Application Report
SPRA529A - April 2002

TMS320C6000 DMA Example Applications

David Bell
Digital Signal Processing Solutions

ABSTRACT

The TMS320C6000™ on-chip direct memory access (DMA) controller from Texas Instruments is used to transfer data between two locations in the memory map in the background of CPU operation. Typically, the DMA is used to:

- Transfer blocks of data between external and internal data memories
- Restructure portions of internal data memory
- Continually service a peripheral
- Page program sections to internal program memory

Four DMA channels can be programmed to perform one or more of these tasks while the CPU is executing a program. Channels can be configured to run continuously throughout the device's entire operation, with only one setup required. The fixed priority scheme between the channels allows high-priority synchronous transfers to be performed during a low-priority block transfer. The DMA channels can communicate their status to the CPU through interrupts, to provide the CPU with control over their operation. Most applications require only an initial setup of DMA control registers, with little intervention by the CPU to maintain their operation.

Contents

List of Figures

List of Tables

1 Introduction

The TMS320C6000 on-chip DMA controller transfers data from one memory-mapped location to another, without the intervention of the central processing unit (CPU). The DMA transfers data between internal memory, peripherals, and external devices in the background, allowing the CPU to remain active during data transfers. Four DMA channels can be independently configured to perform different types of transfers.

The DMA is highly flexible, with many different types of transfers available to enable faster throughput by the CPU through data organization. With the DMA, data can be transferred to and from internal program memory, internal data memory, an external memory space, an external analog front end (AFE) circuit, or the multichannel buffered serial ports (McBSPs). The DMA also allows data currently in memory to be reorganized to increase the CPU's effectiveness.

Each channel of the DMA has the following set of registers that must be configured prior to beginning a data transfer:

- Primary control register – Used to configure the transfer
- Secondary control register – Used to enable interrupts to the CPU, and to monitor the channel's activity
- Transfer counter register – Used to keep track of the transferred elements
- Source address register – Memory location from which the element is transferred
- Destination address register – Memory location to which the element is transferred

In addition, several global DMA registers can be used by any of the DMA channels to perform more complicated transfers:

- Global address registers (A, B, C, and D) – Used as either a split address or address reload value
- Global index registers (A and B) – Used to control address updates during a transfer
- Global count reload registers (A and B) – Used to reload the transfer counter register of a DMA channel

Each of the global DMA registers can be used by any of the DMA channels, and more than one channel can use the same register at a time.

One additional DMA register, the auxiliary control register, is used to set the priority of the auxiliary channel with respect to the four main DMA channels and the CPU. The auxiliary channel is used by the host interface to access the C6000™ memory.

The *TMS320C6000 Peripherals Reference Guide* (SPRU190) offers a complete description of the DMA structure, and should be used in conjunction with this document.

C6000 is a trademark of Texas Instruments.

2 System Structure

All accesses to external memory spaces must go through the external memory interface (EMIF). External memory types supported on the C6000 are synchronous dynamic random-access memory (SDRAM), synchronous burst static random access memory (SBSRAM), and asynchronous memories. To understand how to configure different memory spaces, see the *TMS320C6000 Peripherals Reference Guide* (SPRU190).

External analog front–end (AFE) circuits predominantly use the asynchronous memory interface of the C6000. A typical AFE configuration includes a data-in address, a data-out address, a read sync signal, and a write sync signal. Synchronization events are connected to one of the four external interrupt pins of the device (EXT_INT[7:4]).

The McBSP is the only on-chip peripheral that can require servicing by the DMA. Each McBSP has a data receive register (DRR), a data transmit register (DXR), a transmit-event signal (XEVT), and a receive-event signal (REVT). The DRR and DXR are memory-mapped registers. The events occur whenever data is transferred out (XEVT) or transferred in (REVT).

Internal data memory is divided into several 16-bit banks. The TMS320C6201 includes four banks: the C6201B and C6202 have two blocks of four banks, with each block occupying half of the data memory, and the C6701 has two blocks of eight banks. Each bank can only be accessed once per cycle, either by the DMA or by one of the CPU sides (A or B). If both the DMA and the CPU attempt to access the same bank during the same cycle, the priority bit set in the DMA channel’s priority control register determines the order in which the access is granted.

Internal program memory always gives the CPU priority over the DMA. For the DMA to access program memory, there must be time slots during which it can get in. These slots occur when a fetch packet (eight instructions) contains multiple execute packets (a group of instructions executed in one cycle). This leaves cycles in which the CPU is not requesting a fetch packet, and the DMA can access the program memory. The C6202 has two blocks of program memory. The DMA can access one block, while the CPU is executing code from the second, without contention.

3 Data Relocation

The purpose of the DMA is to move data elements from one location to another. Through proper configuration of the DMA channel control registers, the data to be transferred can be moved in its current format, or restructured to fit a particular application.

The simple case is a block move, in which a contiguous memory space is copied, unaltered from one location to another. This transfer requires the minimum amount of setup, and is usually performed either to transfer a program section from an external memory location to internal program memory, or to transfer a data section between external memory and internal data memory.

By taking advantage of some DMA features, a more complicated transfer can be performed, in which a section of data is reorganized during the transfer. One example of this is sorting, in which a data block divided into contiguous frames of equal size is reorganized in memory by ordinal location within a frame. In other words, the first element of the first frame is located next to the first element of the second frame. This type of transfer is frequently performed when multiple frames of data are arriving to the device via the serial port (or AFE), or when data arrays located in external memory are brought on-chip.

The following examples demonstrate how the DMA can be used to relocate and reorganize data.

3.1 Block Move Example

The block move is used to simply transfer a block of contiguous memory from one location to another. This is ordinarily done to move a data or program section from external memory to internal memory, where the CPU can do single-cycle accesses. For this transfer, four of the five basic registers mentioned in section 1 must be configured: the primary control register, the transfer counter register, the source address register, and the destination address register. Figure 1 depicts a block transfer of elements.

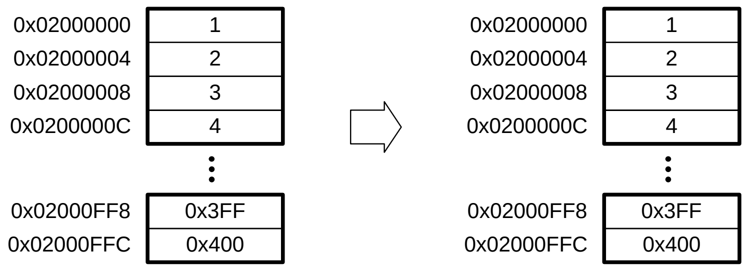

Figure 1. Block Move Example Diagram

Consider an example in which a 1k block of contiguous 32-bit elements is transferred from off-chip memory, located at the base address of CE2 (0x02000000) to the base of internal data memory (0x800000000). To initialize this transfer, the following values should be set for the four control registers:

Primary control register	=	0x00000050
Source address register	=	0x02000000
Destination address register	=	0x80000000
Transfer counter register	=	0x00000400

The primary control register is shown in Figure 2. The transfer counter register is shown in Figure 3.

31 30	29 28	27	26	25	24	23 19	18 16
DST RELOAD	SRC RELOAD	EMOD	FS	TCINT	PRI	WSYNC	RSYNC[4:2]
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

15 14	13	12	11 10	9 8	7 6	5 4	3 2	1 0
RSYNC[1:0]	INDEX	CNT RELOAD	SPLIT	ESIZE	DST DIR	SRC DIR	STATUS	START
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R/W-0

Legend: R/W = Read/Write; R = Read only

Figure 2. Primary Control Register

31 16	15 0
FRAME COUNT	ELEMENT COUNT
R/W-0	R/W-0

Legend: R/W = Read/Write

Figure 3. Transfer Counter Register

The primary control register should be configured as follows:

DST RELOAD	= 00	INDEX	= 0
SRC RELOAD	= 00	CNT RELOAD	= 0
EMOD	= 0	SPLIT	= 00
FS	= 0	ESIZE	= 00
TCINT	= 0	DST DIR	= 01
PRI	= 0	SRC DIR	= 01
WSYNC	= 00000	STATUS	= 00
RSYNC	= 000	START	= 00
RSYNC	= 00		

The transfer counter register should be configured as follows:

FRAME COUNT	= 0x0000
ELEMENT COUNT	= 0x0400

The setting of 01b in the DST DIR and SRC DIR bit fields causes the DMA channel to increment both the source address and the destination address by one element size (4 bytes in this example) following the transfer of each element.

To initiate the transfer, write a value of 01b to the START bit field.

3.2 Extremely Large Block Move Example

Occasionally there is a need to perform a block move of a large section of memory containing more than 65535 elements (the maximum value of ELEMENT COUNT). This transfer type is typically used to perform a data dump from external memory to an off-chip AFE, or to initialize a memory space from an AFE.

This transfer is essentially the same as the basic block move in the previous example, except multiple frames must be used. Using the frame count in conjunction with the element count, it is possible to transfer a single block of up to 0xFFFE0001 (4.3 GB) elements. This is much greater than the 65535 possible using the element count alone. Figure 4 shows an extremely large block transfer of data.

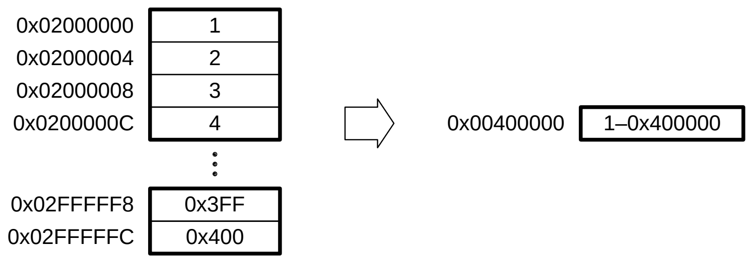

Figure 4. Extremely Large Block Move Example Diagram

The following statements are true for a large block transfer:

- If the address is set to be adjusted using a programmable value ((SRC/DST)_DIR = 11b), the frame index must equal the element index.
- Frame synchronization must be disabled. This prevents a synchronization event from being required in the middle of the transfer.

- The number of elements transferred in the entire block is ((F – 1) x Er) + Ei, where:
  - F = Initial value of the frame count
  - Ei = Initial value of the element count
  - Er = Element count reload value

If the numbers of elements to be transferred is constant for a given application, suitable values can be used explicitly in a program. For a majority of transfer lengths, many count and reload values provide the same performance.

If the block length is not a fixed amount, but is established during run-time, an algorithm to determine the count and reload values during execution is a more convenient solution.

Using the above information, a simple formula can be created to calculate F, Ei, and Er from a given block size. One possible formula is as follows:

- Ei = 15 LSBs (least-significant bits) of total element count. Fix to 0x8000 if 15 LSBs are all 0.
- Er = 0x8000 (fixed)
- F = Total element count divided by Er, plus 1. Do not add 1 if Ei is forced to 0x8000.

The following C code performs the above calculations:

```
F       =(XFER_SIZE >> 15)+1;
Ei      = XFER_SIZE & 0x7FFF;
if (!Ei){
     Ei = 0x8000;
     F –= 1;
}
Er      = 0x8000;
```

For this set of equations, the maximum transfer size is 0x7FFF7FFF (2.15 GB).

For this example, assume that the entire memory space CE2 (16 MB) is to be transferred to an off-chip peripheral located at 0x00400000 (CE0). This transfer is 0x00400000 32-bit words. For this, the following values must be assigned to the DMA registers:[1]

Primary control register	= 0x00000010
Source address register	= 0x02000000
Destination address register	= 0x00400000
Transfer counter register	= 0x00808000
Global count reload register A	= 0x00008000

[1] The sample formulas were used to determine the transfer counter and global count reload A values. If different formulas are used to obtain the count values, those numbers may be different.

The primary control register for channel 0 (see Figure 2) should be configured as follows:

DST RELOAD	=	00	INDEX	=	0
SRC RELOAD	=	00	CNT RELOAD	=	0
EMOD	=	0	SPLIT	=	00
FS	=	0	ESIZE	=	00
TCINT	=	0	DST DIR	=	00
PRI	=	0	SRC DIR	=	01
WSYNC	=	00000	STATUS	=	00
RSYNC	=	000	START	=	00
RSYNC	=	00			

The transfer counter register (see Figure 3) should be configured as follows:

FRAME COUNT	=	0x0080
ELEMENT COUNT	=	0x8000

The setting of 01b in the SRC DIR bit field would cause the DMA channel to increment the source address by one element size (4 bytes in this example) following each element. Since the destination is a fixed address, set DST DIR to 00b. See Figure 2.

To initiate the transfer, write a value of 01b to the START bit field.

3.3 Data-Sorting Transfer Example

Many applications require the use of multiple data arrays. It is often desirable to have the arrays arranged so that the first elements of each array are adjacent, the second elements are adjacent, and so on. Often, this is not the format in which the data is presented to the device. Either data is transferred via a peripheral, with the data arrays arriving one after the other, or the arrays are located in memory, with each array occupying a portion (frame) of contiguous memory spaces. For these instances, the DMA can be configured to reorganize the data into the desired format. Figure 5 shows the data sorting of element arrays.

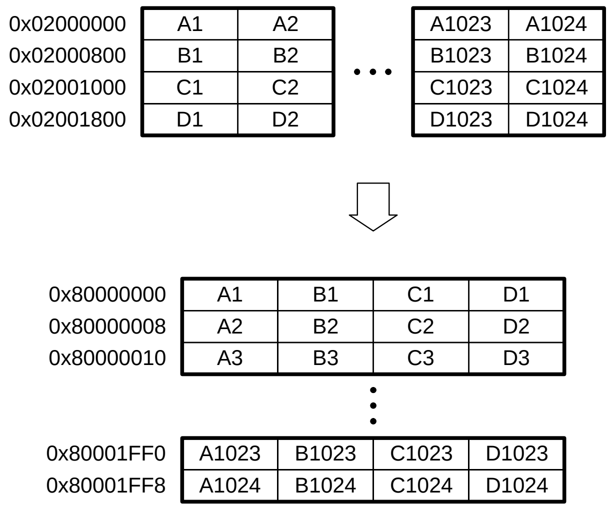

Figure 5. Data Sorting Example Diagram

The following formulas can be used to set up a DMA channel to organize the data in memory by ordinal position:

- FRAME INDEX should be set to –(((E – 1) x F) – 1) x S
- ELEMENT INDEX should be set to F x S, where
  - F = initial value of frame count
  - E = initial value of element count, as well as the element count reload value
  - S = element size in bytes

This example focuses on the second case mentioned above, in which equally sized data arrays are located in external memory. For this transfer to give the desired results, it is necessary that the arrays be the same size and reside in contiguous memory.

For this example, it is assumed that the 16-bit data is located in external RAM beginning at address 0x02000000 (CE2). The DMA channel is configured to bring four frames of 1k half-words from their locations in RAM to internal data memory beginning at 0x80000000.[2] The index values are:

- FRAME INDEX = –(((1024 – 1) x 4) – 1) x 2 = 0xE00A
- ELEMENT INDEX should be set to 4 x 2 = 8

For this, the following values must be assigned to the DMA channel's control registers:

Primary control register	=	0x000001D0
Source address register	=	0x02000000
Destination address register	=	0x80000000
Transfer counter register	=	0x00040400
Global count reload register A	=	0x00000400
Global index register A	=	0xE00A0008

The primary control register for channel 0 should be configured as follows:

DST RELOAD	=	00	INDEX	=	0
SRC RELOAD	=	00	CNT RELOAD	=	0
EMOD	=	0	SPLIT	=	00
FS	=	0	ESIZE	=	01
TCINT	=	0	DST DIR	=	11
PRI	=	0	SRC DIR	=	01
WSYNC	=	00000	STATUS	=	00
RSYNC	=	000	START	=	00
RSYNC	=	00			

[2] Note that if this transfer is performed on the TMS320C62x™, each array will be located in its own memory bank. This allows multiple arrays to be accessed during the same cycle with no contention. See the *TMS320C6000 Peripherals Reference Guide* (SPRU190) for details on the configuration of internal data memory.

TMS320C62x is a trademark of Texas Instruments.

The transfer counter register should be configured as follows:

FRAME COUNT	=	0x0004
ELEMENT COUNT	=	0x0400

The global index register A is shown in Figure 6.

31 16	15 0
FRAME INDEX	ELEMENT INDEX
R/W-0	R/W-0

Legend: R/W = Read/Write

Figure 6. Global Index Register A

The global index register A should be configured as follows:

FRAME INDEX	=	0xE00A
ELEMENT INDEX	=	0x0008

The setting of 01b in the SRC DIR bit field causes the DMA channel to increment source address by one element size (2 bytes in this example) following each element. Set DST DIR to 11b, to modify the destination address, according to global index register A (INDEX = 0). ELEMENT INDEX (value set to 0x0008) is used following each element within each frame, to increment the destination address by 8 bytes (4 elements). FRAME INDEX (value set to 0xE00A) is used following the last element of each frame, to set the destination address to the first element of the subsequent frame.

To initiate the transfer, a value of 01b must be written to the START bit field.

4 Servicing a Peripheral

In many applications, a DMA channel services a peripheral sending data to, and receiving data from, the device. The peripheral is often one of the following:

- Coder-decoder (Codec)
- AFE
- Analog interface chip (AIC)
- Analog-to-digital (A/D) converter
- Digital-to-analog (D/A) converter

These devices send samples to/from the C6000 through the McBSP or the EMIF. If the DMA is to effectively communicate with the peripheral, it must be configured to perform a synchronized data transfer. It must write only when the peripheral is able to accept new data, and read only when the peripheral has new data available.

Three types of synchronization are available to a DMA channel:

- Read synchronization – Each read transfer waits for the selected event to occur before proceeding.
- Write synchronization – Each write transfer waits for the selected event to occur before proceeding.

- Frame synchronization – Each frame transfer waits for the selected event to occur before proceeding.

Each DMA channel can be configured for read synchronization, write synchronization, both read and write synchronization, or frame synchronization. If frame synchronization is used, the read-synchronization event triggers the transfer of an entire frame. The event to which the transfer is synchronized is set in the DMA channel’s primary control register. Table 1 gives a complete list of DMA synchronization events.

Table 1. DMA Channel Synchronization Events

Event Number (Binary)	Event Acronym	Event Description
00000	None	No synchronization
00001	TINT0	Timer 0 interrupt
00010	TINT1	Timer 1 interrupt
00011	SD_INT	EMIF SDRAM timer interrupt
00100	EXT_INT4	External interrupt pin 4
00101	EXT_INT5	External interrupt pin 5
00110	EXT_INT6	External interrupt pin 6
00111	EXT_INT7	External interrupt pin 7
01000	DMA_INT0	DMA channel 0 interrupt
01001	DMA_INT1	DMA channel 1 interrupt
01010	DMA_INT2	DMA channel 2 interrupt
01011	DMA_INT3	DMA channel 3 interrupt
01100	XEVT0	MCSP 0 transmit event
01101	REVT0	MCSP 0 receive event
01110	XEVT1	MCSP 1 transmit event
01111	REVT1	MCSP 1 receive event
10000	DSPINT	Host to DSP interrupt

The transmit-and-receive events to the DMA from each McBSP are XEVT and REVT, respectively. XEVT is issued when a value has been copied from the data transmit register (DXR) to the transmit shift register (XSR), signifying that the most recent data has been transferred out. REVT is issued when a value has been copied from the receive buffer register (RBR) to the data receive register (DRR), indicating that a new data value has been received. An external peripheral on the EMIF typically has similar synchronization events arriving through one or more of the external interrupt pins (EXT_INT[7–4]).

In addition to properly synchronizing the peripherals, you must ensure that the data is transferred to and from the correct location. This becomes an issue when performing transfers for 8- and 16-bit elements, particularly when operating in an endian mode that is different than the peripheral expects. (See section 4 for more details.)

4.1 Synchronized Data Transfer Example

To transfer data to and from a McBSP,[3] it is necessary to use read synchronization to read the DRR, and write synchronization to write to the DXR.

Consider a variation of the block move example. Once again, it is desired to bring a 1k block of 32-bit elements into data memory, beginning at address 0x80000000. Instead of bringing the data from an external memory space, however, it arrives through McBSP0. Each time a new 32-bit data value arrives in the McBSP0 DRR register, the event REVT0 is set. The DMA channel servicing this data must have its read transfers synchronized on this event (RSYNC = 01101b). Figure 7 shows the transfer that the above setup performs.

