **figure 3**.



31-29	28	27	26-25	24	23	22	21-17	16-15	14-0
TY	**PR**	**2DDMA**	**LEN**	**INT**	**DRQ**	**CHEN**	**CHTG**	**MS**	**CHPT** **Quad Word Address**

CHPT Chain Pointer:
Bits [14:0] Bits 16:2 of next TCB address.
Bits 1:0 of address are zero due to quad alignment.

MS Memory Select for chain pointer:
Bits[16:15] 00…Memory Block 0
01…Memory Block 1
10…Memory Block 2
11…Reserved

CHTG Chaining destination channel:
Bits[21:17] Specify device type:
00000…Channel 0 Source TCB
00001…Channel 0 Destination TCB
00010…Channel 1 Source TCB
00011…Channel 1 Destination TCB
00100…Channel 2 Source TCB
00101…Channel 2 Destination TCB
00110…Channel 3 Source TCB
00111…Channel 3 Destination TCB
01000…Channel 4 Link 0 Source TCB
01001…Channel 5 Link 1 Source TCB
01010…Channel 6 Link 2 Source TCB
01011…Channel 7 Link 3 Source TCB
10000…Channel 8 Link 0 Dest TCB
10001…Channel 9 Link 1 Dest TCB
10010…Channel 10 Link 2 Dest TCB
10011…Channel 11 Link 3 Dest TCB
10110...Channel 12 IFIFO DMA0 Dest TCB
10111...Channel 13 IFIFO DMA1 Dest TCB

CHEN Chaining Enabled:
Bit[22] 0…No Chaining
1…DMA loads chaining targets channel TCB registers from internal memory

DRQ DMA Request Enable:
Bit[23] 0…Once DMA channel is enabled, entire block is transferred
1…DMA issies transactions only upon request

INT Interrupt Enable:
Bit [24] 0…DMA interrupt is disabled
1…DMA interrupts the core after transferring whole block

LEN Operand Length:
Bits[26:25] 00…Reserved
01…Normal word
10…Long word
11…Quad word

2DDMA Two-dimensional DMA:
Bits[27] 0…One-dimensional DMA
1…Two-dimensional DMA

PR Determines Priority:
Bit[28] 0…DMA request priority is normal
1…DMA request priority is high

TY Specify the device type:
Bit[31:29] 000…DMA Disabled
001…IO Link ports
010…Internal Memory
011…Reserved
100…External Memory
101…External IO Device (fly-by)
110…Boot EPROM
111…Reserved

Figure 3: TCB DP Register

The cluster bus (EP) TCBs are loaded by writing to the DCSx registers for the source TCB and the DCDx registers for the destination TCBs, where x can be any value ranging from 0 to 3. To load the TCBs for the link ports or the AutoDMA registers, the DCx registers are written to where x can be any value from 4 to 13 depending on whether you are writing to link port receive, transmit or AutoDMA channels as shown in **figure 1** previously.

DMA Demonstrations

There are five demonstrations described in the following pages. These are:

- SDRAM
- Link port
- 2 Dimensional
- Chaining
- Ext_mem_to_link

These examples have been provided in both C and assembly language and are for use on VisualDSP++® Release 2.0 for the TigerSHARC processor. The Link port, Chaining and Ext_mem_to_link examples are multiprocessor projects created for use in a TigerSHARC Multiprocessor system along with either an Apex-ICE® or Summit-ICE® JTAG Emulator. Although the simulator environment does not support multiprocessing at the time that this document was published, it is possible to simulate each executable created from the multiprocessor project individually by setting up the DMA File I/O Configuration window. This enables the user to set up source and destination files for all DMA channels allowing simulation of link port and external port transfers.
All the other examples are single processor examples and can also be used successfully on hardware or simulated.

In all the examples ***N*** is defined at the top of the program. This is the number of normal words that are contained within the file ***tx_data.dat***. This file contains the data that is used to be transmitted in all the DMA transfers. The code in the examples has been written in a way that the value N can be altered without having to alter the count value in the TCB. All examples use an operand length relating to quad word transfers. If the operand length is altered then the modify value must be altered to reflect this.