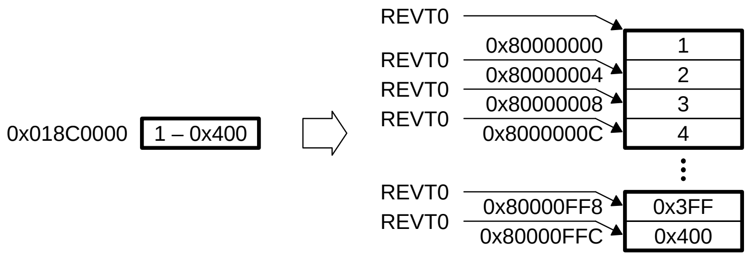

Figure 7. Synchronized Data Transfer Example Diagram

To initialize this transfer, set the four control registers to the following values:

Primary control register	= 0x00034040
Source address register	= 0x018C0000
Destination address register	= 0x80000000
Transfer counter register	= 0x00000400

The primary control register should be configured as follows:

DST RELOAD	= 00	INDEX	= 0
SRC RELOAD	= 00	CNT RELOAD	= 0
EMOD	= 0	SPLIT	= 00
FS	= 0	ESIZE	= 00
TCINT	= 0	DST DIR	= 01
PRI	= 0	SRC DIR	= 00
WSYNC	= 00000	STATUS	= 00
RSYNC	= 011	START	= 00
RSYNC	= 01		

Setting 01b in the DST DIR bit field configures the DMA channel to increment the destination address by one element size (4 bytes in this example) following each element. Since the source is a fixed address, SRC DIR is set to 00b. RSYNC is set to REVT0, to synchronize the reading of the DRR.

To initiate the transfer, write a 01b to the START bit field.

[3] This information is valid for servicing an external AFE as well. The more frequent the accesses to the AFE, however, the more favorable a frame-synchronized transfer solution is. It is usually important to keep the arbitration within the EMIF to a minimum.

4.2 Split-Mode Transfer Example

A McBSP or AFE is more commonly used for bidirectional communication with the device, which means that the DMA needs to both read and write to the peripheral. Adding to the previous example (see section 4.1), consider that, at the same time, 1k words are received from the McBSP, and 1k words are transmitted to it as well. To facilitate this, a channel must be set up to transfer data from internal memory to McBSP0 DXR. This channel must be write-synchronized on the event , XEVT0.

Although this could easily be done with a second DMA channel, one of the features of the DMA is that a single channel can be used to service both the input and output data streams of a peripheral for which the transmit- and receive-data addresses are fixed. Using this feature, the synchronized data transfer example, described in the previous section, could be modified so that a single DMA channel is used to perform both reads from, and writes to, the McBSP. A diagram of a split-mode transfer is shown in Figure 8.

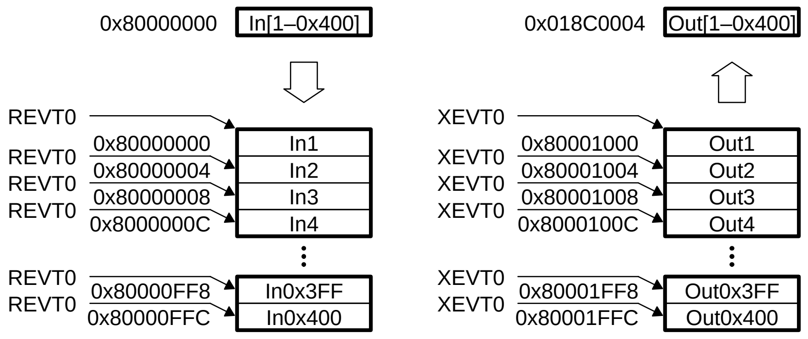

Figure 8. Split-Mode Transfer Example Diagram

Setting the SPLIT bit field in the DMA channel primary control register enables a split-mode transfer and selects the location of the split address. Possible SPLIT values are listed in Table 2.

Table 2. DMA Channel SPLIT Settings

SPLIT Value	Split Address
00	Split mode disabled
01	DMA global address register A
10	DMA global address register B
11	DMA global address register C

Global address registers A, B, and C can be used to hold the split address. This address is assumed to be on an even word boundary, as the three LSBs are reserved and fixed at zero. This address is used as the split source address. The split destination address is automatically set to be one word address greater than the split source address. If an external peripheral is to be serviced by a DMA channel in split-mode, this addressing convention must be followed.

In this example, a block of 1k 32-bit words is transferred to McBSP0, and a block of 1k 32-bit words is transferred from McBSP0 to memory using the same DMA channel.[4] The 1k data block to be transferred to the McBSP begins at address 0x80001000, while the input data block is again written to 0x80000000. For this, the following values must be assigned to the DMA registers:

Primary control register	= 0x00634450
Source address register	= 0x80001000
Destination address register	= 0x80000000
Transfer counter register	= 0x00000400
Global address register A	= 0x018C0000

The primary control register should be configured as follows:

DST RELOAD	= 00	INDEX	= 0
SRC RELOAD	= 00	CNT RELOAD	= 0
EMOD	= 0	SPLIT	= 01
FS	= 0	ESIZE	= 00
TCINT	= 0	DST DIR	= 01
PRI	= 0	SRC DIR	= 01
WSYNC	= 01100	STATUS	= 00
RSYNC	= 011	START	= 00
RSYNC	= 01		

The setting of 01b in the SRC DIR and DST DIR bit fields causes the DMA channel to increment both the source address and the destination address by one element size (4 bytes in this example) following each element. Since this is a split transfer, the elements from the source address are written to the split destination address (DXR) and the elements transferred from the split source address (DRR) are written to the destination address. RSYNC is set to REVT0, and WSYNC is set to XEVT0.

To initiate the transfer, write a 01b to the START bit field. See Figure 2.

4.3 Frame-Synchronized Data Transfer Example

If accesses to an external AFE are frequent, it can be beneficial to transfer elements in bursts, rather than making single accesses through the EMIF. Bursting makes more efficient use of the EMIF, as there are fewer cycles lost to arbitration between requesters. To facilitate bursting, it can be necessary to have an intermediate first in, first out (FIFO) as part of the AFE subsystem. By using a FIFO, there is still a Data-In address and a Data-Out address from the perspective of the DMA. The synchronization event is an external signal from the FIFO (or external control logic), indicating when the FIFO has sufficient data to burst a frame.

To perform a synchronized burst, the DMA channel must be configured with frame synchronization. The FS bit field must equal 1, and the RSYNC[5] bit field should be set to the desired synchronization event, both in the primary control register.

[4] Note that the DXR address is the next adjacent memory address above the DRR.

[5] Usually, an external interrupt or a timer interrupt synchronizes the DMA channel, depending on whether the data transfer is internally or externally mastered.

Consider a modification of the split-mode transfer example with an external AFE as the peripheral (see section 4.2). If the elements arrive and depart through the AFE at a rate that does not seriously limit the bandwidth of the EMIF, the only modification to the previous setup is to replace the global address register A value with the AFE Data-In address.[6]

If, however, servicing transmit and receive elements individually prevent the EMIF from allowing further accesses (either by the CPU or another DMA channel), the bursting method can be used. When performing frame synchronized transfers, two DMA channels must be used, as split-mode transfers do not allow bursting.

For this example system, assume there is an input FIFO and an output FIFO, each capable of holding 1k 32-bit elements (one frame size). An external interrupt (EXT_INT4) selects when the frame of data is ready to be read from the input FIFO. The output FIFO is written to as soon as the input frame is completed. In this fashion, the input and output transfer rates are identical.[7] Figure 9 shows the transfers to and from external FIFOs.

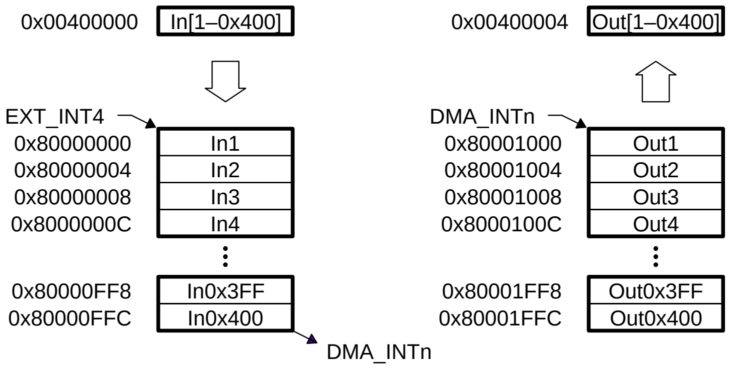

Figure 9. Frame-Synchronized Transfer Example Diagram

The AFE is mapped into CE0 space (Map 1), with the address of the input FIFO at 0x00400000, and the address of the output FIFO at 0x00400004.

For the channel servicing the input data, the following values must be assigned to the DMA registers:

Primary control register	= 0x06010040
Secondary control register	= 0x00000008
Source address register	= 0x00400000
Destination address register	= 0x80000000
Transfer counter register	= 0x00010400

[6] This assumes that the Data Out address is one word size above the Data In address.

[7] The input data transfer and the output data transfer can both have external synchronization as well.

The primary control register should be configured as follows:

DST RELOAD	= 00	INDEX	= 0
SRC RELOAD	= 00	CNT RELOAD	= 0
EMOD	= 0	SPLIT	= 00
FS	= 1	ESIZE	= 00
TCINT	= 1	DST DIR	= 01
PRI	= 0	SRC DIR	= 00
WSYNC	= 00000	STATUS	= 00
RSYNC	= 001	START	= 00
RSYNC	= 00		

The secondary control register is shown in Figure 10.

31–24
Reserved
R/W-0

23	22	21	20	19	18	17	16
Reserved		WSPOL	RSPOL	FSIG	DMAC		
R/W-0		R/W-0	R/W-0	R/W-0	R/W-0		

15	14	13	12	11	10	9	8
WYSYNC CLR	WYSYNC STAT	RSYNC CLR	RSYNC STAT	WDROP IE	WDROP COND	RDROP IE	RDROP COND
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
BLOCK IE	BLOCK COND	LAST IE	LAST COND	FRAME IE	FRAMD COND	SX IE	SX COND
R/W-1	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0

Legend: R/W = Read/Write

Figure 10. Secondary Control Register

The secondary control register should be configured as follows:

DMAC	= 000	BLOCK IE	= 0
WSYNC CLR	= 0	BLOCK COND	= 0
WSYNC STAT	= 0	LAST IE	= 0
RSYNC CLR	= 0	LAST COND	= 0
RSYNC STAT	= 0	FRAME IE	= 1
WDROP IE	= 0	FRAME COND	= 0
WDROP COND	= 0	SX IE	= 0
RDROP IE	= 0	SX COND	= 0
RDROP COND	= 0		

The transfer counter register should be configured as follows:

FRAME COUNT	= 0x0001
ELEMENT COUNT	= 0x0400

The settings of 01b in the DST DIR bit field cause the DMA channel to increment the destination address by one element size (4 bytes in this example) following each element. Since this is a frame-synchronized transfer (FS = 1), the entire frame of elements from the source address are read from the AFE as soon as the read synchronization event (RSYNC = EXT_INT4) is received. Setting TCINT to 1 causes the DMA channel to generate an interrupt that occurs at the end of a frame (FRAME IE = 1). This interrupt initiates the transfer to the output FIFO by another DMA channel.

For the DMA channel servicing the output data, the following values must be assigned to the DMA registers:

Primary control register	= 0x0402X010[8]
Source address register	= 0x80001000
Destination address register	= 0x00400004
Transfer counter register	= 0x00010400

The primary control register should be configured as follows:

DST RELOAD	= 00	INDEX	= 0
SRC RELOAD	= 00	CNT RELOAD	= 0
EMOD	= 0	SPLIT	= 00
FS	= 1	ESIZE	= 00
TCINT	= 0	DST DIR	= 00
PRI	= 0	SRC DIR	= 01
WSYNC	= 00000	STATUS	= 00
RSYNC	= 010	START	= 00
RSYNC	= {*n*}[9]		

The transfer counter register should be configured as follows:

FRAME COUNT	= 0x0001
ELEMENT COUNT	= 0x0400

The settings of 01b in the SRC DIR bit field causes the DMA channel to increment the source address by one element size (4 bytes in this example) following each element. Since this is a frame synchronized transfer (FS = 1), the entire frame of elements from the source address is written to the AFE as soon as the read synchronization event (RSYNC = DMA_INT*n*) is received.

To initiate the two transfers, write a value of 01b to the START bit field of each channel's primary control register.

[8] The value of X is equal to 4**n*, where *n* is the DMA channel number servicing the input data. See note 9.

[9] The value of RSYNC for this channel depends on the channel servicing the input transfer, where *n* equals the DMA channel number.

4.4 Endian Mode Considerations

Endianness plays an important role when using a peripheral for element sizes other than 32 bits. This is usually only true for the McBSPs, as external peripherals typically match the endianness of the entire system. In a system that operates in little endian, external peripherals should have the LSB located at ED0. In big-endian systems, the most significant bit (MSB) should always be located at ED31.[10] However, if a non-32-bit peripheral is connected to the EMIF with the LSB at ED0 for a big-endian system, the following directly applies.

The McBSPs are easiest to picture as a physical register set with a right and left side. The right side corresponds to data bit 0, and the left side corresponds to bit 31. By default the McBSP assumes that the LSB of the element transferred is located in bit 0. Data is always transmitted out from the right side of the DXR.

By default, data is received from the right side of the DRR with the LSB of the element at bit 0. The data justification of received data is programmable to be left-justified as well. It is possible to configure the McBSP such that received data always has the MSB at bit 31.

The DXR and DRR (in its default state) hold the LSB in the rightmost position and the MSB on the left side (actual bit location depends on the element size to be transferred). The byte that accesses the rightmost location of the McBSP registers depends on which endian mode the DSP is in. The DXR and DRR of the McBSPs are depicted in Figure 11 and Figure 12, with the byte ordering for each endian mode shown.

	31			0
Little Endian	Byte 3	Byte 2	Byte 1	Byte 0
Big Endian	Byte 0	Byte 1	Byte 2	Byte 3
	MS		MS	MS LS

Figure 11. DXR Byte Locations

	31			0
Little Endian	Byte 3	Byte 2	Byte 1	Byte 0
Big Endian	Byte 0	Byte 1	Byte 2	Byte 3
	MS		MS	MS LS

Figure 12. DRR Byte Locations

When in little-endian mode, the rightmost data location is the base address of the DXR, so no matter what the element size, a write to the DXR base address properly aligns the element; however, this is the upper portion of the register in big-endian mode. Depending on the size of the element, the write must be made to the address of either Byte 0 (32-bit), Byte 2 (16-bit), or to Byte 3 (8-bit). It is not possible to left-justify outgoing data.

The DRR is configurable to either right- or left-justify the incoming data. The justification determines the source address of the data element. For right-justified data (default), the source address is always Byte 0 in little-endian mode. It is in Byte 0 (32-bit), Byte 2 (16-bit), or Byte 3 (8-bit) in big-endian mode. For left-justified data, the reverse is true.

[10] The MSB should be either bit 31, 15, or 7 only. If a peripheral that is not 32, 16, or 8 bits wide is used, the upper data bits should be unconnected.

Table 3 shows the possible endian mode, element size, and DRR justification combinations that can be encountered in a system. Only the source and destination addresses are given for each. All of the necessary configurations described previously still apply.

Table 3. Possible DMA Source and Destination Address for Servicing McBSP0[11,12,13]

Element Size	Endian Mode	DRR Justification	Source Address	Destination Address
8 bits	Little	Right	0x018C0000	0x018C0004
		Left	0x018C0003	0x018C0004
	Big	Right	0x018C0003	0x018C0007
		Left	0x018C0000	0x018C0007
12 bits 16 bits	Little	Right	0x018C0000	0x018C0004
		Left	0x018C0002	0x018C0004
	Big	Right	0x018C0002	0x018C0006
		Left	0x018C0000	0x018C0006
20 bits 24 bits 32 bits	Little	Right	0x018C0000	0x018C0004
		Left	0x018C0000	0x018C0004
	Big	Right	0x018C0000	0x018C0004
		Left	0x018C0000	0x018C0004

5 Repetitive DMA Operation

When data flow requires that a peripheral be repetitively serviced throughout device operation, or when program sections are to be continuously swapped in and out of program memory, it is appropriate to configure the DMA to automatically reprogram for subsequent transfers. This capability can be realized by running the DMA channel in auto-initialization mode, and providing reload values for the source and destination addresses and the transfer counter.

Once the DMA is programmed to run repetitively, an important concern is how to effectively buffer the incoming and outgoing data. To keep memory usage to a minimum, the DMA should write over old data. This type of buffering is considered “circular”, as data is continuously cycling through the same memory space.

When a high throughput is required of the device and time cannot be spared waiting on the DMA to transfer a frame of data to the device before processing, a ping-pong buffering system should be used. This scheme requires slightly more complicated reload settings, but allows the CPU to process data while the DMA transfers new data on-chip and old data off-chip.

The following sections illustrate these two buffering schemes.

[11] Note that the source addresses and destination addresses are identical for both the big- and little-endian modes when transferring 32-bit elements.

[12] 12-bit data should be accessed as a 16-bit element.

[13] 20- and 24-bit data should be accessed as 32-bit elements.

5.1 Transferring Data To and From Circular Buffers

In many DSP applications, data is stored in on-chip circular buffers, in which new data is written directly over old data so that a minimum amount of memory space is consumed by an application. This is shown in Figure 13. For such an application, it is desirable for the DMA to continually bring frames of data to the same block of data memory. This capability can be implemented by configuring the DMA channel(s) for the initial block move, and taking a few extra steps to allow the DMA to reset itself after each transfer.

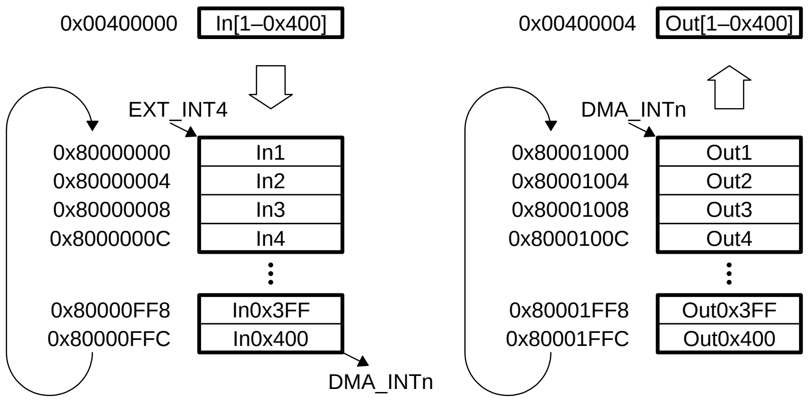

Figure 13. Circular Buffer Example Diagram

As an example of how to use circular buffering, the frame-synchronized data transfer method, described in section 4.3, can be modified to run continuously, with the input and output data buffers reused for each frame of data. The address of the input FIFO is 0x00400000. The address of the output FIFO is 0x00400004.

For the DMA channel that services the input data, the primary control register value must be modified to allow the channel to use an index for the destination address. Global index register A is used to return the destination address to the beginning of the frame. The control registers for this channel should be:

Primary control register	=	0x060100C0
Secondary control register	=	0x00000008
Source address register	=	0x00400000
Destination address register	=	0x80000000
Transfer counter register	=	0x00010400
Global index register A	=	0xF0040004

The primary control register should be configured as follows:

DST RELOAD	=	00	INDEX	=	0
SRC RELOAD	=	00	CNT RELOAD	=	0
EMOD	=	0	SPLIT	=	00
FS	=	1	ESIZE	=	00
TCINT	=	1	DST DIR	=	11
PRI	=	0	SRC DIR	=	00
WSYNC	=	00000	STATUS	=	00
RSYNC	=	001	START	=	00
RSYNC	=	00			

The secondary control register should be configured as follows:

DMAC	= 000	BLOCK IE	= 0
WSYNC CLR	= 0	BLOCK COND	= 0
WSYNC STAT	= 0	LAST IE	= 0
RSYNC CLR	= 0	LAST COND	= 0
RSYNC STAT	= 0	FRAME IE	= 1
WDROP IE	= 0	FRAME COND	= 0
WDROP COND	= 0	SX IE	= 0
RDROP IE	= 0	SX COND	= 0
RDROP COND	= 0		

The transfer counter register should be configured as follows:

FRAME COUNT	= 0x0001
ELEMENT COUNT	= 0x0400

The global index register A should be configured as follows:

FRAME INDEX	= 0xF004
ELEMENT INDEX	= 0x0004

The settings of 11b in the DST DIR bit field causes the DMA channel to modify the destination address using global index register A (INDEX = 0). Following each element the address is modified by four bytes using ELEMENT INDEX, and after each frame the destination address returns to 0x80000000 using FRAME INDEX. Since this is a frame synchronized transfer (FS = 1), the entire frame of elements from the source address is read from the AFE as soon as the read-synchronization event (RSYNC = EXT_INT4) is received. Setting TCINT to 1 causes the DMA channel to generate an interrupt that occurs at the end of a frame (FRAME IE = 1). This interrupt is used to initiate the transfer to the output FIFO by another DMA channel.[14]

For the DMA channel servicing the output data, the primary control register value must be modified to allow the channel to use an index for the source address. Global index register A is used to return the source address to the beginning of the frame. The control registers for this channel should be:

Primary control register	= 0x0402X030[15]
Source address register	= 0x80001000
Destination address register	= 0x00400004
Transfer counter register	= 0x00010400

[14] Note that the FRAME COND bit must be manually cleared following each frame transfer by this channel. See section 6 for information on how to do this.

[15] The value of X is equal to 4**n*, where *n* is the DMA channel number servicing the input data. See note 16.

The primary control register should be configured as follows:

DST RELOAD	= 00	INDEX	= 0
SRC RELOAD	= 00	CNT RELOAD	= 0
EMOD	= 0	SPLIT	= 00
FS	= 1	ESIZE	= 00
TCINT	= 0	DST DIR	= 00
PRI	= 0	SRC DIR	= 11
WSYNC	= 00000	STATUS	= 00
RSYNC	= 010	START	= 00
RSYNC	= {*n*}[16]		

The transfer counter register should be configured as follows:

FRAME COUNT	= 0x0001
ELEMENT COUNT	= 0x0400

The settings of 11b in the SRC DIR bit field cause the DMA channel to modify the source address using global index register A (INDEX = 0). Following each element, the address is modified by four bytes using ELEMENT INDEX, and after each frame the destination address returns to 0x80001000 using FRAME INDEX. Since this is a frame-synchronized transfer (FS = 1), the entire frame of elements from the source address is written to the AFE as soon as the read-synchronization event (RSYNC = DMA_INT*n*) is received.

To initiate the two transfers, a value of 11b must be written to the START bit field of each channel's primary control register. Since the output buffer does not have valid data in it until after the CPU has processed the initial input buffer, the DMA channel servicing the output should not be started until after the first frame completes.

5.2 Ping-Pong Transfer Example

The circular buffering example provides an easy way to renew the buffer of data being operated on by the CPU (see section 5.1). One drawback associated with the single pair of input and output buffers is that, while the DMA is filling the input buffer or reading from the output buffer, the CPU cannot access the same space.[17]

A dual-buffering scheme is one way to ensure that the CPU does not operate on an incorrect set of data or change data that has not yet been moved off-chip. This means simply that there are two input buffers and two output buffers. This doubles the amount of internal memory consumed, but greatly increases throughput for applications to which the CPU is dedicated to, converting the input data set to an output data set with few breaks. The benefit of this is improved greatly when the ping and pong buffers are in different data memory blocks, since there is no contention between the CPU and DMA when they are accessing different blocks.

For this type of application, the DMA moves data to and from one pair of input/output buffers, while the CPU operates on the other pair. As soon as both the CPU and DMA are finished, they switch input/output buffer pairs, as shown in Figure 14.

[16] The value of RSYNC for this channel depends on the channel servicing the input transfer, where *n* equals the DMA channel number.

[17] This is true unless care is taken that the CPU is always ahead of the input data and behind the output data.

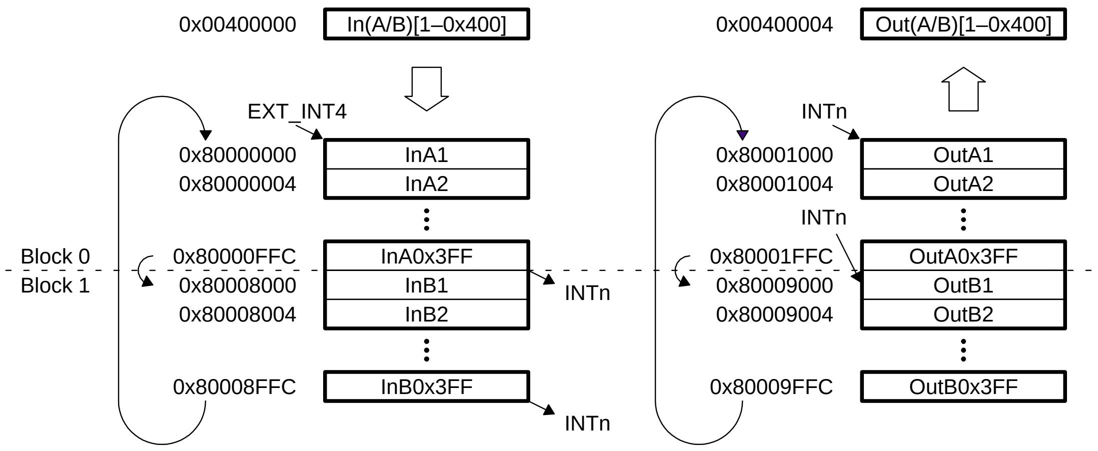

Figure 14. Ping-Pong Buffer Example Diagram

Consider a variation of the previous transfer, in which there are two 1k-word-input buffers and two 1k-word output buffers in data memory. The DMA transfers 1k block of 32-bit words to one input buffer, located at 0x80000000, and a 1k block of 32-bit words from one output buffer, located at 0x80001000. After these sets of data are transferred, a new block is transferred to a second input buffer at 0x80008000, and a new block is transferred from a second output buffer at 0x80009000. The next pair of transfers returns to the original input/output pair. The CPU computes data located in the first input buffer, and stores the results in the first output buffer, following the first DMA transfer.

Once completed, the CPU uses the second input buffer, storing the results in the second output buffer. The CPU then switches back to the first pair and continues. The control registers from the previous example are modified such that the source and destination addresses are reloaded to their original values following each block (two frames) of data transferred.

For the DMA channel that services the input data, the primary control register value must be modified to allow the channel to post-increment the destination address. Global address register B is used to return the destination address to the beginning of the first buffer. The control registers for this channel should be:

Primary control register	= 0x460100C0
Secondary control register	= 0x00000088
Source address register	= 0x00400000
Destination address register	= 0x80000000
Transfer counter register	= 0x00020400
Global index register A	= 0x70040004
Global address register B	= 0x80000000

The primary control register should be configured as follows:

DST RELOAD	= 01	INDEX	= 0
SRC RELOAD	= 00	CNT RELOAD	= 0
EMOD	= 0	SPLIT	= 00
FS	= 1	ESIZE	= 00
TCINT	= 1	DST DIR	= 11
PRI	= 0	SRC DIR	= 00
WSYNC	= 00000	STATUS	= 00
RSYNC	= 001	START	= 00
RSYNC	= 00		

The secondary control register should be configured as follows:

DMAC	= 000	BLOCK IE	= 1
WSYNC CLR	= 0	BLOCK COND	= 0
WSYNC STAT	= 0	LAST IE	= 0
RSYNC CLR	= 0	LAST COND	= 0
RSYNC STAT	= 0	FRAME IE	= 1
WDROP IE	= 0	FRAME COND	= 0
WDROP COND	= 0	SX IE	= 0
RDROP IE	= 0	SX COND	= 0
RDROP COND	= 0		

The transfer counter register should be configured as follows:

FRAME COUNT	= 0x0002
ELEMENT COUNT	= 0x0400

The setting of 11b in the DST DIR bit field causes the DMA channel to modify the destination address by the index selected by the INDEX field following each element. Since global index register A is selected, the address is modified by one word size (4 bytes) following each element, and jumps to the beginning of the second data memory block, following the last element of the frame.

Since this is a frame-synchronized transfer (FS = 1), an entire frame of elements is read from the AFE as soon as the read-synchronization event (RSYNC = EXT_INT4) is received. Setting TCINT to 1 causes the DMA channel to generate an interrupt that occurs at the end of a frame (FRAME IE = 1). This interrupt is used to initiate the transfer to the output FIFO by another DMA channel.[18] Following the second frame of each block, the destination address is reloaded to 0x80000000, to allow the DMA to overwrite old data with new data.

For the DMA channel that services the output data, the primary control register value must be modified to allow the channel to post-increment the source address. Global address register C is used to return the source address to the beginning of the first output buffer. The control registers for this channel should be:

[18] Note that the FRAME COND bit must be manually cleared following each frame transfer by this channel. See section 6 for information on how to do this.

Primary control register	=	0x2402X030[19]
Source address register	=	0x80002000
Destination address register	=	0x00400004
Transfer counter register	=	0x00020400
Global address register C	=	0x80002000

The primary control register should be configured as follows:

DST RELOAD	=	00	INDEX	=	0
SRC RELOAD	=	10	CNT RELOAD	=	0
EMOD	=	0	SPLIT	=	00
FS	=	1	ESIZE	=	00
TCINT	=	0	DST DIR	=	00
PRI	=	0	SRC DIR	=	11
WSYNC	=	00000	STATUS	=	00
RSYNC	=	010	START	=	00
RSYNC	=	{*n*}[20]			

The transfer counter register should be configured as follows:

FRAME COUNT	=	0x0002
ELEMENT COUNT	=	0x0400

The setting of 11b in the SRC DIR bit field causes the DMA channel to modify the source address following each element, also with global index register A.

Since this is a frame-synchronized transfer (FS = 1), an entire frame of elements is written to the AFE as soon as the read-synchronization event (RSYNC = DMA_INT*n*) is received. Following the second frame of each block, the source address is reloaded to 0x80001000.

To initiate the two transfers, a value of 11b must be written to the START bit field of each channel's primary control register. Since the output buffers do not have valid data in them until after the CPU has processed the initial two input buffers, the DMA channel servicing the output should not be started until after the first block completes. By enabling the BLOCK IE bit in the secondary control register of the DMA channel servicing the input data, the DMA channel servicing the output data can be started following the first input block (two frames).[21]

[19] The value of X is equal to 4**n*, where *n* is the DMA channel number servicing the input data. See note 20.

[20] The value of RSYNC for this channel depends on the channel servicing the input transfer, where *n* equals the DMA channel number.

[21] An example interrupt service routine that does this, and clears the FRAME COND bit, is given in section 6.

5.3 Program Paging Example

The methodology behind the ping-pong data transfer can also be applied to program paging. When a program requires more than 64KB of memory, and it is not desired to operate either from external memory or in cache mode, it becomes necessary to implement program paging. Programming a DMA channel to transfer a block of code from external memory to internal program memory does this function. To maintain efficiency with the CPU, it is necessary to have at least two sections of program space in the internal program memory. By doing so, existing code can be executed while new code is brought to the device.

To facilitate an efficient paging scheme, a program should be divided into sections, with the sections to be paged occupying the same amount of memory space.[22] These sections should then be stored in known external memory locations so that the DMA is capable of retrieving specific blocks of code. A typical breakdown of code includes:

- Interrupt Service Table, which resides in either program memory or external memory.[23]
- Main block of code, which resides in program memory
- Program pages, which reside in external memory

Bringing data into internal program memory with the DMA differs from accessing internal data memory because the CPU always has priority over the DMA for accesses. This results in transfers to program memory taking longer than those to data memory. One exception to this is the C6202, which has two program memory blocks. Since there is no contention between the CPU and DMA accesses when accessing separate blocks, the DMA transfer is able to complete quickly.

The primary factor in enabling a program page to be transferred more effectively when the CPU and DMA are accessing the same memory block is code parallelism. The less parallel a program is, the more frequent the accesses to the internal program memory space by the DMA can be. Any optimized loop in a program slows the transfer.

The sample program in this example includes the following sections:

- Interrupt Service Table (1k), linked to the base address of internal program memory: 0x00000000 (Map 1). This section contains interrupt.
- Main block of code (15k), linked to 0x00000400 in internal program memory. This section contains main subroutine.
- Initialization section linked to 0x0000A000. This block of code sets up the EMIF, interrupts, data initialization, and any DMA transfers to be performed during the program that can be configured ahead of time. This section is overwritten as it is only used once.
- Page 1 section linked to external memory location 0x20000000 (CE2), with run-time location set to 0x00006000. This page of code is transferred into internal program memory multiple times throughout the program execution.

[22] For information on how to create program sections and organize them in memory, see the *TMS320C6000 Assembly Language Tools User’s Guide* (SPRU186) and the *TMS320C6000 Optimizing C Compiler User’s Guide* (SPRU187).

[23] The location of the Interrupt Vector Table should depend on the frequency of interrupts that need to be serviced. If interrupts are frequent, it is more efficient for the IST to be in internal program memory.

- Page 2 section linked to 0x20004000, with run-time location set to 0x0000A000. This page of code is transferred into internal program memory multiple times throughout the program execution.
- Page 3 section linked to 0x20008000, with run-time location set to 0x00006000. This page of code is transferred into internal program memory multiple times throughout the program execution.
- Page 4 section linked to 0x2000C000, with run-time location set to 0x0000A000. This page of code is brought into internal program memory multiple times throughout the program execution.

Each page (1–4) listed above is of length 16k (0x4000), and each page branches to the next sequentially. To facilitate this, a DMA channel should be set up by the initialization code to transfer pages 1 through 4 to their run-time program space. Figure 15 shows a block diagram of this four-page system.

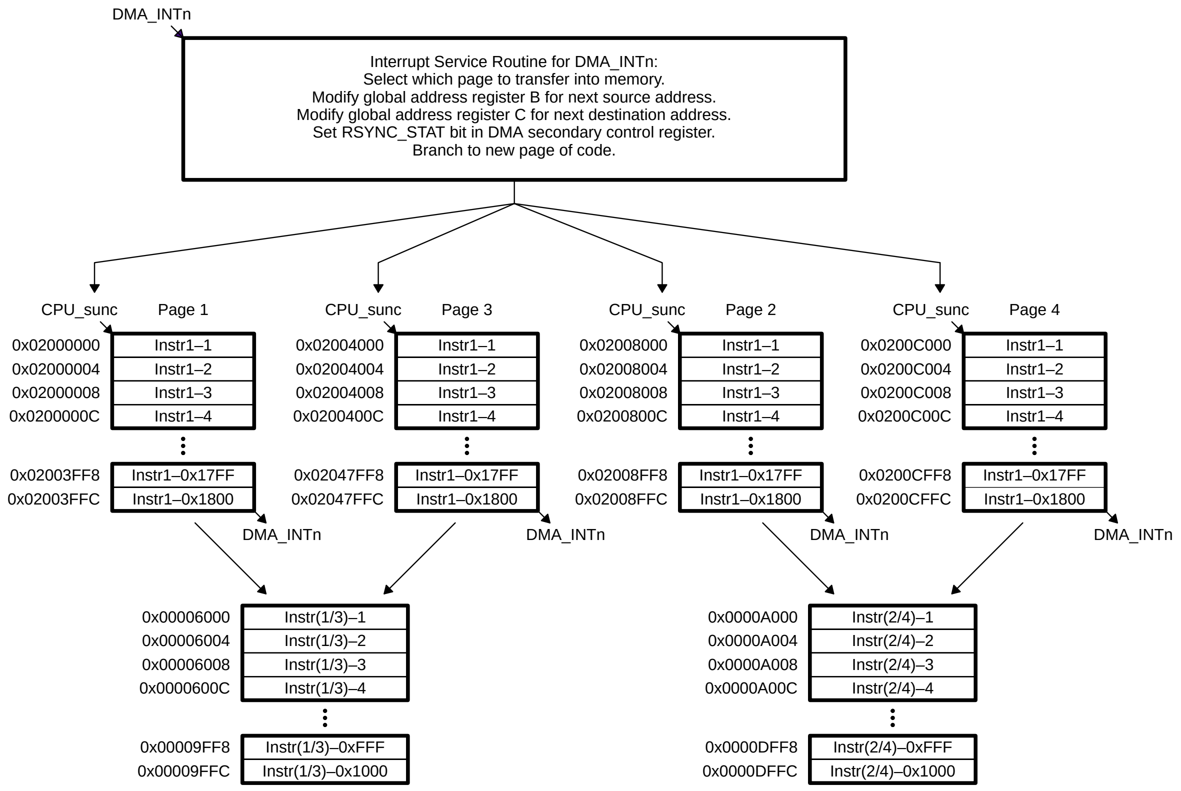

Figure 15. Program Paging Example Diagram

Program paging is typically a background transfer, as it is not desirable to interfere with the servicing of peripherals or other data transfers. Paging is normally done using a low-priority channel.

For this example, the DMA channel is set up as follows:

Primary control register	= 0x9601A050
Secondary control register	= 0x00000080
Source address register	= 0x20000000
Destination address register	= 0x00006000
Transfer counter register	= 0x00011000
Global address register B	= 0x20006000
Global address register C	= 0x0000A000
Global reload register A	= 0x00011000

The primary control register should be configured as follows:

DST RELOAD	= 10	INDEX	= 1
SRC RELOAD	= 01	CNT RELOAD	= 0
EMOD	= 0	SPLIT	= 00
FS	= 1	ESIZE	= 00
TCINT	= 1	DST DIR	= 01
PRI	= 0	SRC DIR	= 01
WSYNC	= 00000	STATUS	= 00
RSYNC	= 001	START	= 00
RSYNC	= 10		

The secondary control register should be configured as follows:

DMAC	= 000	BLOCK IE	= 1
WSYNC CLR	= 0	BLOCK COND	= 0
WSYNC STAT	= 0	LAST IE	= 0
RSYNC CLR	= 0	LAST COND	= 0
RSYNC STAT	= 0	FRAME IE	= 0
WDROP IE	= 0	FRAME COND	= 0
WDROP COND	= 0	SX IE	= 0
RDROP IE	= 0	SX COND	= 0
RDROP COND	= 0		

The transfer counter register should be configured as follows:

FRAME COUNT	= 0x0001
ELEMENT COUNT	= 0x1800

The setting of 01b in the DST DIR and SRC DIR bit fields causes the DMA channel to increment the destination and source addresses following each element. Since this is a frame-synchronized transfer (FS = 1), an entire page is transferred to program memory as soon as the read-synchronization event (RSYNC = EXT_INT6) is received. Setting TCINT to 1 causes the DMA channel to generate an interrupt that occurs at the end of a block (BLOCK_IE = 1). This interrupt is used to let the CPU know that valid code is present.

Following each block, global reload registers should be set with the destination and source addresses for the subsequent transfer. External interrupt EXT_INT6 should not actually be used during program execution. Instead, the CPU should directly set the RSYNC STAT bit in the channel's secondary control register to initiate each transfer. This provides control between the CPU and the DMA. Each time the CPU finishes executing a page, it should set the RSYNC STAT bit, then poll the interrupt flag for the interrupt number of the DMA channel. If it is known that the CPU completes before the DMA every time, the CPU can be placed in IDLE to decrease the transfer time.

6 DMA Interrupt Service Routines

Through configuration of a DMA channel's primary and secondary control register, each DMA channel can interrupt the CPU when one or more conditions occur. When any of the enabled conditions occur, the interrupt flag for the DMA channel is set. If this interrupt is enabled in the interrupt enable register (IER), the interrupt is serviced. The conditions that can be used are given in Table 4.

Table 4. DMA Channel Condition Descriptions

Bit Field	Event	Occurs…
BLOCK	Block transfer complete	After the last write transfer in a block transfer is written to memory.
FRAME	Frame complete	After the last write transfer in each frame is written to memory.
LAST	Last frame	After all counter adjustments for the next-to-last frame in a block transfer complete.
WDROP RDROP	Dropped read/write synchronization	If a subsequent synchronization event occurs before the last one is cleared.
SX	Split transmit overrun receive	If the split-mode is enabled, and transmit-element transfers get seven or more element transfers ahead of receive-element transfers.

The IE bits in the channel's secondary control register must be set for each condition to generate an interrupt to the CPU. The TCINT bit in the channel's primary control register must also be set. This causes an interrupt to occur whenever the enabled condition transitions from a 0 to a 1, which is reported in the COND bit fields of the channel's secondary control register.

If the IE bit for a condition is enabled, the CPU must manually clear the COND bit to receive subsequent interrupts. This feature avoids confusion in the case that multiple events trigger the same interrupt. The most common way to perform this is to have an interrupt service routine (ISR) that services each DMA channel in use.[24] ISRs range in function from simplistic to complex, depending on the application. They are typically designed to be as short as possible so that little time is taken away from the processor.

[24] For information on how to set up CPU interrupts, see the *TMS320C6000 Peripherals Reference Guide* (SPRU190) and the *TMS320C6000 CPU and Instruction Set Reference Guide* (SPRU189).