By default, all DMA channel interrupts are enabled in the IMASK register. In all the assembly language programs, these interrupts have been disabled and only the interrupts for the channels being used within that example are enabled. All programs except for Chaining generate interrupts after every DMA block transfer. The interrupt service routine is ***_dma_int*** in the assembly examples and simply consists of a **nop** and **rti** command. It is important to have the **nop** (in these examples) before the **rti** as the RETI register is updated only when the first instruction of the interrupt service routine reaches the Ex2 stage of the pipeline. This means that the first instruction cannot be an instruction that uses its value such as RTI or RETI. Within the C examples the interrupt service routine is named ***dmax_int***, where ***x*** is the number for the DMA channel that caused the interrupt. These interrupt service routines cause a sentence to be printed to the output window stating that the dma has completed. The Chaining example differes from the rest slightly as all dma transfers have their interrupts disabled except the last dma transfer in the chain.

The following sections describe all the code examples that compliment this EE Note. The areas of the IO processor shown in **figure 4** that are utilised are described for each transfer that takes place.

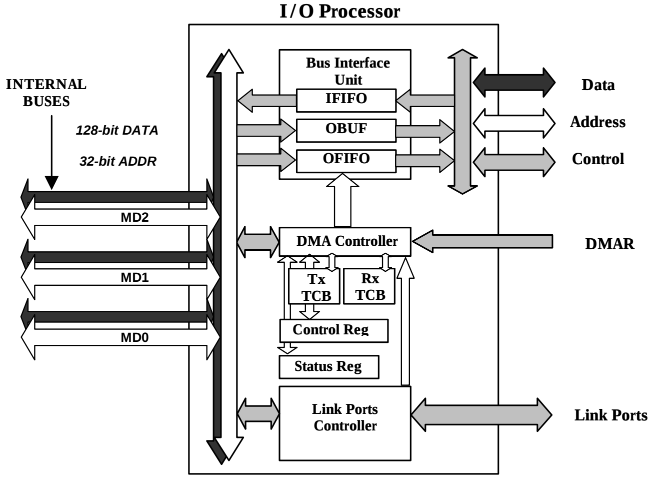


Figure 4: IO Processor

External_mem

This example shows the DMA transfer from internal memory of the TigerSHARC processor to external SDRAM and back again. A simplified diagram showing the layout and connections of the SDRAM to the TigerSHARC processor DSP0 on the MBUB is shown in **figure 5**.

In the example, DMA channel 0 is used to transmit data to SDRAM and channel 1 is used to transfer the data back from SDRAM to internal memory. The data to be transmitted, from the ***tx_data.dat*** file, is stored in the internal memory under the variable name **data_tx**. The data transferred to the SDRAM is stored as the variable **sdram_data**, and the data transferred from the SDRAM to internal memory is stored as the variable **data_rx**.

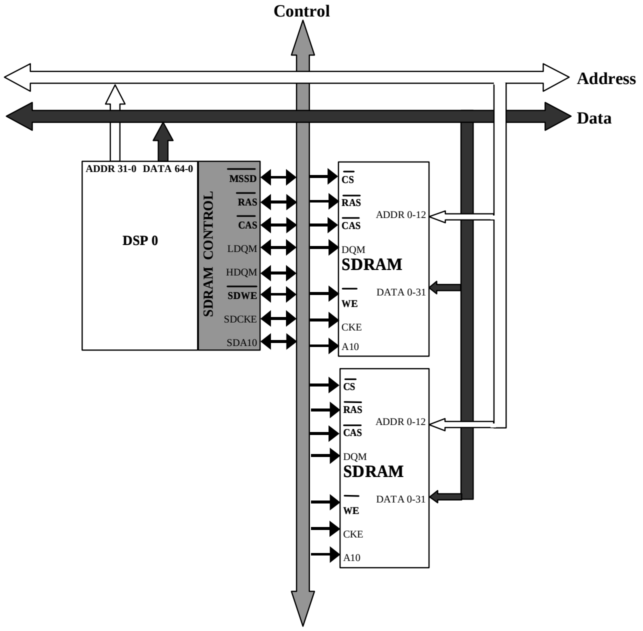


Figure 5: System diagram for SDRAM example

As this transfer of data falls into the internal memory to cluster bus category, two TCBs must be written to for each transfer. The source TCB for the first transfer (DMA channel 0) is loaded as follows:

```
XR0 = data_tx;;        ← DI Register
XR1 = 0x00400004;;  ← DX Register
XR2 = 0x00000000;;  ← DY Register
XR3 = 0x47000000;;  ← DP Register
DCS0 = XR3:0;;
```