To enable a DMA-generated interrupt to be taken, several steps must be taken by the CPU. The global interrupt enable (GIE) bit must be set in the control status register (CSR) and the appropriate interrupt number’s interrupt enable (IE*n*) bit and the non-maskable interrupt enable (NMIE) bit must be set in the IER. The setting of NMIE is to prevent the processor from being interrupted until it is fully out of the system initialization following reset, and no interrupts can be taken until this is done. The GIE bit globally enables any enabled interrupt to be serviced by the CPU. This bit can be cleared to protect certain routines. The IE*n* bit is used to enable the specific interrupt number of the DMA channel in use.

The most common DMA conditions used to interrupt the CPU are the BLOCK, FRAME, and LAST conditions.

- BLOCK is signaled after the last element of the last frame is transferred. This is typically used to signal that a transfer is complete for non-auto-initialized transfers or to cause an ISR that modifies the global registers used by the channel. The new reloads would be used at the end of the next block.
- FRAME is signaled after the last element of any frame is transferred. This is typically used to synchronize other channels or signal to the CPU that a set of data is available to be processed.
- LAST is signaled after the last element of the second-to-last frame is transferred. This is typically used to modify the count reload of an auto-initialized transfer. The new count would take effect on the first frame of the subsequent block.

These are for planned services that occur during program execution. The remaining conditions are used to service unplanned situations. (R/W)DROP is used in the instance that a synchronized transfer was skipped, and the SX condition is used in the case that a split-mode transfer is not symmetric. An ISR for a DMA channel can address any number of these conditions.

A typical sequence of events in servicing a DMA interrupt is:

- Read the secondary control register.
- Check COND bits to see which condition generated the interrupt.
- Clear the condition(s) by writing 0 to the COND bits.
- Write the secondary control register.[25]
- Perform necessary tasks to service the condition.
- Resume program execution.

An example of a basic ISR is one that can be used in the ping-pong transfer example (see section 5.2). In that example, it is desired to initiate the second DMA channel (servicing the output data) after the second input frame is completed. The FRAME COND bit is also required to be cleared following each frame as the FRAME IE bit is set. The following C code performs what is necessary for the case where DMA channel 1 services the input data, and DMA channel 2 services the output data:

[25] In a synchronized transfer, it is a good idea to mask the RSYNC STAT and WSYNC STAT bits. If a synchronization event is serviced during the ISR, writing a 1 to either may cause a spurious synchronization event. Writing a 0 has no effect.

```
/*----------------------------------------------------------------------------*/
/* DMA Channel 1 Interrupt Service Routine will execute upon                  */
/*  completion of a frame Transfer by Channel 1.  Since Channel 1            */
/*  is servicing the input data, when it completes its transfer,             */
/*  the CPU will be free to begin executing code.                            */
/*----------------------------------------------------------------------------*/
interrupt void     /* vecs.asm hooks this up to IRQ 09                        */
c_int09(void)      /* DMA ch1                                                 */
{
      /* If second frame has completed, start DMA Channel 2 in                */
/*  autoinitialization mode, then clear the Block condition                  */
      /*  bit and disable Block Interrupt Enable.                             */
      if (DMA_FGET(SECCTL1, BLOCKCOND))
      {
            DMA_autoStart(hDma2);
            DMA_FSET(SECCTL1, BLOCKCOND, 0);
            DMA_FSET(SECCTL1, BLOCKIE, 0);
      }

      /* Clear the FRAME COND bit in DMA Channel 1                            */
      DMA_FSET(SECCTL1, FRAMECOND, 0);
} /* END DMA_Ch1_ISR */
```

7 Conclusion

The TMS320C6000 DMA is a versatile tool that can be used to perform data transfers throughout device operation with little setup required. Through proper initialization, both simple and complex data transfers can be run concurrently to provide data to the CPU, and to transmit data to external devices. The examples provided here offer some of the more common DMA applications for a system with setups shown to service peripherals, arrange incoming data streams into useful data, and operate continuously. Knowing how to tie DMA operation into a system allows designers to maximize data throughput and CPU performance.

8 References

1. *TMS320C6000 Peripherals Reference Guide* (SPRU190).
2. *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186).
3. *TMS320C6000 Optimizing C Compiler User's Guide* (SPRU187).
4. *TMS320C6000 CPU and Instruction Set Reference Guide* (SPRU189).
5. *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401).

Appendix A Code Segments for DMA Transfer Examples

Appendix A includes code segments for each of the DMA transfer examples provided in this document. The following code can be modified as required for a given system. Many of these transfers can be used within the same system to perform multiple transfers.

A.1 Block Move Example Code

```
/*-------------------------------------------------------------------------*/
/*  blocktrans_dma.c V1.00                                                 */
/*  Copyright (c) 2001 Texas Instruments Incorporated                      */
/*-------------------------------------------------------------------------*/
/*
   7/16/01
   Original by: Dave Bell
   Modified by: Vassos S. Soteriou

   blocktrans_dma.c:
     Set up the DMA registers to perform data transfer of a block of data.
   This sample code moves a block of data in the DSP internal memory from
   one location (dmaOutbuff) to another location (dmaInbuff). These memory
   locations are not predefined, but are dynamically allocated by the
   program, solely for example purposes (the user can define the memory
   space to be allocated if needed). This program uses interrupt 9 to stop
   the data transfer process when the transfer is complete by setting a
   flag to the CPU (the user can choose any DMA channel and corrsponding
   INT associated with it)
     The sample code is based on TI’s CSL 2.0. See the TMS320C6000 Chip
   Support Library API User’s Guide (SPRU401) for further information.
*/

/* Chip definition, change this accordingly, only DMA supporting DSPs */
#define CHIP_6202 1

/* Include files */
#include <c6x.h>
#include <csl.h>         /* CSL library   */
#include <csl_dma.h>     /* DMA_SUPPORT   */
#include <csl_irq.h>     /* IRQ_SUPPORT   */

/*---------------------------------------------------------------------------*/
/* Define constants */
#define FALSE 0
#define TRUE 1
```

```
#define DMA_TRANS 8
#define XFER_TYPE DMA_TRANS
#define BUFFER_SIZE 256  /* BUFFER_SIZE should be >= ELEMENT_COUNT */
#define ELEMENT_COUNT 32

/* Global variables used in interrupt ISRs */
volatile int transfer_done = FALSE;

/*----------------------------------------------------------------------*/
/* Declare CSL objects                                                  */
DMA_Handle hDma1;           /* Handle for DMA   */
/*----------------------------------------------------------------------*/
/* External functions and function prototypes */
void set_interrupts_dma(void);
/* Include the vector table to call the IRQ ISRs hookup */
extern far void vectors();
/*----------------------------------------------------------------------*/
/* main()                                                               */
/*----------------------------------------------------------------------*/
void main(void)
{

/* Declaration of local variables */
static int element_count, xfer_type;

static Uint32 dmaInbuff[BUFFER_SIZE];   /* define In and Out buffer  */
static Uint32 dmaOutbuff[BUFFER_SIZE];

IRQ_setVecs(vectors); /* point to the IRQ vector table */

element_count = ELEMENT_COUNT;
xfer_type = XFER_TYPE;

/* initialize the CSL library */
CSL_init();

switch (xfer_type) {
case DMA_TRANS:

DMA_reset(INV);         /* Reset all DMA channels */
```

```
/*----------------------------------------------------------------------*/
/* DMA channels 1 config structure                                      */
/*----------------------------------------------------------------------*/
  /* DMA channel 1 */
  hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);  /* Handle to DMA ch1 */
    DMA_configArgs(hDma1,
       DMA_PRICTL_RMK(
           DMA_PRICTL_DSTRLD_DEFAULT,
           DMA_PRICTL_SRCRLD_DEFAULT,
           DMA_PRICTL_EMOD_DEFAULT,
           DMA_PRICTL_FS_DEFAULT,
           DMA_PRICTL_TCINT_ENABLE, /* TCINT =1 */
           DMA_PRICTL_PRI_DMA,      /* DMA high priority */
           DMA_PRICTL_WSYNC_DEFAULT,
           DMA_PRICTL_RSYNC_DEFAULT,
           DMA_PRICTL_INDEX_DEFAULT,
           DMA_PRICTL_CNTRLD_DEFAULT,
           DMA_PRICTL_SPLIT_DISABLE,
           DMA_PRICTL_ESIZE_32BIT,  /* Element size is 32 bits        */
           DMA_PRICTL_DSTDIR_INC,   /* Increment dest by element size    */
           DMA_PRICTL_SRCDIR_INC,   /* Increment src by element size     */
           DMA_PRICTL_START_DEFAULT
           ),
        DMA_SECCTL_RMK(
           DMA_SECCTL_WSPOL_NA,  /* only available for 6202/6203 devices */
           DMA_SECCTL_RSPOL_NA,  /* only available for 6202/6203 devices */
           DMA_SECCTL_FSIG_NA,   /* only available for 6202/6203 devices */
           DMA_SECCTL_DMACEN_DEFAULT,
           DMA_SECCTL_WSYNCCLR_DEFAULT,
           DMA_SECCTL_WSYNCSTAT_DEFAULT,
           DMA_SECCTL_RSYNCCLR_DEFAULT,
           DMA_SECCTL_RSYNCSTAT_DEFAULT,
           DMA_SECCTL_WDROPIE_DEFAULT,
           DMA_SECCTL_WDROPCOND_DEFAULT,
           DMA_SECCTL_RDROPIE_DEFAULT,
           DMA_SECCTL_RDROPCOND_DEFAULT,
           DMA_SECCTL_BLOCKIE_ENABLE, // BLOCK IE=1 enables DMA channel int
           DMA_SECCTL_BLOCKCOND_DEFAULT,
           DMA_SECCTL_LASTIE_DEFAULT,
           DMA_SECCTL_LASTCOND_DEFAULT,
           DMA_SECCTL_FRAMEIE_DEFAULT,
           DMA_SECCTL_FRAMECOND_DEFAULT,
           DMA_SECCTL_SXIE_DEFAULT,
           DMA_SECCTL_SXCOND_DEFAULT
           ),
        DMA_SRC_RMK((Uint32)dmaOutbuff),  /* source buffer      */
        DMA_DST_RMK((Uint32)dmaInbuff),   /* destination buffer */
        DMA_XFRCNT_RMK(
           DMA_XFRCNT_FRMCNT_DEFAULT,
           DMA_XFRCNT_ELECNT_OF(element_count) /* set xfer element count */
           )
);
/* initialize the interrupts:                              */
/*  Enable the interrupts after the DMA channels are opened */
/*  as the DMA_OPEN_RESET clears and disables the channel   */
/*  interrupt when specified and clears the corresponding   */
/*  interrupt bits in the IER.                                  */
set_interrupts_dma();

DMA_start(hDma1);     /* Start DMA channel 1               */

} /* end of switch here */

/* To flag an interrupt to the CPU when DMA transfer/receive is done   */
while (!transfer_done);

DMA_close(hDma1); /* close the channel when the transfer is complete */
} /* end main, program ends here */

/*----------------------------------------------------------------------*/
/* set_interrupts_dma()                                                 */
/*----------------------------------------------------------------------*/
void                          /* Set the interrupts */
set_interrupts_dma(void)
{
   IRQ_nmiEnable();
   IRQ_globalEnable();
   IRQ_disable(IRQ_EVT_DMAINT1);  /* INT9 */
   IRQ_clear(IRQ_EVT_DMAINT1);
   IRQ_enable(IRQ_EVT_DMAINT1);
   return;
}
```

```
/*----------------------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISRs                                       */
/*----------------------------------------------------------------------------*/
interrupt void       /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)        /* DMA ch1                          */
{
   transfer_done = TRUE;
   return;
}
/*-----------------------End of blocktrans_dma.c------------------------------*/
```

A.2 Extremely Large Block Move Example Code

```
/*-----------------------------------------------------------------------------*/
/*  largeblocktrans_dma.c V1.00                                                */
/*  Copyright (c) 2001 Texas Instruments Incorporated                          */
/*-----------------------------------------------------------------------------*/
/*
   7/16/01
   Original by: Dave Bell
   Modified by: Vassos S. Soteriou

   largeblocktrans_dma.c:
      Setup the DMA control registers to perform a large block data transfer.
   This sample code moves a block of data from an external memory/space/buffer
   (AFEbuff) to the internal DSP data memory (dmaInbuff). The user can define
   the external memory space to transfer data from as required in a specific
   application, for instance CE0 EMIF space. This program uses interrupt 9 to
   stop the data transfer process when the transfer is complete by setting a
   flag to the CPU (the user can choose any DMA channel and corrsponding INT
   associated with it)
   The sample code is based on TI's CSL 2.0. See the TMS320C6000 Chip Support
   Library API User's Guide (SPRU401) for further information.
*/

/* Chip definition, change this accordingly, only DMA supporting DSPs though */
#define  CHIP_6202 1
/* Include files */
#include <c6x.h>
#include <csl.h>         /* CSL library   */
#include <csl_dma.h>     /* DMA_SUPPORT   */
#include <csl_irq.h>     /* IRQ_SUPPORT   */
```

```
/*----------------------------------------------------------------------------*/
/* Define constants */
#define FALSE 0
#define TRUE 1
#define DMA_TRANS 8
#define XFER_TYPE DMA_TRANS
#define BUFFER_SIZE 0x8000     /* BUFFER_SIZE contains 65535 memeory locs.  */
#define ELEMENT_COUNT 0xFFFFFF /* Element count greater than 65535 elements */
                               /* change the element count accordingly      */

/* Global variables used in interrupt ISRs */
volatile int transfer_done = FALSE;

/* Declare CSL objects */
DMA_Handle hDma1;                  /* Handle for DMA                            */
Uint32 dmaGblRegMsk;               /* DMA Global Register Mask                  */
Uint32 dmaGblRegId = DMA_GBLCNTA;  /* Select Global Count Reload Register A  */
/*----------------------------------------------------------------------------*/
/* External functions and function prototypes */
void set_interrupts_dma(void);
/* Include the vector table to call the IRQ ISRs hookup */
extern far void vectors();

/*----------------------------------------------------------------------------*/
/* main()                                                                     */
/*----------------------------------------------------------------------------*/
void main(void)
{

/* Declaration of local variables */
static int element_count, xfer_type, frame_count, initial_element,
reload_element;

static Uint32 dmaInbuff[BUFFER_SIZE]; /*define internal data mem buffer      */
#pragma DATA_ALIGN (AFEbuff, 0x8000); /*Assign AFEbuff to alignment boundary */

/* Allocate space for AFEbuff in MEMORY/SECTIONS section of the link  */
/* command file, *.cmd. Note that the user  also has to define the    */
/* origin and length of this ext memory in the cmd file               */

#pragma DATA_SECTION(AFEbuff, ”external_data”);
static Uint32 AFEbuff[BUFFER_SIZE];   /* define external memory buffer       */
```

```
IRQ_setVecs(vectors);                /* point to the IRQ vector table       */

/* Establish initial count value, and reload value, based on transfer size   */
/* ELEMENT_COUNT) of large block.  The formula used to calculate the initial */
/* and reload is the following:                                              */
/*                                                                           */
/*   Initial element count = 15 LSBs of total transfer size                  */
/*   Frame count = bits 15 through 30, plus 1                                */
/*                                                                           */
/* NOTE: The maximum size using this method is 0x7FFF7FFF. For larger sizes, */
/*       a new formula must be used.                                         */

element_count = ELEMENT_COUNT;
xfer_type = XFER_TYPE;
frame_count = (element_count >> 15 ) + 1;
initial_element = element_count & 0x7FFF;

if (!initial_element)  /* element count of 0 not allowed */
{
     initial_element = 0x8000;
     frame_count –= 1;
}
reload_element = 0x8000;

/* initialize the CSL library */
CSL_init();

switch (xfer_type) {
case DMA_TRANS:

DMA_reset(INV);    /* Reset all DMA channels */
/*-------------------------------------------------------------------------*/
/* DMA channel 1 config structures                                           */
/*-------------------------------------------------------------------------*/
/* Establish Global Register Values in the following configuration structure */
dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);
DMA_globalConfigArgs(dmaGblRegMsk,
      DMA_GBLADDR_GBLADDR_DEFAULT,  /* DMA global address registers A to D */
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLIDX_DEFAULT,   /* DMA global index registers A to B      */
      DMA_GBLIDX_DEFAULT,
      DMA_GBLCNT_RMK(   /* config DMA global count relaod register A  */
         DMA_GBLCNT_FRMCNT_DEFAULT,
         DMA_GBLCNT_ELECNT_OF((Uint32) reload_element) // element count reload
      ),
      DMA_GBLCNT_DEFAULT
);  /* end of DMA global control register configuration structure */

/* Channel 1 receives the data */
hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);  /* Handle to DMA channel 1 */
DMA_configArgs(hDma1,
       DMA_PRICTL_RMK(
           DMA_PRICTL_DSTRLD_DEFAULT,
           DMA_PRICTL_SRCRLD_DEFAULT,
           DMA_PRICTL_EMOD_DEFAULT,
           DMA_PRICTL_FS_DEFAULT,
           DMA_PRICTL_TCINT_ENABLE, /* TCINT =1                            */
           DMA_PRICTL_PRI_DMA,      /* DMA high priority                   */
           DMA_PRICTL_WSYNC_DEFAULT,
           DMA_PRICTL_RSYNC_DEFAULT,
           DMA_PRICTL_INDEX_DEFAULT,
           DMA_PRICTL_CNTRLD_A,  /* Reload with DMA global reload countrer A */
           DMA_PRICTL_SPLIT_DISABLE,
           DMA_PRICTL_ESIZE_32BIT,  /* Element size is 32 bits             */
           DMA_PRICTL_DSTDIR_INC,   /* Increment destination by element size */
           DMA_PRICTL_SRCDIR_INC,   /* Increment source by element size      */
           DMA_PRICTL_START_DEFAULT
           ),

        DMA_SECCTL_RMK(
           DMA_SECCTL_WSPOL_NA,  /* only available for 6202 and 6203 devices */
           DMA_SECCTL_RSPOL_NA,  /* only available for 6202 and 6203 devices */
           DMA_SECCTL_FSIG_NA,   /* only available for 6202 and 6203 devices */
           DMA_SECCTL_DMACEN_DEFAULT,
           DMA_SECCTL_WSYNCCLR_DEFAULT,
           DMA_SECCTL_WSYNCSTAT_DEFAULT,
           DMA_SECCTL_RSYNCCLR_DEFAULT,
           DMA_SECCTL_RSYNCSTAT_DEFAULT,
           DMA_SECCTL_WDROPIE_DEFAULT,
           DMA_SECCTL_WDROPCOND_DEFAULT,
           DMA_SECCTL_RDROPIE_DEFAULT,
           DMA_SECCTL_RDROPCOND_DEFAULT,
           DMA_SECCTL_BLOCKIE_ENABLE, /* BLOCK IE=1 enables DMA channel int */
           DMA_SECCTL_BLOCKCOND_DEFAULT,
           DMA_SECCTL_LASTIE_DEFAULT,
           DMA_SECCTL_LASTCOND_DEFAULT,
           DMA_SECCTL_FRAMEIE_DEFAULT,
           DMA_SECCTL_FRAMECOND_DEFAULT,
           DMA_SECCTL_SXIE_DEFAULT,
           DMA_SECCTL_SXCOND_DEFAULT
           ),
        DMA_SRC_RMK((Uint32)AFEbuff),    /* external source buffer */
        DMA_DST_RMK((Uint32)dmaInbuff),  /* destination buffer   */
        DMA_XFRCNT_RMK(
           DMA_XFRCNT_FRMCNT_OF(frame_count),
           DMA_XFRCNT_ELECNT_OF(initial_element) /* set xfer element count */
           )
);

/* Initialize the interrupts:                               */
/* Enable the interrupts after the DMA channels are opened   */
/*  as the DMA_OPEN_RESET clears and disables the channel    */
/*  interrupt once specified and clears the corresponding    */
/*  interrupt bits in the IER.                               */
set_interrupts_dma();

DMA_start(hDma1);     /* Start DMA channel 1 */
} /* end of switch here */

/* To flag an interrupt to the CPU when DMA transfer/receive is done   */
  while (!transfer_done);

DMA_close(hDma1); /* close the channel when the transfer is complete */
} /* end main, program ends here */

/*--------------------------------------------------------------------------------*/
/* set_interrupts_dma()                                                           */
/*--------------------------------------------------------------------------------*/
void                           /* Set the interrupts */
set_interrupts_dma(void)
{
     IRQ_nmiEnable();
     IRQ_globalEnable();
     IRQ_disable(IRQ_EVT_DMAINT1);  /* INT9 */
     IRQ_clear(IRQ_EVT_DMAINT1);
     IRQ_enable(IRQ_EVT_DMAINT1);
     return;
}

/*-------------------------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISRs                                          */
/*-------------------------------------------------------------------------------*/
interrupt void       /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)        /* DMA ch1                          */
{
   transfer_done = TRUE;
   return;
}
/*-----------------------End of largeblocktrans_dma.c---------------------------*/
```