The destination TCB is loaded with the following values:

```
XR8 = sdram_data;;
XR9 = 0x00400004;;
XR10 = 0x00000000;;
XR11 = 0x87000000;;
DCD0 = XR11:8;;
```

The values in registers XR2 and XR10 are irrelevant due the fact that 2-dimensional DMA is not selected. As soon as both the source and destination TCBs are loaded with values (with a non-zero TY field) the DMA transfer starts. The processor then remains in the idle state due to the idle command following the write to the DCD0 TCB until the DMA is completed at which point an interrupt occurs and the ***_dma_ int*** vector interrupt routine is then run. The transfer of the data back to internal memory from the SDRAM is executed with minor alterations to the afore mentioned commands. The source TCB is loaded with the contents of registers XR11:8 and the destination TCB is written with the values of registers XR3:0 where XR0 was altered from data_tx to data_rx.

XR0 = data_rx;;
DCS1 = XR11:8;;
DCD1 = XR3:0;;

Notice how DMA channel 1 is used for the transfer by replacing DCS0 and DCD0 with DCS1 and DCD1 respectively. Again the processor remains in the idle state until the DMA has completed and an interrupt is generated.

The external write transaction is initiated by the internal master, either the DMA or the core, in the case of DMA transfers the internal master is the DMA. This happens as soon as both the source and destination TCBs are written to. The transaction is initiated on one of the three internal buses. The transaction is then strobed by the OFIFO and in turn driven onto the external bus. The address for the transaction is passed from the DMA controller to the OFIFO and out onto the external port address bus. The data for the transaction is strobed from the internal bus into the OBUF and out onto the external data bus.

In the case of the read transaction, the read is once again initiated by the internal master, the DMA, as soon as both TCBs are written to. The data read on the external transaction is strobed by the IFIFO and then written to its target which was defined by the original internal transaction.

Link_port

The link port example is a multiprocessor example which consists of a single project with two assembler or source files. One file for the transmitting processor (DSP0 in this example) and one for the receiving processor (DSP1 in this example). The data to be transmitted is stored in **Link_data_tx**, this data is transferred to the receiving processor via link port 1. The receiving processor receives the data in through link port 0 and stores the received data to internal memory in the variable **link_data_rx**. The data is then transferred back from the receiving processor, through link port 0 and received through link port 1 on the transmitter processor. The received data is then stored in the variable **link_data_rx**.

A simplified diagram of the system setup is shown below in **figure 6** and the link port architecture of the TigerSHARC in **figure 7**.

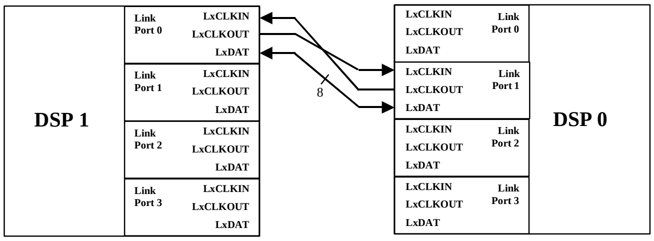


Figure 6: Link Port Layout in example

In this example as the cluster bus is not being used only one TCB has to be written to for each DMA.

As the transmitter is transmitting and receiving through link port 1, DMA channels 5 and 9 are used. On the receiving DSP which receives and then transmits through link port 0, DMA channels 4 and 8 are used.

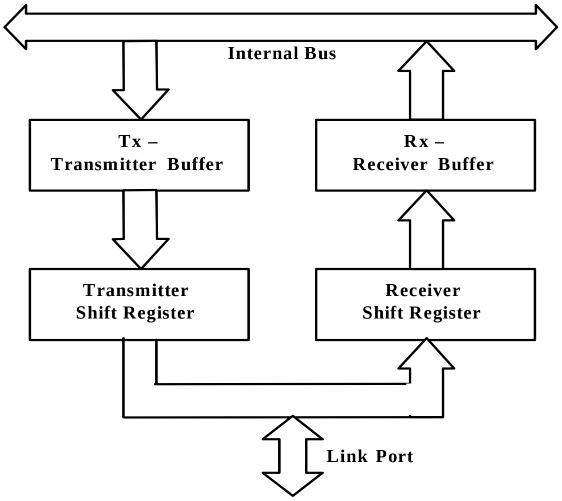


Figure 7: Link Port Architecture

When the link port DMA TCB is written to enabling the DMA channel, if the transmit buffer is empty (for transmit channel) or if the receive buffer is full (for receive channel), a DMA request is immediately issued.