A.3 Data-Sorting Example Code

```
/*------------------------------------------------------------------------------*/
/*   datasort_dma.c V1.00                                                       */
/*   Copyright (c) 2001 Texas Instruments Incorporated                          */
/*------------------------------------------------------------------------------*/
/*
   7/17/01
   Original by: Dave Bell
   Modified by: Vassos S. Soteriou

   datasort_dma.c:
      This sample code sets up the DMA control registers to perform column-wise
   sort of data arrays located in internal memory, from dmaOutbuff to dmaInbuff
   DSP internal memory.  These memory arrays are not predefined by the user but
   are dynamically allocated by the program, solely for excample purposes (the
   user can define the memory space to be allocated if needed). This program
   uses interrupt 9 to stop the data transfer process when the transfer is
   complete by setting a flag to the CPU (the user can choose any DMA channel
   and corrsponding INT associated with it)
      The sample code is based on TI’s CSL 2.0. See the TMS320C6000 Chip Support
   Library API User’s Guide (SPRU401) for further information.
*/
```

```
/* Chip definition, change this accordingly, only DMA DSPs though */
#define CHIP_6202 1
/* Include files */
#include <c6x.h>
#include <csl.h>         /* CSL library   */
#include <csl_dma.h>     /* DMA_SUPPORT   */
#include <csl_irq.h>     /* IRQ_SUPPORT   */
/*-------------------------------------------------------------------------*/
/* Define constants */
#define FALSE 0
#define TRUE 1

#define DMA_TRANS 8
#define XFER_TYPE DMA_TRANS  // Let BUFFER SIZE be greater or equal in size to:
#define BUFFER_SIZE 16  // >= (ELEMENT_COUNT * FRAME_COUNT) * (ELEMENT_SIZE/4)
#define ELEMENT_SIZE 1  // ELEMENT_SIZE is the number of bytes/element, 1,2,4
#define FRAME_COUNT 2   // Define the number of frames
#define ELEMENT_COUNT 8 // Define the number of elements to transfer

/* Global variables used in interrupt ISRs */
volatile int transfer_done = FALSE;

/* Declare CSL objects */
DMA_Handle hDma1;           /* Handle for DMA            */
Uint32 dmaGblRegMsk;        /* DMA Global Register Mask  */
Uint32 dmaGblRegId = DMA_GBLCNTA | DMA_GBLIDXA;

/* Select Global Address Reload Register A or Global Index Register A */
/*--------------------------------------------------------------------------*/
/* External functions and function prototypes */
void set_interrupts_dma(void);
/* Include the vector table to call the IRQ ISRs hookup */
extern far void vectors();

/*--------------------------------------------------------------------------*/
/* main()                                                                   */
/*--------------------------------------------------------------------------*/
void main(void)
{

/* Declaration of local variables */
static int element_count, xfer_type, element_size, element_index;
static int frame_count, frame_index;
static unsigned int dmaInbuff[BUFFER_SIZE];   /* define In and Out buffer  */
static unsigned int dmaOutbuff[BUFFER_SIZE];

IRQ_setVecs(vectors); /* point to the IRQ vector table */
element_count = ELEMENT_COUNT;
xfer_type = XFER_TYPE;
frame_count = FRAME_COUNT;

/* Calculate the index values, as well as the ESIZE, based on the number of */
/* elements per frame (element_count), the number of frames per block       */
/* (frame_count), and the number of bytes in each element (element_size)    */

element_index = frame_count * ELEMENT_SIZE;
frame_index = –(((element_count – 1) * frame_count) –1) * ELEMENT_SIZE;

/* covert the number of bytes/element into the CSL HAL MACRO conversion */
if (ELEMENT_SIZE == 1) element_size = 2;       /* 8BIT element size  */
else if (ELEMENT_SIZE == 2) element_size = 1;  /* 16BIT element size */
else element_size = 0;                         /* 32BIT element size */

/* initialize the CSL library */
CSL_init();

switch (xfer_type) {
case DMA_TRANS:

DMA_reset(INV);    /* Reset all DMA channels */

/*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
/* DMA channel 1 config structure                                             */
/*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);
DMA_globalConfigArgs(dmaGblRegMsk,
      DMA_GBLADDR_GBLADDR_DEFAULT,  /* DMA global address registers A to D */
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLADDR_GBLADDR_DEFAULT,

     /* DMA global index registers A to B   */
     DMA_GBLIDX_RMK(
          DMA_GBLIDX_FRMIDX_OF(frame_index),  /* Set frame index   */
          DMA_GBLIDX_ELEIDX_OF(element_index) /* Set element index */
          ),
     DMA_GBLIDX_DEFAULT,
     /* config DMA global count relaod register A  */
     DMA_GBLCNT_RMK(
          DMA_GBLCNT_FRMCNT_OF(frame_count),  /* frame count reload   */
          DMA_GBLCNT_ELECNT_OF(element_count) /* element count reload */
          ),
      DMA_GBLCNT_DEFAULT
);  /* end of DMA global control register configuration structure */

hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);  /* Handle to DMA channel 1 */
DMA_configArgs(hDma1,
      DMA_PRICTL_RMK(
           DMA_PRICTL_DSTRLD_DEFAULT,
           DMA_PRICTL_SRCRLD_DEFAULT,
           DMA_PRICTL_EMOD_DEFAULT,
           DMA_PRICTL_FS_DEFAULT,
           DMA_PRICTL_TCINT_ENABLE, /* TCINT =1             */
           DMA_PRICTL_PRI_DMA,      /* DMA high priority    */
           DMA_PRICTL_WSYNC_DEFAULT,
           DMA_PRICTL_RSYNC_DEFAULT,
           DMA_PRICTL_INDEX_A,    /* Use Global Index Register A             */
           DMA_PRICTL_CNTRLD_A,   /* Reload with DMA global reload counter A */
           DMA_PRICTL_SPLIT_DISABLE,
           DMA_PRICTL_ESIZE_OF(element_size), // Element size defined by user
           DMA_PRICTL_DSTDIR_IDX,  //Adjust dest. using DMA Global Index Reg. A

      DMA_PRICTL_SRCDIR_INC,
           DMA_PRICTL_START_DEFAULT
           ),

   DMA_SECCTL_RMK(
           DMA_SECCTL_WSPOL_NA,  /* only available for 6202 and 6203 devices */
           DMA_SECCTL_RSPOL_NA,  /* only available for 6202 and 6203 devices */
           DMA_SECCTL_FSIG_NA,   /* only available for 6202 and 6203 devices */
           DMA_SECCTL_DMACEN_DEFAULT,
           DMA_SECCTL_WSYNCCLR_DEFAULT,
           DMA_SECCTL_WSYNCSTAT_DEFAULT,
           DMA_SECCTL_RSYNCCLR_DEFAULT,
           DMA_SECCTL_RSYNCSTAT_DEFAULT,
           DMA_SECCTL_WDROPIE_DEFAULT,
           DMA_SECCTL_WDROPCOND_DEFAULT,
           DMA_SECCTL_RDROPIE_DEFAULT,
           DMA_SECCTL_RDROPCOND_DEFAULT,
           DMA_SECCTL_BLOCKIE_ENABLE, /* BLOCK IE=1 enables DMA channel int */
           DMA_SECCTL_BLOCKCOND_DEFAULT,
           DMA_SECCTL_LASTIE_DEFAULT,
           DMA_SECCTL_LASTCOND_DEFAULT,
           DMA_SECCTL_FRAMEIE_DEFAULT,
           DMA_SECCTL_FRAMECOND_DEFAULT,
           DMA_SECCTL_SXIE_DEFAULT,
           DMA_SECCTL_SXCOND_DEFAULT
           ),
        DMA_SRC_RMK((unsigned int) dmaOutbuff), /* source buffer      */
        DMA_DST_RMK((unsigned int) dmaInbuff),  /* destination buffer */
        DMA_XFRCNT_RMK(
           DMA_XFRCNT_FRMCNT_OF(frame_count),  /* set xfer frame count   */
           DMA_XFRCNT_ELECNT_OF(element_count) /* set xfer element count */
           )
);

/* Initialize the interrupts                                 */
/*   Enable the interrupts after the DMA channels are opened   */
/* as the DMA_OPEN_RESET clears and disables the channel       */
/* interrupt once specified and clears the corresponding       */
/* interrupt bits in the IER.                                 */
set_interrupts_dma();

DMA_start(hDma1);     /* Start DMA channel 1 */
} /* end of switch here */

/* To flag an interrupt to the CPU when DMA transfer/receive is done   */
  while (!transfer_done);

DMA_close(hDma1); /* close the channel when the transfer is complete */
} /* end main, program ends here */

/*----------------------------------------------------------------------------*/
/* set_interrupts_dma()                                                       */
/*----------------------------------------------------------------------------*/
void                           /* Set the interrupts */
set_interrupts_dma(void)
{
   IRQ_nmiEnable();
   IRQ_globalEnable();
   IRQ_disable(IRQ_EVT_DMAINT1);  /* INT9 */
   IRQ_clear(IRQ_EVT_DMAINT1);
   IRQ_enable(IRQ_EVT_DMAINT1);
   return;
}

/*---------------------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISR                                       */
/*---------------------------------------------------------------------------*/
interrupt void       /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)        /* DMA ch1                          */
{
     transfer_done = TRUE;
     return;
}
/*-----------------------End of datasort_dma.c-------------------------------*/
```

A.4 Synchronized Data Transfer Example Code

```
/*---------------------------------------------------------------------------*/
/*  synctrans_dma.c V1.00                                                    */
/*  Copyright (c) 2001 Texas Instruments Incorporated                        */
/*---------------------------------------------------------------------------*/
/*
   7/19/01
   Original by: Dave Bell
   Modified by: Vassos S. Soteriou

   synctrans_dma.c:
   This program sets up the DMA control registers to perform data transfer from
   the McBSP0 of a TMS320C6000 device to the DSP internal data memory (dmaInbuff)
   using DMA channel 1.  This program uses interrupt 9 to stop the
   data transfer process when the transfer is complete by setting a flag to the
   CPU (the user can choose any DMA channel and corrsponding INT associated with
   it). Also note that the McBSP can be configured to match the requirements of
   a specific application.
   The sample code is based on TI’s CSL 2.0.  See the TMS320C6000 Chip Support
   Library API User’s Guide (SPRU401) for further information.                 */

/* Chip definition, change this accordingly */

#define CHIP_6202 1
```

```
/* Include files */
#include <c6x.h>
#include <csl.h>         /* CSL library   */
#include <csl_dma.h>     /* DMA_SUPPORT   */
#include <csl_irq.h>     /* IRQ_SUPPORT   */
#include <csl_mcbsp.h>   /* MCBSP_SUPPORT */

/* Define constants */
#define FALSE 0
#define TRUE 1

#define DMA_XFER 8
#define XFER_TYPE DMA_XFER
#define BUFFER_SIZE 256
#define ELEMENT_COUNT 32

/* Global variables used in interrupt ISRs */
volatile int transfer_done = FALSE;

/* Declare CSL objects */
MCBSP_Handle hMcbsp0;       /* Handles for McBSP         */
DMA_Handle hDma1;           /* Handle for DMA            */
Uint32 dmaGblRegMsk;        /* DMA Global Register Mask  */
Uint32 dmaGblRegId = DMA_GBLCNTA;  /* Select Global Addr. Count Reload Reg A */

/*----------------------------------------------------------------------------*/
/* External functions and function prototypes */
void init_mcbsp0_dma(void);   /* Function prototypes */
void set_interrupts_dma(void);

/* Include the vector table to call the IRQ ISRs hookup */
extern far void vectors();

/*----------------------------------------------------------------------------*/
/* main()                                                                     */
/*----------------------------------------------------------------------------*/
void main(void)
{
/* Declaration of local variables */
static int element_count, xfer_type;

static Uint32 dmaInbuff[BUFFER_SIZE];  /* buffer for DMA supporting devices  */

IRQ_setVecs(vectors); /* point to the IRQ vector table */
```

```
element_count = ELEMENT_COUNT;
xfer_type = XFER_TYPE;

/* initialize the CSL library */
CSL_init();

init_mcbsp0_dma();

/* Enable sample rate generator GRST=1 */
MCBSP_enableSrgr(hMcbsp0);  /* Handle to SRGR */

switch (xfer_type) {
case DMA_XFER:

DMA_reset(INV);         /* reset all DMA channels */
/*----------------------------------------------------------------------------*/
/* DMA channel 1 config structure                                             */
/*----------------------------------------------------------------------------*/
/* Channel 1 receives the data */
/* Establish Global Register Values in the following configuration structure */
dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);
DMA_globalConfigArgs(dmaGblRegMsk, /* DMA global address registers A to D */
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLIDX_DEFAULT,
      DMA_GBLIDX_DEFAULT,
      DMA_GBLCNT_RMK(   /* config DMA global count relaod register A  */
         DMA_GBLCNT_FRMCNT_DEFAULT,
         DMA_GBLCNT_ELECNT_OF(element_count) /* element count reload  */
      ),
      DMA_GBLCNT_DEFAULT
);  /* end of DMA global control register configuration structure */

  hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);  /* Handle to DMA channel 1 */
    DMA_configArgs(hDma1,
       DMA_PRICTL_RMK(
           DMA_PRICTL_DSTRLD_DEFAULT,
           DMA_PRICTL_SRCRLD_DEFAULT,
           DMA_PRICTL_EMOD_HALT, /* DMA cahnnel pauses during emulation halt */
           DMA_PRICTL_FS_DEFAULT,
           DMA_PRICTL_TCINT_ENABLE, /* TCINT =1                              */
           DMA_PRICTL_PRI_DMA,      /* DMA high priority                    */
           DMA_PRICTL_WSYNC_DEFAULT,
           DMA_PRICTL_RSYNC_XEVT0,  /* Set synchronization event XEVT0=01100 */
           DMA_PRICTL_INDEX_DEFAULT,
           DMA_PRICTL_CNTRLD_A,  /* Reload with DMA global reload countrer A */
           DMA_PRICTL_SPLIT_DEFAULT,
           DMA_PRICTL_ESIZE_32BIT,  /* Element size 32 bits                  */
           DMA_PRICTL_DSTDIR_INC,   /* Increment destination by element size */
           DMA_PRICTL_SRCDIR_DEFAULT,
           DMA_PRICTL_START_DEFAULT
           ),

        DMA_SECCTL_RMK(
           DMA_SECCTL_WSPOL_NA,  /* only available for 6202 and 6203 devices */
           DMA_SECCTL_RSPOL_NA,  /* only available for 6202 and 6203 devices */
           DMA_SECCTL_FSIG_NA,   /* only available for 6202 and 6203 devices */
           DMA_SECCTL_DMACEN_DEFAULT,
           DMA_SECCTL_WSYNCCLR_DEFAULT,
           DMA_SECCTL_WSYNCSTAT_DEFAULT,
           DMA_SECCTL_RSYNCCLR_DEFAULT,
           DMA_SECCTL_RSYNCSTAT_DEFAULT,
           DMA_SECCTL_WDROPIE_DEFAULT,
           DMA_SECCTL_WDROPCOND_DEFAULT,
           DMA_SECCTL_RDROPIE_DEFAULT,
           DMA_SECCTL_RDROPCOND_DEFAULT,
           DMA_SECCTL_BLOCKIE_ENABLE, //BLOCK IE=1 enables DMA channel int
           DMA_SECCTL_BLOCKCOND_DEFAULT,
           DMA_SECCTL_LASTIE_DEFAULT,
           DMA_SECCTL_LASTCOND_DEFAULT,
           DMA_SECCTL_FRAMEIE_DEFAULT,
           DMA_SECCTL_FRAMECOND_DEFAULT,
           DMA_SECCTL_SXIE_DEFAULT,
           DMA_SECCTL_SXCOND_DEFAULT
           ),

     /* McBSP DRR0 is the data source     */
      DMA_SRC_RMK(MCBSP_ADDRH(hMcbsp0, DRR)),

      /* Data destination internal DSP data memory*/
      DMA_DST_RMK((Uint32)dmaInbuff),
      DMA_XFRCNT_RMK(
         DMA_XFRCNT_FRMCNT_DEFAULT,
         DMA_XFRCNT_ELECNT_OF(element_count)  /* set recv element count */
         )
);
/* Initialize the interrupt(s)                                    */
/*   Enable the interrupt after the DMA channels are opened as    */
/* the DMA_OPEN_RESET clears and disables the channel interrupt   */
/* once specified and clears the corresponding interrupt bits     */
/* in the IER.                                                    */
set_interrupts_dma();

DMA_start(hDma1);     /* Start DMA channel 1 */
} /* end of switch here */

/* Enable McBSP channel */
MCBSP_enableRcv(hMcbsp0);  /* McBSP port 0 as the transmitter  */

/* To flag an interrupt to the CPU when DMA transfer/receive is done */
while (!transfer_done);

MCBSP_close(hMcbsp0);  /* close McBSP port */

DMA_close(hDma1);  /* close DMA channels */
} /* end main, program ends here */

/*----------------------------------------------------------------------------*/
/* init_mcbsp0_dma()                                                          */
/*----------------------------------------------------------------------------*/
/* MCBSP Config structure */
/* Setup the MCBSP_0 for data receive */
void
init_mcbsp0_dma(void)
{
MCBSP_Config mcbspCfg0 = {
     MCBSP_SPCR_RMK(
          MCBSP_SPCR_FRST_DEFAULT,  //All fields in SPCR set to default values
          MCBSP_SPCR_GRST_DEFAULT,
          MCBSP_SPCR_XINTM_DEFAULT,
          MCBSP_SPCR_XSYNCERR_DEFAULT,
          MCBSP_SPCR_XRST_DEFAULT,
          MCBSP_SPCR_DLB_DEFAULT,
          MCBSP_SPCR_RJUST_DEFAULT,
          MCBSP_SPCR_CLKSTP_DEFAULT,
          MCBSP_SPCR_RINTM_DEFAULT,
          MCBSP_SPCR_RSYNCERR_DEFAULT,
          MCBSP_SPCR_RRST_DEFAULT
          ),

     MCBSP_RCR_RMK(
           MCBSP_RCR_RPHASE_SINGLE,    /* Single phase receive frame */
           MCBSP_RCR_RFRLEN2_DEFAULT,
           MCBSP_RCR_RWDLEN2_DEFAULT,
           MCBSP_RCR_RCOMPAND_DEFAULT,
           MCBSP_RCR_RFIG_DEFAULT,
           MCBSP_RCR_RDATDLY_1BIT,     /* 1-bit receive data delay    */
           MCBSP_RCR_RFRLEN1_DEFAULT,
           MCBSP_RCR_RWDLEN1_DEFAULT
           ),

     MCBSP_XCR_RMK(
           MCBSP_XCR_XPHASE_DEFAULT, // All fields in XCR set to default values
           MCBSP_XCR_XFRLEN2_DEFAULT,
           MCBSP_XCR_XWDLEN2_DEFAULT,
           MCBSP_XCR_XCOMPAND_DEFAULT,
           MCBSP_XCR_XFIG_DEFAULT,
           MCBSP_XCR_XDATDLY_DEFAULT,
           MCBSP_XCR_XFRLEN1_DEFAULT,
           MCBSP_XCR_XWDLEN1_DEFAULT
           ),

     MCBSP_SRGR_RMK(
           MCBSP_SRGR_GSYNC_DEFAULT, //All fields in SRGR set to default values
           MCBSP_SRGR_CLKSP_DEFAULT,
           MCBSP_SRGR_CLKSM_DEFAULT,
           MCBSP_SRGR_FSGM_DEFAULT,
           MCBSP_SRGR_FPER_DEFAULT,
           MCBSP_SRGR_FWID_DEFAULT,
           MCBSP_SRGR_CLKGDV_DEFAULT
           ),

     MCBSP_MCR_RMK(
           MCBSP_MCR_XPBBLK_DEFAULT, // All fields in MCR set to default values
           MCBSP_MCR_XPABLK_DEFAULT,
           MCBSP_MCR_XMCM_DEFAULT,
           MCBSP_MCR_RPBBLK_DEFAULT,
           MCBSP_MCR_RPABLK_DEFAULT,
           MCBSP_MCR_RMCM_DEFAULT
           ),

     MCBSP_RCER_RMK(
           MCBSP_RCER_RCEB_DEFAULT, // All fields in RCER set to default values
           MCBSP_RCER_RCEA_DEFAULT
           ),

     MCBSP_XCER_RMK(
           MCBSP_XCER_XCEB_DEFAULT, // All fields in XCER set to default values
           MCBSP_XCER_XCEA_DEFAULT
           ),

     MCBSP_PCR_RMK(
           MCBSP_PCR_XIOEN_DEFAULT,
           MCBSP_PCR_RIOEN_DEFAULT,
           MCBSP_PCR_FSXM_DEFAULT,
           MCBSP_PCR_FSRM_DEFAULT,
           MCBSP_PCR_CLKXM_DEFAULT,
           MCBSP_PCR_CLKRM_DEFAULT,
           MCBSP_PCR_CLKSSTAT_DEFAULT,
           MCBSP_PCR_DXSTAT_DEFAULT,
           MCBSP_PCR_FSXP_DEFAULT,
           MCBSP_PCR_FSRP_DEFAULT,
           MCBSP_PCR_CLKXP_DEFAULT,
           MCBSP_PCR_CLKRP_DEFAULT
           )
};
hMcbsp0 = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET); /* McBSP port 0 */
MCBSP_config(hMcbsp0, &mcbspCfg0);
}
/*----------------------------------------------------------------------------*/
/* set_interrupts_dma()                                                       */
/*----------------------------------------------------------------------------*/
void                           /* Set the interrupts */
set_interrupts_dma(void)
{
   IRQ_nmiEnable();
   IRQ_globalEnable();
   IRQ_disable(IRQ_EVT_DMAINT1);  /* INT09 */
   IRQ_clear(IRQ_EVT_DMAINT1);
   IRQ_enable(IRQ_EVT_DMAINT1);
   return;
}

/*---------------------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISR                                       */
/*---------------------------------------------------------------------------*/
interrupt void     /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)      /* DMA ch1                          */
{
   transfer_done = TRUE;
   return;
}
/*-----------------------End of synctrans_dma.c------------------------------*/
```