The link port only issues a DMA request to the transmit DMA channel, when both the link port TCB is written to (enabling the DMA channel) and the Tx register is empty. Data is written to the Tx register and then to the shift register, when it is empty, and then transmitted. All the registers of **figure 7** are 128-bit registers so after the quad word is copied to the shift register after being placed into the Tx register, new data is written to the Tx register. The link transmitter tries to transmit all data that is written to the transmit shift register. The receiver enables data movement in only when the receive shift register is empty, at which point the received data is shifted into the receive shift register. After the entire quad-word is received, the receiver waits until the Rx register is free and copies the data from the shift register to the Rx register. Once the shift register is free it can receive data again.

2_dimensional_dma

The two dimensional DMA example is very similar to previously described external_mem example, the main difference being that the DMA from internal memory to external memory and back again is achieved with the use of the DX and DY registers within the source and destination TCBs. The count number of words in the DX and DY registers is set to 8 resulting in 8x8 normal words being transferred. The same DMA channels are used as described in external_mem example.

The example can be altered to transfer an 8x8 matrix of words to any other size by modifying the XCOUNT and YCOUNT values defined at the top of the code.

XCOUNT

YCOUNT

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	58	60	61	62	63

XCOUNT = 0x8
YCOUNT = 0x8

X modify = 0x4 to transfer the 2 quad words per row

Y modify = difference between address of last element in a row (4) and the first element in the following row (8)

Figure 8: X and Y count/modify for 8x8 matrix

XCOUNT

YCOUNT

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	58	60	61	62	63

XCOUNT = 0x10 **YCOUNT = 0x4**

X modify = 0x4 to transfer the 4 quad words per row

Y modify = difference between address of last element in a row (12) and the first element in the following row (16)

Figure 9: X and Y count/modify for 16x4 matrix

DMA_chaining

The DMA_chaining example is a multiprocessor example. There is one project with two assembler or source codes which create two executables, similar to that of the link_port example, a Chained_tx and a Chained_rx. The Chained_tx example performs a chained DMA consisting of four transfers. The first transfer is data from internal memory to external SDRAM using DMA channel 0. The second DMA transfers the first half of the data just received in SDRAM back to internal memory using DMA channel 1. The third DMA transfers the second half of the data that was written to SDRAM directly to link port 1 of DSP0 (Chained_tx) for transmission to the receiving processor using DMA channel 5. The fourth DMA uses link port 1 receive DMA channel 9 to receive data transmitted by DSP1 (Chained_rx) and store it in internal memory.

The receiving processor, DSP1 is loaded with the Chained_rx program which consists of a DMA transfer from link port 0 to internal memory and then from internal memory back to link port 0, using DMA channels 4 and 8 respectively. This processor does not utilise the chaining method.

With DMA chaining, the DCS and DCD TCBs only need to be written to once. Once the DMA has completed, the DMA controller automatically loads the TCB values for the next DMA from internal memory into the TCB registers.

Chaining is enabled by setting the chain enable bit (bit 22) in the TCB DP register and the chain pointer to a valid address.

The CHPT field within the DP register can be loaded at any time during the DMA sequence. This allows a DMA channel to have chaining disabled until some event occurs that loads the chain pointer field (CHPT) with a target address, sets the chaining enable bit (CHEN) and difines the chaining destination channel (CHTG).

In the example two chains are set up. This is due to the the fact that some of the DMA transfers to occur require the use of the external port DMA channels. External port DMA channels require the setting of of both a source and destination TCB. The two chains can be seen in **figure 10**.