A.5 Split-Mode Transfer Example Code

```
/*---------------------------------------------------------------------------*/
/*   splitmode_dma.c V1.00                                                   */
/*   Copyright (c) 2001 Texas Instruments Incorporated                       */
/*---------------------------------------------------------------------------*/
/*
   7/19/01
   Original by: Dave Bell
   Modified by: Vassos S. Soteriou

   splitmode_dma.c:
     This program sets up the DMA control registers to service the McBSP0 of a
   TMS320C6000 device in the data rcev/xmit split mode. Data is transferred
   to/from the internal DSP data memory (dmaInbuff stores data in the DSP
   internal memory and dmaOutbuff loads data from the DSP internal memory to
   external memory) to an external memory/buffer via the McBSP.  Although this
   transfer can easily be done using a second DMA channel, one channel to handle
   data transmit and one to handle data receive, one of the features of the DMA
   controller is that a single DMA channel can be used to service both the input
   and output data streams of a peripheral for which the transmit and receive
   addresses are fixed.  In this sample code DMA channel 1 is hooked up to
   interrupt 09 (this is a DMA-INT default mapping). Also note that the McBSP
   can be configured to match the requirements of a specific application.
   The sample code is based on TI’s CSL 2.0.  See the TMS320C6000 Chip
   Support Library API User’s Guide (SPRU401) for further information. Note that
   any DMA channel with the corresponding interrupt and any McBSP port (0 or 1)
   can be used for this data transfer.
*/

/* Chip definition, change this accordingly */
#define CHIP_6202 1

/* Include files */
#include <c6x.h>
#include <csl.h>         /* CSL library   */
#include <csl_dma.h>     /* DMA_SUPPORT   */
#include <csl_irq.h>     /* IRQ_SUPPORT   */
#include <csl_mcbsp.h>   /* MCBSP_SUPPORT */

/* Define constants */
#define FALSE 0
#define TRUE 1

#define DMA_XFER 8
#define XFER_TYPE DMA_XFER
#define BUFFER_SIZE 256  /* set same value as xmit */
#define ELEMENT_COUNT 32 /* set element_count =< buffer_size, set same value as
xmit*/

/* Global variables used in interrupt ISRs */
volatile int transfer_done = FALSE;
/*---------------------------------------------------------------------------*/
/* Declare CSL objects */
MCBSP_Handle hMcbsp0;       /* Handles for McBSP         */
DMA_Handle hDma1;           /* Handle for DMA            */
Uint32 dmaGblRegMsk;        /* DMA Global Register Mask  */
Uint32 dmaGblRegId = DMA_GBLCNTA | DMA_GBLADDRA;

/* Select Global Address Reload Register A or Global Address Register A */
/*--------------------------------------------------------------------------*/
/* External functions and function prototypes */

void init_mcbsp0_dma(void);   /* Function prototypes */
void set_interrupts_dma(void);

/* Include the vector table to call the IRQ ISRs hookup */
extern far void vectors();
```

```
/*----------------------------------------------------------------------------*/
/* main()                                                                     */
/*----------------------------------------------------------------------------*/
void main(void)
{
/* Declaration of local variables */
static int element_count, xfer_type;

static Uint32 dmaInbuff[BUFFER_SIZE];  /* define DSP internal mem buffers  */
static Uint32 dmaOutbuff[BUFFER_SIZE];

IRQ_setVecs(vectors); /* point to the IRQ vector table */

element_count = ELEMENT_COUNT;
xfer_type = XFER_TYPE;

/* initialize the CSL library */
CSL_init();

init_mcbsp0_dma();

/* Enable sample rate generator GRST=1 */
MCBSP_enableSrgr(hMcbsp0);  /* Handle to SRGR */

switch (xfer_type) {
case DMA_XFER:

DMA_reset(INV);         /* reset all DMA channels */
/*---------------------------------------------------------------------------*/
/* DMA channel 1 config structure                                            */
/*---------------------------------------------------------------------------*/
/* Channel 1 receives the data */
/* Establish Global Register Values in the following configuration structure */
dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);
DMA_globalConfigArgs(dmaGblRegMsk, /* DMA global address registers A to D */
      DMA_GBLADDR_GBLADDR_OF(MCBSP_ADDRH(hMcbsp0, DRR)), /* McBSP DRR0    */
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLIDX_DEFAULT,
      DMA_GBLIDX_DEFAULT,
      DMA_GBLCNT_RMK(   /* config DMA global count relaod register A  */
         DMA_GBLCNT_FRMCNT_DEFAULT,
         DMA_GBLCNT_ELECNT_OF(element_count) /* element count reload */
      ),
      DMA_GBLCNT_DEFAULT
);  /* end of DMA global control register configuration structure */

hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);  /* Handle to DMA channel 1 */
    DMA_configArgs(hDma1,
       DMA_PRICTL_RMK(
           DMA_PRICTL_DSTRLD_DEFAULT,
           DMA_PRICTL_SRCRLD_DEFAULT,
           DMA_PRICTL_EMOD_HALT, /* DMA cahnnel pauses during emulation halt */
           DMA_PRICTL_FS_DEFAULT,
           DMA_PRICTL_TCINT_ENABLE, /* TCINT =1                              */
           DMA_PRICTL_PRI_DMA,      /* DMA high priority                     */
           DMA_PRICTL_WSYNC_REVT0,  /* Set synchronization event REVT0=01101 */
           DMA_PRICTL_RSYNC_XEVT0,  /* Set synchronization event XEVT0=01100 */
           DMA_PRICTL_INDEX_DEFAULT,
           DMA_PRICTL_CNTRLD_A,  /* Reload with DMA global reload countrer A */
           DMA_PRICTL_SPLIT_A,   /* Split channel mode enabled, use GBLADDRA */
           DMA_PRICTL_ESIZE_32BIT,  /* Element size 32 bits                  */
           DMA_PRICTL_DSTDIR_INC,   /* Increment dest. by element size       */
           DMA_PRICTL_SRCDIR_INC,   /* Increment source by element size      */
           DMA_PRICTL_START_DEFAULT
           ),

        DMA_SECCTL_RMK(
           DMA_SECCTL_WSPOL_NA,  /* only available for 6202 and 6203 devices */
           DMA_SECCTL_RSPOL_NA,  /* only available for 6202 and 6203 devices */
           DMA_SECCTL_FSIG_NA,   /* only available for 6202 and 6203 devices */
           DMA_SECCTL_DMACEN_DEFAULT,
           DMA_SECCTL_WSYNCCLR_DEFAULT,
           DMA_SECCTL_WSYNCSTAT_DEFAULT,
           DMA_SECCTL_RSYNCCLR_DEFAULT,
           DMA_SECCTL_RSYNCSTAT_DEFAULT,
           DMA_SECCTL_WDROPIE_DEFAULT,
           DMA_SECCTL_WDROPCOND_DEFAULT,
           DMA_SECCTL_RDROPIE_DEFAULT,
           DMA_SECCTL_RDROPCOND_DEFAULT,
           DMA_SECCTL_BLOCKIE_ENABLE, // BLOCK IE=1 enables DMA channel int
           DMA_SECCTL_BLOCKCOND_DEFAULT,
           DMA_SECCTL_LASTIE_DEFAULT,
           DMA_SECCTL_LASTCOND_DEFAULT,
           DMA_SECCTL_FRAMEIE_DEFAULT,
           DMA_SECCTL_FRAMECOND_DEFAULT,
           DMA_SECCTL_SXIE_DEFAULT,
           DMA_SECCTL_SXCOND_DEFAULT
           ),

        DMA_SRC_RMK((Uint32)dmaOutbuff), // Data src internal DSP data memory
        DMA_DST_RMK((Uint32)dmaInbuff),  // Data dest internal DSP data memory
        DMA_XFRCNT_RMK(
           DMA_XFRCNT_FRMCNT_DEFAULT,
           DMA_XFRCNT_ELECNT_OF(element_count)  /* set recv element count */
           )
);

/* Initialize the interrupt(s)                                   */
/*   Enable the interrupt after the DMA channels are opened as    */
/* the DMA_OPEN_RESET clears and disables the channel interrupt   */
/* once specified and clears the corresponding interrupt bits     */
/* in the IER.                                                   */
set_interrupts_dma();

DMA_start(hDma1);     /* Start DMA channel 1 */
} /* end of switch here */

/* Enable McBSP channel */
MCBSP_enableRcv(hMcbsp0); /* McBSP port 0 as the transmitter         */

/* To flag an interrupt to the CPU when DMA transfer/receive is done */
while (!transfer_done);

MCBSP_close(hMcbsp0);  /* close McBSP port */

DMA_close(hDma1);  /* close DMA channels */
} /* end main, program ends here */

/*---------------------------------------------------------------------------*/
/* init_mcbsp0_dma()                                                        */
/*---------------------------------------------------------------------------*/
/* MCBSP Config structure */
/* Setup the MCBSP_0 for data receive */
```

```
void
init_mcbsp0_dma(void)
{
MCBSP_Config mcbspCfg0 = {
     MCBSP_SPCR_RMK(
          MCBSP_SPCR_FRST_DEFAULT,  // All fields in SPCR set to default values
          MCBSP_SPCR_GRST_DEFAULT,
          MCBSP_SPCR_XINTM_DEFAULT,
          MCBSP_SPCR_XSYNCERR_DEFAULT,
          MCBSP_SPCR_XRST_DEFAULT,
          MCBSP_SPCR_DLB_DEFAULT,
          MCBSP_SPCR_RJUST_DEFAULT,
          MCBSP_SPCR_CLKSTP_DEFAULT,
          MCBSP_SPCR_RINTM_DEFAULT,
          MCBSP_SPCR_RSYNCERR_DEFAULT,
          MCBSP_SPCR_RRST_DEFAULT
          ),

     MCBSP_RCR_RMK(
           MCBSP_RCR_RPHASE_SINGLE,   /* Single phase receive frame */
           MCBSP_RCR_RFRLEN2_DEFAULT,
           MCBSP_RCR_RWDLEN2_DEFAULT,
           MCBSP_RCR_RCOMPAND_DEFAULT,
           MCBSP_RCR_RFIG_DEFAULT,
           MCBSP_RCR_RDATDLY_1BIT,    /* 1-bit receive data delay   */
           MCBSP_RCR_RFRLEN1_DEFAULT,
           MCBSP_RCR_RWDLEN1_DEFAULT
           ),

     MCBSP_XCR_RMK(
           MCBSP_XCR_XPHASE_DEFAULT, // All fields in XCR set to default values
           MCBSP_XCR_XFRLEN2_DEFAULT,
           MCBSP_XCR_XWDLEN2_DEFAULT,
           MCBSP_XCR_XCOMPAND_DEFAULT,
           MCBSP_XCR_XFIG_DEFAULT,
           MCBSP_XCR_XDATDLY_DEFAULT,
           MCBSP_XCR_XFRLEN1_DEFAULT,
           MCBSP_XCR_XWDLEN1_DEFAULT
           ),
     MCBSP_SRGR_RMK(
           MCBSP_SRGR_GSYNC_DEFAULT, //All fields in SRGR set to default values
           MCBSP_SRGR_CLKSP_DEFAULT,
           MCBSP_SRGR_CLKSM_DEFAULT,
           MCBSP_SRGR_FSGM_DEFAULT,
           MCBSP_SRGR_FPER_DEFAULT,
           MCBSP_SRGR_FWID_DEFAULT,
           MCBSP_SRGR_CLKGDV_DEFAULT
           ),

     MCBSP_MCR_RMK(
           MCBSP_MCR_XPBBLK_DEFAULT, // All fields in MCR set to default values
           MCBSP_MCR_XPABLK_DEFAULT,
           MCBSP_MCR_XMCM_DEFAULT,
           MCBSP_MCR_RPBBLK_DEFAULT,
           MCBSP_MCR_RPABLK_DEFAULT,
           MCBSP_MCR_RMCM_DEFAULT
           ),

     MCBSP_RCER_RMK(
           MCBSP_RCER_RCEB_DEFAULT, // All fields in RCER set to default values
           MCBSP_RCER_RCEA_DEFAULT
           ),

     MCBSP_XCER_RMK(
           MCBSP_XCER_XCEB_DEFAULT, // All fields in XCER set to default values
           MCBSP_XCER_XCEA_DEFAULT
           ),

     MCBSP_PCR_RMK(
           MCBSP_PCR_XIOEN_DEFAULT,
           MCBSP_PCR_RIOEN_DEFAULT,
           MCBSP_PCR_FSXM_DEFAULT,
           MCBSP_PCR_FSRM_DEFAULT,
           MCBSP_PCR_CLKXM_DEFAULT,
           MCBSP_PCR_CLKRM_DEFAULT,
           MCBSP_PCR_CLKSSTAT_DEFAULT,
           MCBSP_PCR_DXSTAT_DEFAULT,
           MCBSP_PCR_FSXP_DEFAULT,
           MCBSP_PCR_FSRP_DEFAULT,
           MCBSP_PCR_CLKXP_DEFAULT,
           MCBSP_PCR_CLKRP_DEFAULT
```

```
           )
};
hMcbsp0 = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET); /* McBSP port 0 */
MCBSP_config(hMcbsp0, &mcbspCfg0);
}

/*----------------------------------------------------------------------------*/
/* set_interrupts_dma()                                                       */
/*----------------------------------------------------------------------------*/
void                           /* Set the interrupts */
set_interrupts_dma(void)
{
     IRQ_nmiEnable();
     IRQ_globalEnable();
     IRQ_disable(IRQ_EVT_DMAINT1);  /* INT09 */
     IRQ_clear(IRQ_EVT_DMAINT1);
     IRQ_enable(IRQ_EVT_DMAINT1);
     return;
}

/*----------------------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISR                                        */
/*----------------------------------------------------------------------------*/
interrupt void     /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)      /* DMA ch1                          */
{
     transfer_done = TRUE;
     return;
}
/*-----------------------End of splitmode_dma.c------------------------------*/
```