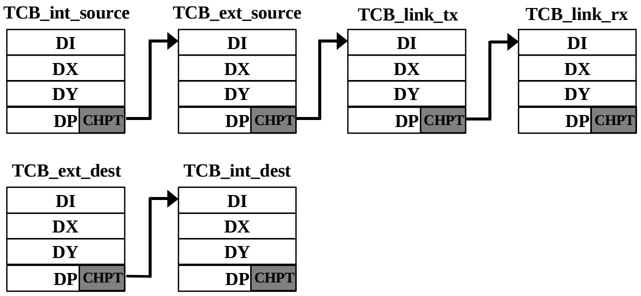


Figure 10: DMA chaining

The TCB for DMA channel 0 (DCS0) is initialized with the quad word stored in the variable array TCB_int_source. The destination TCB for DMA channel 0 (DCD0) is initialized with the quad word stored in the variable array TCB_ext_dest. As soon as the DCD0 TCB is written to the DMA chaining is kicked off. This first DMA transfers the data from internal memory to external SDRAM as in the example Ext_mem. Once this transfer is completed the quad word stored in TCB_ext_source is written into DCS1 and the the quad word stored in TCB_int_dest is written into DCD1. The writing to both these source and destination TCBs starts the second DMA transfer consisting of the transfer of the first half of the data that was written to the SDRAM in the first transfer being written back to the internal memory of the processor and stored in ***data_rx***. Once this transfer is completed the second chain is terminated as the transfers to follow use the link ports and this only requires the initialization of a single TCB and not a source and destination TCB. The third transfer to take place (TCB_link_tx), transfers the second half of the data stored in the external memory directly to link port 1 of DSP0 for transfer to DSP1. Once DSP1 receives all the data then DSP0 loads the final TCB (TCB_link_rx) and receives the data back and stores it in the second half of ***data_rx*** making up the original 64 word block that was originally transmitted.

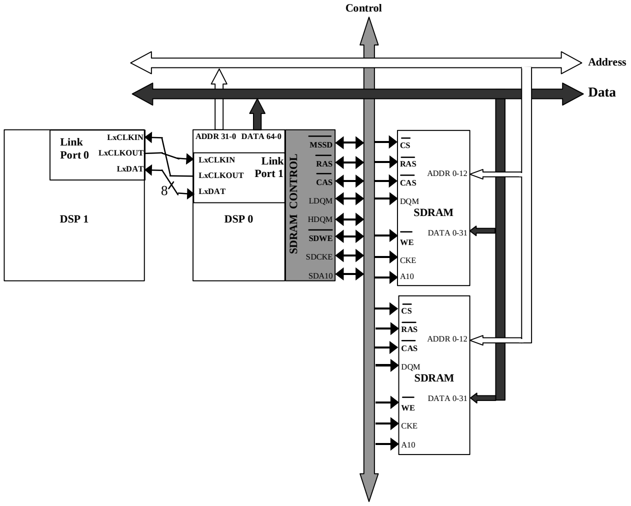


Figure 11: System diagram for Chaining example

A point to note in the DMA chaining examples is that there is a slight difference in the declaration of the TCBs in the assembly example and the C example. In the assembly example, when the project is built, the variables are placed into memory in the order that they appear in the code example. This differs slightly to the C example where a separate memory section was defined within the linker description file to store the TCB data. This is necessary as the addresses of the next TCB to be loaded must be known. By defining an individual section in memory for the TCB you have better control of where the variable arrays are stored and are assured that no other variable is placed between the TCBs making their location in memory less predictable.

It is possible to insert a high priority DMA or chain into an active DMA chain. To achieve this the current channel transactions must first of all be suspended by setting the corresponding pause bit in the DCNT register.

Bit 0-7 of the DCNT register are for DMA channels 0-7. Bits 10-13 for DMA channels 8-11 and bits 16-17 for DMA channels 12 and 13 respectively. The bits in the DCNT register is set by writing to the DCNST alias register. The value written to this register is Ored with DCNT, and the result loaded into DCNT.

Once the current DMA transfer has been halted a new chain is inserted by the TigerSHARC core by writing the CHEN, CHTG and CHPT bits in the DP register of the TCB. Writing to an active TCB register will cause a HW ERROR interrupt. Once the TCB is updated the pause bit in the DCNT register must be reset and the data transfer continues from where it left off. The pause bit is reset by writing to the DCNTCL alias register. The value written to this register is ANDed with DCNT, and the result loaded into DCNT.