A.6 Frame-Synchronized Data Transfer Example Code

```
/*----------------------------------------------------------------------------*/
/*  framesync_dma.c V1.00                                                     */
/*  Copyright (c) 2001 Texas Instruments Incorporated                         */
/*----------------------------------------------------------------------------*/
/*
  7/20/01
  Original by: Dave Bell
  Modified by: Vassos S. Soteriou
  framesync_dma.c:
    This program sets up the DMA control registers to perform frame-synchronized
  data transfers between internal data memory and the external AFE.  DMA channel 1
  is setup to transfer data from an external AFE  data buffer (AFEInbuff) to
  theinternal DSP data memory (dmaInbuff). This channel uses interrupt 9 to stop
  thedata transfer process when the transfer is complete by setting a flag to the
  CPU.DMA channel 2 is setup to transfer data from the internal DSP data memory
  (dmaOutbff) to an external AFE buffer (AFEOutbuff). This channel uses interrupt
  11 to stop the data transfer process when the transfer is complete by setting a
  flag to the CPU.
     Note that the user can modify this program to fit a specific application by
  defining the mapping of the external source/destination buffer (for instance,
  CE0 EMIF space can be used) and by selecting any of the DMA channels to perform
  the transfers along with the corresponding interrupts.
     The sample code is based on TI’s CSL 2.0. See the TMS320C6000 Chip Support
  Library API User’s Guide (SPRA401) for further information.
*/

/* Chip definition, change this accordingly */
#define CHIP_6202 1

/* Include files */
#include <c6x.h>
#include <csl.h>         /* CSL library   */
#include <csl_dma.h>     /* DMA_SUPPORT   */
#include <csl_irq.h>     /* IRQ_SUPPORT   */

/* Define constants */
#define FALSE 0
#define TRUE 1

#define DMA_XFER 8
#define XFER_TYPE DMA_XFER
#define BUFFER_SIZE 256   /* set element_count =< buffer_size */
#define ELEMENT_COUNT 32

/* Global variables used in interrupt ISRs */
volatile int recv_done = FALSE;
volatile int xmit_done = FALSE;
```

```
/*--------------------------------------------------------------------------*/
/* Declare CSL objects */
DMA_Handle hDma1;           /* Handles for DMA           */
DMA_Handle hDma2;
Uint32 dmaGblRegMsk;        /* DMA Global Register Mask  */
Uint32 dmaGblRegId = DMA_GBLCNTA | DMA_GBLADDRB | DMA_GBLADDRC;
/* Select Global Count Reload Register A or Global Address Register B or C  */
/*--------------------------------------------------------------------------*/

/* External functions and function prototypes */
void set_interrupts_dma(void); /* Function prototypes */

/* Include the vector table to call the IRQ ISRs hookup */
extern far void vectors();

/*-------------------------------------------------------------------------*/
/* main()                                                                  */
/*-------------------------------------------------------------------------*/
void main(void)
{
/* Declaration of local variables */
static int element_count, xfer_type;

static Uint32 dmaInbuff[BUFFER_SIZE];  // Define interan DSP data mem arrays
static Uint32 dmaOutbuff[BUFFER_SIZE];

#pragma DATA_ALIGN (AFEInbuff, 128); // Asign AFEbuff to alignment boundary

/* Allocate space for AFEInbuf in MEMORY/SECTIONS section of the link    */
/*   command file, *.cmd. Note that the user also has to define the      */
/*   origin and length of this ext memory in the cmd file                */
#pragma DATA_SECTION(AFEInbuff, ”external_data”);

#pragma DATA_ALIGN (AFEOutbuff, 128); // Asign AFEOutbuff to alignment bound

/* Allocate space for AFEOutbuf in MEMORY/SECTIONS section of the link   */
/* command file, *.cmd. Note that the user also has to define the        */
/* origin and length of this ext memory in the cmd file                  */
#pragma DATA_SECTION(AFEOutbuff, ”external_data”);

static Uint32 AFEInbuff[BUFFER_SIZE];  /*define external memory src buffer   */
static Uint32 AFEOutbuff[BUFFER_SIZE]; /*define external memory dest buffer  */

IRQ_setVecs(vectors); /* point to the IRQ vector table */
```

```
element_count = ELEMENT_COUNT;
xfer_type = XFER_TYPE;

/* initialize the CSL library */
CSL_init();

switch (xfer_type) {
case DMA_XFER:

DMA_reset(INV);  /* reset all DMA channels */

/*---------------------------------------------------------------------------*/
/* DMA channels 1 & 2 config structures                                      */
/*---------------------------------------------------------------------------*/
/* Establish Global Register Values in the following configuration structure */
dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);

DMA_globalConfigArgs(dmaGblRegMsk,
     DMA_GBLADDR_GBLADDR_DEFAULT, /* DMA global address registers A to D  */
     DMA_GBLADDR_GBLADDR_OF((unsigned int) dmaInbuff),  // point to dmaInbuff
     DMA_GBLADDR_GBLADDR_OF((unsigned int) dmaOutbuff), // point to dmaOutbuff
     DMA_GBLADDR_GBLADDR_DEFAULT,

     /* DMA global index registers A to B   */
     DMA_GBLIDX_RMK(
         DMA_GBLIDX_FRMIDX_DEFAULT,
         DMA_GBLIDX_ELEIDX_DEFAULT
         ),
     DMA_GBLIDX_DEFAULT,

     /* config DMA global count reload register A  */
     DMA_GBLCNT_RMK(
         DMA_GBLCNT_FRMCNT_DEFAULT,
         DMA_GBLCNT_ELECNT_OF(element_count)
     ),
      DMA_GBLCNT_DEFAULT
);  /* end of DMA global control register configuration structure */

/* DMA channel 1 */
hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);  /* Handle to DMA channel 1 */
DMA_configArgs(hDma1,
     DMA_PRICTL_RMK(
         DMA_PRICTL_DSTRLD_B,   /* Use DMA Global Address Reg. B for reload */
```

```
        DMA_PRICTL_SRCRLD_DEFAULT,
        DMA_PRICTL_EMOD_HALT,  /* DMA cahnnel pauses during emulation halt */
        DMA_PRICTL_FS_RSYNC,   /* RSYNC event used to synch entire frame   */
        DMA_PRICTL_TCINT_ENABLE,     /* TCINT =1                           */
        DMA_PRICTL_PRI_DMA,          /* DMA high priority                  */
        DMA_PRICTL_WSYNC_DEFAULT,
        DMA_PRICTL_RSYNC_EXTINT4,    /* Set sync. event EXT_INT4=00100     */
        DMA_PRICTL_INDEX_DEFAULT,
        DMA_PRICTL_CNTRLD_A,    /* Reload with DMA global reload countrer A  */
        DMA_PRICTL_SPLIT_DEFAULT,
        DMA_PRICTL_ESIZE_32BIT,   /* Element size 32 bits                    */
        DMA_PRICTL_DSTDIR_INC,    /* Increment desctination by el. Size */
        DMA_PRICTL_SRCDIR_DEFAULT,
        DMA_PRICTL_START_DEFAULT
        ),

    DMA_SECCTL_RMK(
        DMA_SECCTL_WSPOL_NA,    /* only available for 6202 and 6203 devices */
        DMA_SECCTL_RSPOL_NA,    /* only available for 6202 and 6203 devices */
        DMA_SECCTL_FSIG_NA,     /* only available for 6202 and 6203 devices */
        DMA_SECCTL_DMACEN_DEFAULT,
        DMA_SECCTL_WSYNCCLR_DEFAULT,
        DMA_SECCTL_WSYNCSTAT_DEFAULT,
        DMA_SECCTL_RSYNCCLR_DEFAULT,
        DMA_SECCTL_RSYNCSTAT_DEFAULT,
        DMA_SECCTL_WDROPIE_DEFAULT,
        DMA_SECCTL_WDROPCOND_DEFAULT,
        DMA_SECCTL_RDROPIE_DEFAULT,
        DMA_SECCTL_RDROPCOND_DEFAULT,
        DMA_SECCTL_BLOCKIE_DEFAULT,
        DMA_SECCTL_BLOCKCOND_DEFAULT,
        DMA_SECCTL_LASTIE_DEFAULT,
        DMA_SECCTL_LASTCOND_DEFAULT,
        DMA_SECCTL_FRAMEIE_ENABLE, /* Frame condition enables DMA ch. inter.*/
        DMA_SECCTL_FRAMECOND_DEFAULT,
        DMA_SECCTL_SXIE_DEFAULT,
        DMA_SECCTL_SXCOND_DEFAULT
        ),
   DMA_SRC_RMK((Uint32)AFEInbuff),  /* source data from the AFEInbuff */
   DMA_DST_RMK((Uint32)dmaInbuff),  /* write data to the dmaInbuff    */
   DMA_XFRCNT_RMK(
       DMA_XFRCNT_FRMCNT_OF(1),             /* transfer one frame     */
       DMA_XFRCNT_ELECNT_OF(element_count)  /* set rcev element count */
       )
);

/* DMA channel 2 */
hDma2= DMA_open(DMA_CHA2, DMA_OPEN_RESET);  /* Handle to DMA channel 2 */
DMA_configArgs(hDma2,
     DMA_PRICTL_RMK(
         DMA_PRICTL_DSTRLD_DEFAULT,
         DMA_PRICTL_SRCRLD_C,    /* Use DMA Global Address Reg. C as reload */
         DMA_PRICTL_EMOD_HALT, /* DMA cahnnel pauses during emulation halt  */
         DMA_PRICTL_FS_RSYNC,  /* RSYNC event to synchronize entire frame   */
         DMA_PRICTL_TCINT_ENABLE,  /* TCINT = 1                             */
         DMA_PRICTL_PRI_DMA,       /* DMA high priority                     */
         DMA_PRICTL_WSYNC_DEFAULT,
         DMA_PRICTL_RSYNC_DMAINT1, /* Set sync. event DMA_INT1=01001        */
         DMA_PRICTL_INDEX_DEFAULT,
         DMA_PRICTL_CNTRLD_A,    /* Reload with DMA global reload countrer A */
         DMA_PRICTL_SPLIT_DEFAULT,
         DMA_PRICTL_ESIZE_32BIT,  /* Element size 32 bits                    */
         DMA_PRICTL_DSTDIR_DEFAULT,
         DMA_PRICTL_SRCDIR_INC,   /* Increment destination by element size   */
         DMA_PRICTL_START_DEFAULT
         ),

     DMA_SECCTL_RMK(
           DMA_SECCTL_WSPOL_NA, /* only available for 6202 and 6203 devices */
           DMA_SECCTL_RSPOL_NA, /* only available for 6202 and 6203 devices */
           DMA_SECCTL_FSIG_NA,  /* only available for 6202 and 6203 devices */
           DMA_SECCTL_DMACEN_DEFAULT,
           DMA_SECCTL_WSYNCCLR_DEFAULT,
           DMA_SECCTL_WSYNCSTAT_DEFAULT,
           DMA_SECCTL_RSYNCCLR_DEFAULT,
           DMA_SECCTL_RSYNCSTAT_DEFAULT,
           DMA_SECCTL_WDROPIE_DEFAULT,
           DMA_SECCTL_WDROPCOND_DEFAULT,
           DMA_SECCTL_RDROPIE_DEFAULT,
           DMA_SECCTL_RDROPCOND_DEFAULT,
           DMA_SECCTL_BLOCKIE_DEFAULT,
           DMA_SECCTL_BLOCKCOND_DEFAULT,
           DMA_SECCTL_LASTIE_DEFAULT,
           DMA_SECCTL_LASTCOND_DEFAULT,
           DMA_SECCTL_FRAMEIE_ENABLE, // Frame cond enables DMA ch. inter.*/
           DMA_SECCTL_FRAMECOND_DEFAULT,
           DMA_SECCTL_SXIE_DEFAULT,
           DMA_SECCTL_SXCOND_DEFAULT
           ),

     DMA_SRC_RMK((Uint32)dmaOutbuff), /* source data from the dmaOutbuff */
    DMA_DST_RMK((Uint32)AFEOutbuff), /* write data to the AFEOutbuff    */
    DMA_XFRCNT_RMK(
        DMA_XFRCNT_FRMCNT_OF(1),            /* transfer one frame     */
        DMA_XFRCNT_ELECNT_OF(element_count) /* set rcev element count */
        )
);

/* Initialize the interrupt(s)                                  */
/*   Enable the interrupt after the DMA channels are opened as   */
/* the DMA_OPEN_RESET clears and disables the channel interrupt  */
/* once specified and clears the corresponding interrupt bits    */
/* in the IER.                                                  */
set_interrupts_dma();

DMA_start(hDma1);     /* Start DMA channels 1 & 2 */
DMA_start(hDma2);

} /* end of switch here */
/* To flag an interrupt to the CPU when DMA transfer/receive is done */
while (!recv_done || !xmit_done);

DMA_close(hDma1);  /* close DMA channels after data trasnfer is complete */
DMA_close(hDma2);
} /* end main, program ends here */
```

```
/*----------------------------------------------------------------------------*/
/* set_interrupts_dma()                                                       */
/*----------------------------------------------------------------------------*/
void                           /* Set the interrupts         */
set_interrupts_dma(void)
{
    IRQ_nmiEnable();
    IRQ_globalEnable();
    IRQ_disable(IRQ_EVT_DMAINT1);  /* INT09 */
    IRQ_disable(IRQ_EVT_DMAINT2);  /* INT11 */
    IRQ_clear(IRQ_EVT_DMAINT1);
    IRQ_clear(IRQ_EVT_DMAINT2);
    IRQ_enable(IRQ_EVT_DMAINT1);
    IRQ_enable(IRQ_EVT_DMAINT2);
    return;
}

/*----------------------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISRs                                       */
/*----------------------------------------------------------------------------*/
interrupt void     /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)      /* DMA ch1                          */
{
   recv_done = TRUE;
   return;
}

interrupt void     /* vecs.asm hooks this up to IRQ 11  */
c_int11(void)      /* DMA ch2                           */
{
   xmit_done = TRUE;
   return;
}

/*----------------------End of framesync_dma.c-------------------------------*/
```

A.7 Circular Buffering Transfer Example Code

```
/*------------------------------------------------------------*/
/*  circularbuff_dma.c V1.00                                  */
/*  Copyright (c) 2001 Texas Instruments Incorporated         */
/*------------------------------------------------------------*/

/*************************************************************
 9/2/01:
 Original by: Dave Bell
 Modified by: Michael Haag and Vassos S. Soteriou

 Description: This program sets up the DMA control registers to
  perform data transfers between internal data memory and the
  external AFE, using circular buffering for both input & output
  data.  The sample code is based on TI's CSL 2.0. See the
  TMS320C6000 Chip Support Library API User's Guide (SPRU401)
  for further information.
*************************************************************/
/* Chip definition, change this accordingly */
#define CHIP_6202 1

/* Include files */
#include <stdio.h>
#include <c6x.h>
#include <csl.h>          /* CSL library   */
#include <csl_dma.h>      /* DMA_SUPPORT   */
#include <csl_irq.h>      /* IRQ_SUPPORT   */

/* Define constants */
#define FALSE         0
#define TRUE          1

#define DMA_XFER       8
#define XFER_TYPE      DMA_XFER

#define ELEMENT_COUNT  1024
#define FRAME_COUNT    2
#define ELEMENT_SIZE   4  /* number of bytes/element, 1,2,4   */
#define BUFFER_SIZE    ELEMENT_COUNT
#define LOOPS          3  /* number of needed transfers       */
```

```
/* Global variables used in interrupt ISRs */
volatile int recv_done = FALSE;
volatile int xmit_done = FALSE;
volatile int count_recv = 0;
volatile int count_xmit = 0;

/* Define the number of times to loop transfering */
unsigned int loops = LOOPS;

/* Set up buffers in internal memory to simulate the DMA transfers */
Uint32 interInbuff[BUFFER_SIZE]; // intermediate buffer
Uint32 dmaInbuff[BUFFER_SIZE];   // buffer for DMA supporting devices
Uint32 dmaOutbuff[BUFFER_SIZE];
Uint32 interOutbuff[BUFFER_SIZE]; // to simulate the external buffer
Uint32 y;

/* Declare CSL objects */
DMA_Handle hDma1;           /* Handle for DMA      */
DMA_Handle hDma2;
Uint32 dmaGblRegMsk;        /* DMA Global Register Mask */
/* Select Global Count Reload Register A or Global Index Register A  */
Uint32 dmaGblRegId = DMA_GBLCNTA | DMA_GBLIDXA;

/* External functions and function prototypes */
void set_interrupts_dma(void);
void init_dma(void);

/* Include the vector table to call the IRQ ISRs hookup */
extern far void vectors();

/************************************************************/
/*----------------------------------------------------------*/
/* BEGIN main()                                             */
/*----------------------------------------------------------*/
/************************************************************/
void main(void)
{
/* Declaration of local variables */
static int xfer_type, frame_index, element_index;

IRQ_setVecs(vectors); /* point to the IRQ vector table */

xfer_type = XFER_TYPE;
```

```
/* Calculate the index values, as well as the ESIZE, based on the
   number of elements per frame (element_count), the number of
   frames per block (frmame_count), and the number of bytes in
   each element (element_size)              */
element_index = ELEMENT_SIZE;
frame_index = –((ELEMENT_COUNT – 1) * ELEMENT_SIZE);

/* initialize the CSL library */
CSL_init();

/**** START SWITCH HERE ****/
switch (xfer_type) {
case DMA_XFER:

DMA_reset(INV);         /* reset all DMA channels */

/* Initialize DMA global register mask value */
dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);

/* Sets up the DMA global registers */
DMA_globalConfigArgs(dmaGblRegMsk,

    /* DMA global address regs A to D */
    DMA_GBLADDR_GBLADDR_DEFAULT,
    DMA_GBLADDR_GBLADDR_DEFAULT,
    DMA_GBLADDR_GBLADDR_DEFAULT,
    DMA_GBLADDR_GBLADDR_DEFAULT,

    /* DMA global index registers A to B   */
    DMA_GBLIDX_RMK(
     DMA_GBLIDX_FRMIDX_OF(frame_index),
        DMA_GBLIDX_ELEIDX_OF(element_index)
     ),
     DMA_GBLIDX_DEFAULT,

    /* config DMA global count reload register A  */
    DMA_GBLCNT_RMK(
     DMA_GBLCNT_FRMCNT_OF(FRAME_COUNT),
     DMA_GBLCNT_ELECNT_OF(ELEMENT_COUNT)
     ),
     DMA_GBLCNT_DEFAULT
);
/* end of DMA global control register configuration structure */
init_dma();         /* call function to initialize DMA channel 1 and 2 */

/* Initialize the interrupt(s):                                      */
/*    Enable the interrupt after the DMA channels are opened as      */
/*  the DMA_OPEN_RESET clears and disables all channel interrupts    */
/*  previously specified and clears the corresponding interrupt      */
/*  bits in the IER. This is not applicable for the EDMA channel     */
/*  open case                                                        */
set_interrupts_dma();

/* Start DMA channel 1 with autoinitialization */
DMA_autoStart(hDma1);

}
/**** END SWITCH HERE ****/

/* Continue Circular buffer transfer between buffers until number */
/*   of transfers needed have been performed         */
while (!recv_done && !xmit_done);

DMA_close(hDma1);
DMA_close(hDma2);

}
/*********************************************************************/
/*-----------------------------------------------------------------*/
/*                       END MAIN (PROGRAM ENDS HERE)               */
/*-----------------------------------------------------------------*/
/*********************************************************************/

/*-----------------------------------------------------------*/
/* initialize DMA channels 1 and 2                           */
/*-----------------------------------------------------------*/
void
init_dma(void)
{
     Uint32 element_size;

     /* covert number of bytes/element to CSL HAL MACRO conversion */
     if (ELEMENT_SIZE == 1) element_size = 2;       // 8BIT element
     else if (ELEMENT_SIZE == 2) element_size = 1;  // 16BIT element
     else element_size = 0;                         // 32BIT element
```

```
/* Configure DMA channel 1 */
hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);
DMA_configArgs(hDma1,
 DMA_PRICTL_RMK(
     DMA_PRICTL_DSTRLD_DEFAULT,
     DMA_PRICTL_SRCRLD_DEFAULT,
     DMA_PRICTL_EMOD_DEFAULT,
     DMA_PRICTL_FS_RSYNC,   // FS = 1
     DMA_PRICTL_TCINT_ENABLE,  // TCINT =1
     DMA_PRICTL_PRI_CPU,
     DMA_PRICTL_WSYNC_DEFAULT,
     DMA_PRICTL_RSYNC_EXTINT4,  /* Set sync. event EXT_INT4=00100      */
     DMA_PRICTL_INDEX_A,
     DMA_PRICTL_CNTRLD_A, // Reload with global reload counter A
     DMA_PRICTL_SPLIT_DEFAULT,
     DMA_PRICTL_ESIZE_OF(element_size), // Element size defined by user
     DMA_PRICTL_DSTDIR_IDX,  //Adjust dest. using DMA Global Index Reg. A
     DMA_PRICTL_SRCDIR_NONE, // Increase source by element_size
     DMA_PRICTL_START_DEFAULT
     ),

  DMA_SECCTL_RMK(
    DMA_SECCTL_WSPOL_NA,  /* only available for 6202/6203 devices */
    DMA_SECCTL_RSPOL_NA,  /* only available for 6202/6203 devices */
    DMA_SECCTL_FSIG_NA,   /* only available for 6202/6203 devices */
    DMA_SECCTL_DMACEN_DEFAULT,
    DMA_SECCTL_WSYNCCLR_DEFAULT,
    DMA_SECCTL_WSYNCSTAT_DEFAULT,
    DMA_SECCTL_RSYNCCLR_DEFAULT,
    DMA_SECCTL_RSYNCSTAT_DEFAULT,
    DMA_SECCTL_WDROPIE_DEFAULT,
    DMA_SECCTL_WDROPCOND_DEFAULT,
    DMA_SECCTL_RDROPIE_DEFAULT,
    DMA_SECCTL_RDROPCOND_DEFAULT,
    DMA_SECCTL_BLOCKIE_DISABLE,
    DMA_SECCTL_BLOCKCOND_DEFAULT,
    DMA_SECCTL_LASTIE_DEFAULT,
    DMA_SECCTL_LASTCOND_DEFAULT,
    DMA_SECCTL_FRAMEIE_ENABLE,   /* Enables DMA ch. inter.*/
    DMA_SECCTL_FRAMECOND_DEFAULT,
    DMA_SECCTL_SXIE_DEFAULT,
        DMA_SECCTL_SXCOND_DEFAULT
        ),

    /* Set up source – data from the interInbuff   */
     DMA_SRC_RMK((Uint32)interInbuff),

    /* Set up destination – transfer to the dmaInbuff    */
     DMA_DST_RMK((Uint32)dmaInbuff),

    /* Set up transfer count register         */
     DMA_XFRCNT_RMK(
           DMA_XFRCNT_FRMCNT_OF(FRAME_COUNT),
           DMA_XFRCNT_ELECNT_OF(ELEMENT_COUNT)
       )
  );

  /* Configure DMA channel 2 */
  hDma2= DMA_open(DMA_CHA2, DMA_OPEN_RESET);  /* Handle to DMA channel 2 */
  DMA_configArgs(hDma2,
    DMA_PRICTL_RMK(
        DMA_PRICTL_DSTRLD_DEFAULT,
        DMA_PRICTL_SRCRLD_DEFAULT,
        DMA_PRICTL_EMOD_DEFAULT,
        DMA_PRICTL_FS_RSYNC,     //FS = 1, Synchronize entire frame xfer
        DMA_PRICTL_TCINT_DISABLE,
        DMA_PRICTL_PRI_CPU,
        DMA_PRICTL_WSYNC_DEFAULT,
        DMA_PRICTL_RSYNC_DMAINT1, // Sync. event is DMA_INT1=01001
        DMA_PRICTL_INDEX_A,
       DMA_PRICTL_CNTRLD_A,   // Reload with DMA global reload counter A */
        DMA_PRICTL_SPLIT_DEFAULT,
        DMA_PRICTL_ESIZE_OF(element_size), // Element size defined by user
        DMA_PRICTL_DSTDIR_NONE,
        DMA_PRICTL_SRCDIR_IDX,  // Adjust src using DMA Global Index Reg. A
        DMA_PRICTL_START_DEFAULT
        ),

    DMA_SECCTL_RMK(
        DMA_SECCTL_WSPOL_NA, /* only available for 6202 and 6203 devices */
        DMA_SECCTL_RSPOL_NA, /* only available for 6202 and 6203 devices */
        DMA_SECCTL_FSIG_NA,  /* only available for 6202 and 6203 devices */
        DMA_SECCTL_DMACEN_DEFAULT,
        DMA_SECCTL_WSYNCCLR_DEFAULT,
        DMA_SECCTL_WSYNCSTAT_DEFAULT,
        DMA_SECCTL_RSYNCCLR_DEFAULT,
        DMA_SECCTL_RSYNCSTAT_DEFAULT,
        DMA_SECCTL_WDROPIE_DEFAULT,
        DMA_SECCTL_WDROPCOND_DEFAULT,
        DMA_SECCTL_RDROPIE_DEFAULT,
        DMA_SECCTL_RDROPCOND_DEFAULT,
        DMA_SECCTL_BLOCKIE_DISABLE,
        DMA_SECCTL_BLOCKCOND_DEFAULT,
        DMA_SECCTL_LASTIE_DEFAULT,
        DMA_SECCTL_LASTCOND_DEFAULT,
        DMA_SECCTL_FRAMEIE_ENABLE,
        DMA_SECCTL_FRAMECOND_DEFAULT,
        DMA_SECCTL_SXIE_DEFAULT,
        DMA_SECCTL_SXCOND_DEFAULT
        ),

      /* Set up source – data from the dmaOutbuff    */
      DMA_SRC_RMK((Uint32)dmaOutbuff),

     /* Set up destination – transfer to the interOutbuff */
     DMA_DST_RMK((Uint32)interOutbuff),

     /* Set up transfer count register       */
     DMA_XFRCNT_RMK(
         DMA_XFRCNT_FRMCNT_OF(FRAME_COUNT),
         DMA_XFRCNT_ELECNT_OF(ELEMENT_COUNT)
         )
     );
}

/*-------------------------------------------------------------*/
/* set_interrupts_dma()                                        */
/*-------------------------------------------------------------*/
#if (DMA_SUPPORT)
void                           /* Set the interrupts */
set_interrupts_dma(void)       /* if the device supports DMA */
{
     IRQ_nmiEnable();
     IRQ_globalEnable();
     IRQ_disable(IRQ_EVT_DMAINT1);  /* INT09 */
     IRQ_disable(IRQ_EVT_DMAINT2);  /* INT11 */
     IRQ_clear(IRQ_EVT_DMAINT1);
     IRQ_clear(IRQ_EVT_DMAINT2);
     IRQ_enable(IRQ_EVT_DMAINT1);
     IRQ_enable(IRQ_EVT_DMAINT2);
     return;
}
#endif

/*------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISR                        */
/*------------------------------------------------------------*/
interrupt void     /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)      /* DMA ch1                          */
{
     ++count_recv;

    /* Add condition to trigger the end of the transfer from the  */
    /*  external memory to the DMA and vice versa.  In this code, */
    /*  the trigger is a set number of transfers, equal to the    */
    /*  variable ”loops”.                                         */
    if (count_recv == (loops))
    recv_done = TRUE;

    /* Since output buffer initially empty, Channel 2 does not   */
    /* start until after the first frame is transfered      */
    DMA_autoStart(hDma2);

    /* Manually clear the FRAME COND bit following each frame transfer   */
    DMA_FSET(SECCTL1, FRAMECOND, 0);

    return;
}

/* Ch2 ISR below entered ONLY if TCINT field in DMA PRICTL2 is enabled */
interrupt void     /* vecs.asm hooks this up to IRQ 11  */
c_int11(void)      /* DMA ch2                           */
{
     ++count_xmit;
     if (count_xmit == (loops)) // – 1) )
          xmit_done = TRUE;
     /* Manually clear the FRAME COND bit following each frame transfer */
     DMA_FSET(SECCTL2, FRAMECOND, 0);
     return;
}

/*-------------------End of circularbuff_dma.c---------------*/
```