Ext_mem_to_link

Again this example is a multiprocessor example. This example shows how to declare and use shared external memory. This requires the modifying of the linker description file to accommodate for shared memory and multiple processor memory layout. Two executables are generated when the project is built, *External_mem_to_link_tx.dxe* and *External_mem_to_link_rx.dxe*. The *External_mem_to_link_tx* program consists of 3 DMA transfers. The first is from internal memory to shared external SDRAM using DMA channel 0, the second is from the SDRAM directly to link port 1 for transmission using DMA channel 5 hence not using any internal memory to store the data first as in previous examples. The final DMA for this processor is the receiving of data through link port 1 which is transferred to internal memory with the use of DMA channel 9.

The *External_mem_to_link_rx* program receives data through link port 0 and transfers it directly to the shared external SDRAM using DMA channel

8. With the use of DMA channel 5, the data just placed into SDRAM is then transferred directly back to link port 0 to be transmitted back to the other processor which receives it through link port 1 as previously described.

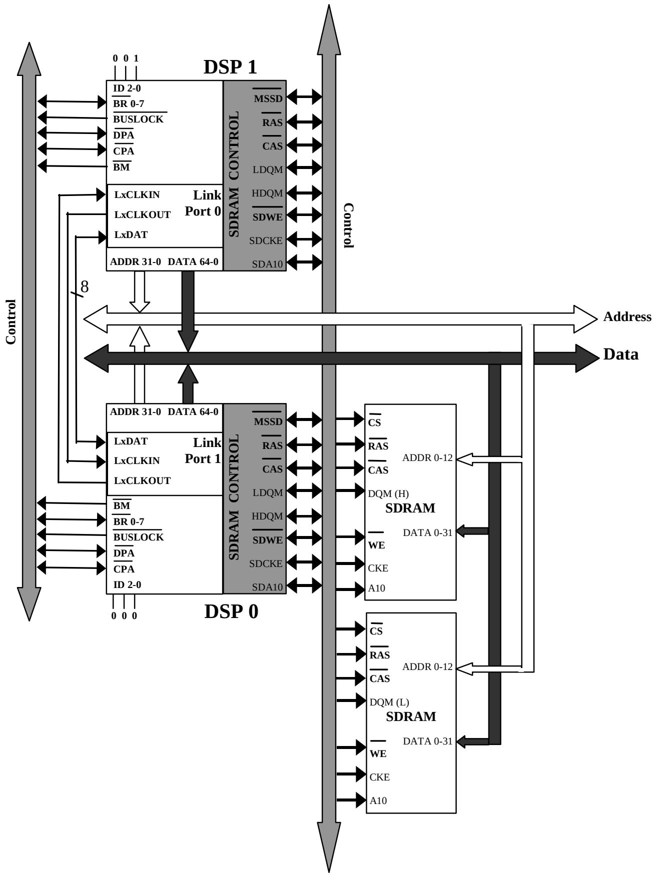


Figure 12: System diagram for Ext_mem_to_link example

External_mem_to_link_tx.dxe is loaded into DSP0 and *External_mem_to_link_rx.dxe* is loaded into DSP1 for use on the MBUB. In the shared external memory, the data written to by DSP0 is stored in ***sdram_data*** and the data written by DSP1 is stored in ***link_data_rx2***. If the shared memory is not defined then although both the external memory variables are viewable to both processors DSP0 places ***sdram_data*** at memory location 0x4000000 and ***link_data_rx2*** at 0x4000040. On the other hand DSP1 places ***link_data_rx2*** at 0x4000000 and ***sdram_data*** at 0x4000040. Thus when DSP1 writes data to the SDRAM it overwrites what DSP1 placed in SDRAM.

Note: The C examples perform identical DMA transfers as the assembly coded examples. All variables are identical. The only difference between the two lots of code being the interrupt service routines. In the C examples there is an interrupt service routine for every interrupt and during this isr, a line is printed to the output window stating that the dma transfer has completed.

References

TigerSHARC DSP Hardware Specification Part # ADSP-TS101, Analog Devices Inc.

TigerSHARC DSP Instruction Set Specification Part # ADSP-TS101, Analog Devices Inc.

VisualDSP++ 2.0 Linker & Utilities Manual for TigerSHARC DSPs, Analog Devices Inc.

VisualDSP++ 2.0 Users Guide for TigerSHARC DSPs, Analog Devices Inc.