A.8 Ping-Pong Transfer Example Code

```
/*------------------------------------------------------------------*/
/*  ping_pong_dma.c V1.00                                           */
/*  Copyright (c) 2001 Texas Instruments Incorporated               */
/*------------------------------------------------------------------*/
/*******************************************************************
 9/2/01:
 Original by: Dave Bell
 Modified by: Michael Haag and Vassos S. Soteriou

 Description:
     This program sets up the DMA control registers to perform data
   transfers between internal data memory and the external AFE, using
   a ping–pong buffering scheme for both input and outpt data.  The
   program writes 1 block of data to one input buffer while the other
   input buffer is available to the CPU.  Also, an output buffer is
   read while the other output buffer can receive information from the
   CPU.  Once the second block is being written, the registers are
   edited to reload the first buffers as the source and desitination
   for the input and output transfers, respectively.
     The sample code is based on TI’s CSL 2.0. See the TMS320C6000
   Chip Support Library API User’s Guide (SPRU401) for further
   information.
*******************************************************************/
/* Chip definition, change this accordingly */
#define CHIP_6202 1

/* Include files */
#include <stdio.h>
#include <c6x.h>
#include <csl.h>         /* CSL library   */
#include <csl_dma.h>     /* DMA_SUPPORT   */
#include <csl_irq.h>     /* IRQ_SUPPORT   */
```

```
/* Define constants */
#define DMA_XFER        8
#define XFER_TYPE       DMA_XFER
#define ELEMENT_COUNT   1024
#define ELEMENT_SIZE    4  /* ELEMENT_SIZE is number of bytes/element, 1,2,4  */
#define FRAME_COUNT     2              /* Frames in the block transfer */
#define BUFFER_SIZE     ELEMENT_COUNT    /* Each buffer fits one frame */
#define LOOPS           3

/* Global variables used in interrupt ISRs and functions */
static Uint32 interInbuff[BUFFER_SIZE]; /* define an intermediate buffer     */
static Uint32 dmaInbuff1[BUFFER_SIZE];  /* buffer for DMA supporting devices */
static Uint32 dmaOutbuff1[BUFFER_SIZE];
static Uint32 dmaInbuff2[BUFFER_SIZE];
static Uint32 dmaOutbuff2[BUFFER_SIZE];
static Uint32 interOutbuff[BUFFER_SIZE]; /* to simulate the external buffer */

Uint32 recv_counter = 0;
Uint32 xmit_counter = 0;
Uint32 loops = LOOPS; /* Number of transfers to perform */
/* Declare CSL objects */
DMA_Handle hDma1;          /* Handle for DMA */
DMA_Handle hDma2;
Uint32 dmaGblRegMsk;       /* DMA Global Register Mask */

/* Select Global Count Reload Register A or Global Index Register A or */
/* DMA Global Address Register B or C                                 */
Uint32 dmaGblRegId = DMA_GBLCNTA | DMA_GBLIDXA |DMA_GBLADDRB | DMA_GBLADDRC;

/* External functions and function prototypes */
void set_interrupts_dma(void);   /* Function prototypes  */

/* Include the vector table to call the IRQ ISRs hookup */
extern far void vectors();
```

```
/********************************************************************/
/*------------------------------------------------------------------*/
/* BEGIN main()                                                     */
/*------------------------------------------------------------------*/
/********************************************************************/
void main(void)
{
/* Declaration of local variables */
static int element_count, xfer_type, frame_index, element_index;
static int frame_count, element_size;

IRQ_setVecs(vectors); /* point to the IRQ vector table */

element_count = ELEMENT_COUNT;
xfer_type = XFER_TYPE;
frame_count = FRAME_COUNT;

/* Calculate the index values, as well as the ESIZE, based on the number of */
/*  elements per frame (element_count), the number of frames per block       */
/*  (frmame_count), and the number of bytes in each element (element_size)   */
element_index = ELEMENT_SIZE;

/* Frame_index is used to jump from 1st In/Out buffer to the 2nd         */
/*  In/Out buffer.  In this example, they are separated by one other buffer */
/*  of equal size, so frame_index is:                                 */
frame_index = (element_count + 1) * ELEMENT_SIZE;

/* covert the number of bytes/element into the CSL HAL MACRO conversion */
if (ELEMENT_SIZE == 1) element_size = 2;       /* 8BIT element size  */
else if (ELEMENT_SIZE == 2) element_size = 1;  /* 16BIT element size */
else element_size = 0;                         /* 32BIT element size */

/* initialize the CSL library */
CSL_init();

/**** START SWITCH HERE ****/
switch (xfer_type) {
case DMA_XFER:
```

```
/*------------------------------------------------------------------*/
/* DMA channel 1 & 2 config structures                              */
/*------------------------------------------------------------------*/
DMA_reset(INV);         /* reset all DMA channels */
dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);
DMA_globalConfigArgs(dmaGblRegMsk,
      DMA_GBLADDR_GBLADDR_DEFAULT,  /* DMA global address registers A to D*/
      DMA_GBLADDR_GBLADDR_OF((Uint32) dmaInbuff1), /* point to dmaInbuff1  */
      DMA_GBLADDR_GBLADDR_OF((Uint32) dmaOutbuff1),/* point to dmaOutbuff1 */
      DMA_GBLADDR_GBLADDR_DEFAULT,

      /* DMA global index registers A to B   */
      DMA_GBLIDX_RMK(
         DMA_GBLIDX_FRMIDX_OF(frame_index),
         DMA_GBLIDX_ELEIDX_OF(element_index)
                      ),
      DMA_GBLIDX_DEFAULT,

      /* Configure DMA global count reload register A  */
      DMA_GBLCNT_RMK(
         DMA_GBLCNT_FRMCNT_OF(frame_count),
         DMA_GBLCNT_ELECNT_OF(element_count) /* element count reload */
      ),
      DMA_GBLCNT_DEFAULT

);  /* end of DMA global control register configuration structure */
  /* DMA channel 1 */
  hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);  /* Handle to DMA channel 1 */
    DMA_configArgs(hDma1,
       DMA_PRICTL_RMK(
           DMA_PRICTL_DSTRLD_B,   // Global Address Register B used
           DMA_PRICTL_SRCRLD_DEFAULT,
           DMA_PRICTL_EMOD_DEFAULT,
           DMA_PRICTL_FS_RSYNC,
           DMA_PRICTL_TCINT_ENABLE,    /* TCINT =1 */
           DMA_PRICTL_PRI_CPU,
           DMA_PRICTL_WSYNC_DEFAULT,
           DMA_PRICTL_RSYNC_EXTINT4,   /* Set sync. event EXT_INT4=00100 */
           DMA_PRICTL_INDEX_DEFAULT,
           DMA_PRICTL_CNTRLD_A, /* Reload with DMA global reload counter A */
           DMA_PRICTL_SPLIT_DEFAULT,
           DMA_PRICTL_ESIZE_OF(element_size), /* 32–bit element size */
```

```
            DMA_PRICTL_DSTDIR_IDX,  // Adjust dest using DMA Global Index Reg. A
            DMA_PRICTL_SRCDIR_NONE, // Increase source by element_size
            DMA_PRICTL_START_DEFAULT
            ),
         DMA_SECCTL_RMK(
            DMA_SECCTL_WSPOL_NA,  /* only available for 6202 and 6203 devices */
            DMA_SECCTL_RSPOL_NA,  /* only available for 6202 and 6203 devices */
            DMA_SECCTL_FSIG_NA,   /* only available for 6202 and 6203 devices */
            DMA_SECCTL_DMACEN_DEFAULT,
            DMA_SECCTL_WSYNCCLR_DEFAULT,
            DMA_SECCTL_WSYNCSTAT_DEFAULT,
            DMA_SECCTL_RSYNCCLR_DEFAULT,
            DMA_SECCTL_RSYNCSTAT_DEFAULT,
            DMA_SECCTL_WDROPIE_DEFAULT,
            DMA_SECCTL_WDROPCOND_DEFAULT,
            DMA_SECCTL_RDROPIE_DEFAULT,
            DMA_SECCTL_RDROPCOND_DEFAULT,
            DMA_SECCTL_BLOCKIE_ENABLE, // Enable block interrupt
            DMA_SECCTL_BLOCKCOND_DEFAULT,
            DMA_SECCTL_LASTIE_DEFAULT,
            DMA_SECCTL_LASTCOND_DEFAULT,
            DMA_SECCTL_FRAMEIE_ENABLE, // Frame condition enables DMA ch. inter.
            DMA_SECCTL_FRAMECOND_DEFAULT,
            DMA_SECCTL_SXIE_DEFAULT,
            DMA_SECCTL_SXCOND_DEFAULT
            ),

         /* Set up source – data from the interInbuff     */
         DMA_SRC_RMK((Uint32)interInbuff),

         /* Set up destination – transfer to the dmaInbuff1  */
         DMA_DST_RMK((Uint32)dmaInbuff1),

         /* Set up transfer count register               */
         DMA_XFRCNT_RMK(
           DMA_XFRCNT_FRMCNT_OF(frame_count),
           DMA_XFRCNT_ELECNT_OF(element_count)  /* set rcev element count */
         )
);
```

```
/* DMA channel 2 */
hDma2= DMA_open(DMA_CHA2, DMA_OPEN_RESET);  /* Handle to DMA channel 2 */
 DMA_configArgs(hDma2,
     DMA_PRICTL_RMK(
         DMA_PRICTL_DSTRLD_DEFAULT,
         DMA_PRICTL_SRCRLD_C,  /* Use Global Address Register C for reload */
         DMA_PRICTL_EMOD_DEFAULT,
         DMA_PRICTL_FS_RSYNC,  /* RSYNC event to synchronize entire frame */
         DMA_PRICTL_TCINT_DISABLE,  /* TCINT = 0  */
         DMA_PRICTL_PRI_CPU,
         DMA_PRICTL_WSYNC_DEFAULT,
         DMA_PRICTL_RSYNC_DMAINT1,  /* Set sync. event DMA_INT1=01001  */
         DMA_PRICTL_INDEX_DEFAULT,
         DMA_PRICTL_CNTRLD_A, /* reload with DMA global reload counter A */
         DMA_PRICTL_SPLIT_DEFAULT,
         DMA_PRICTL_ESIZE_OF(element_size), /*Element size defined by user */
         DMA_PRICTL_DSTDIR_NONE, /* Increment source by element size      */
         DMA_PRICTL_SRCDIR_IDX,  // Adjust src using DMA Global Index Reg. A
         DMA_PRICTL_START_DEFAULT
         ),

     DMA_SECCTL_RMK(
         DMA_SECCTL_WSPOL_NA,  /* only available for 6202 and 6203 devices */
         DMA_SECCTL_RSPOL_NA,  /* only available for 6202 and 6203 devices */
         DMA_SECCTL_FSIG_NA,   /* only available for 6202 and 6203 devices */
         DMA_SECCTL_DMACEN_DEFAULT,
         DMA_SECCTL_WSYNCCLR_DEFAULT,
         DMA_SECCTL_WSYNCSTAT_DEFAULT,
         DMA_SECCTL_RSYNCCLR_DEFAULT,
         DMA_SECCTL_RSYNCSTAT_DEFAULT,
         DMA_SECCTL_WDROPIE_DEFAULT,
         DMA_SECCTL_WDROPCOND_DEFAULT,
         DMA_SECCTL_RDROPIE_DEFAULT,
         DMA_SECCTL_RDROPCOND_DEFAULT,
         DMA_SECCTL_BLOCKIE_ENABLE,
         DMA_SECCTL_BLOCKCOND_DEFAULT,
         DMA_SECCTL_LASTIE_DEFAULT,
         DMA_SECCTL_LASTCOND_DEFAULT,
         DMA_SECCTL_FRAMEIE_ENABLE, /* enable */
         DMA_SECCTL_FRAMECOND_DEFAULT,
         DMA_SECCTL_SXIE_DEFAULT,
           DMA_SECCTL_SXCOND_DEFAULT
           ),

        /* Set up source – data from the dmaOutbuff1     */
        DMA_SRC_RMK((Uint32)dmaOutbuff1),

        /* Set up destination – transfer to the interOutbuff */
        DMA_DST_RMK((Uint32)interOutbuff),

        /* Set up transfer count register                */
        DMA_XFRCNT_RMK(
          DMA_XFRCNT_FRMCNT_OF(frame_count),
          DMA_XFRCNT_ELECNT_OF(element_count)
        )
);

/* Initialize the interrupt(s):                                    */
/*  Disable both channels interrupts and clear the corresponding  */
/*  interrupt bits in the IER.  Enable the global interrupts      */
/*  and the two corresponding to the two channels in use.           */
set_interrupts_dma();

/* Start DMA channels with autoinitialization */
DMA_autoStart(hDma1);

}
/**** END SWITCH HERE ****/

/* Continue ping–pong transfer between the buffers until number   */
/*                 of transfers needed have been performed
                   */
while(!(recv_counter == loops));

DMA_close(hDma1);   /* close DMA channel 1 */
DMA_close(hDma2);   /* close DMA channel 2 */
}
/********************************************************************/
/*------------------------------------------------------------------*/
/* END MAIN (PROGRAM ENDS HERE)                                     */
/*------------------------------------------------------------------*/
/********************************************************************/
```

```
/*------------------------------------------------------------------*/
/* set_interrupts_dma()                                             */
/*------------------------------------------------------------------*/
void                           /* Set the interrupts */
set_interrupts_dma(void)       /* if the device supports DMA */
{
 IRQ_nmiEnable();
 IRQ_globalEnable();
 IRQ_disable(IRQ_EVT_DMAINT1);  /* INT09 */
 IRQ_disable(IRQ_EVT_DMAINT2);  /* INT11 */
 IRQ_clear(IRQ_EVT_DMAINT1);
 IRQ_clear(IRQ_EVT_DMAINT2);
 IRQ_enable(IRQ_EVT_DMAINT1);
 IRQ_enable(IRQ_EVT_DMAINT2);
 return;
}

/*-----------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISR                             */
/*-----------------------------------------------------------------*/
interrupt void     /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)      /* DMA ch1                          */
{
     /* Begin Channel 2 after Channel 1 has completed its transfer    */
     /*  of the first block.  Remeber, this means one more block will */
     /*  will be transfered in than is transfered out if an extra     */
     /*  output is not triggered.                                     */
     if (recv_counter == 1)
            DMA_autoStart(hDma2);

     /* Manually clear FRAMECOND bit after each frame transfer    */
     DMA_FSET(SECCTL1, FRAMECOND, 0);

     /* If the second frame has completed, clear the BLOCKCOND bit    */
     /* and set variable to trigger start of Channel 2.  Also,        */
     /* manipulate DSTRLD not to set the reload destination to the    */
     /* beginning (dmaInbuff1) until the second block is transferred  */
     /* to dmaInbuff2.                                        */
     if (DMA_FGET(SECCTL1, BLOCKCOND) )
     {
            DMA_FSET(SECCTL1, BLOCKCOND, 0);
```

```
          /* Count the number of tranfers performed and for this example,  */
          /*  recv_counter is used to trigger the end of the Ping–Pong     */
         /*  transfers.                                                    */
         recv_counter++;
     }
     return;
}

/* NOTE: Ch2 ISR entered ONLY if TCINT field in DMA PRICTL2 is enabled  */
interrupt void     /* vecs.asm hooks this up to IRQ 11      */
c_int11(void)      /* DMA ch2                               */
{
     DMA_FSET(SECCTL2, FRAMECOND, 0);
     if (DMA_FGET(SECCTL2, BLOCKCOND) )
     {
          DMA_FSET(SECCTL2, BLOCKCOND, 0);
          xmit_counter++;
     }
     return;
}

/*–––––––––––––––––––––––End of ping_pong__dma.c––––––––––––––––––––*/
```

A.9 Program Paging Transfer Example Code

```
/*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
/*  paging_dma.c V1.00                                                      */
/*  Copyright (c) 2001 Texas Instruments Incorporated                       */
/*––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––––*/
/*
   7/25/01
   Original by: Dave Bell
   Modified by: Vassos S. Soteriou

   paging_dma.c:
       This sample code sets up the DMA control registers to perform data
   Transfers from external memory to internal DSP program memory using a paging
   scheme. There are four external program pages, extPage1, extPage2, extPage3
   and extPage4.  These are brought into internal program memory locations,
   intPage1 and IntPage2, to be executed.  extPage1 is first brought into
   intPage1, then extPage2 into intPage2, then extPage3 into intPage1
   (overwritten) and lastly extPage4 to intPage1 (overwritten). Note that the
   extPage1 and extPage 2 are transferred to the internal program memory first
   and that in the interrupt service routine the DMA_GBLADDRB and DMA_GBLADDRC
   are adjusted (reloaded) with the addresses of extPage3 and extPage4 (and
   also with the address of intPage1 and intPage2) to complete the transfers
   from extPage3 and extPage4 to the internal DSP program memory.
       The sample code is based on TI’s CSL 2.0. See the TMS320C6000 Chip
   Support Library API User’s Guide (SPRU401) for further information.
*/

/* Chip definition, change this accordingly, only DMA DSPs though */
#define CHIP_6202 1
/* Include files */
#include <c6x.h>
#include <csl.h>         /* CSL library   */
#include <csl_dma.h>     /* DMA_SUPPORT   */
#include <csl_irq.h>     /* IRQ_SUPPORT   */

/* Define constants */
#define FALSE 0
#define TRUE 1

#define DMA_TRANS      8
#define XFER_TYPE      DMA_TRANS
#define BUFFER_SIZE    256    /* >= (ELEMENT_COUNT * FRAME_COUNT) */
#define FRAME_COUNT    1      /* Keep this value to 1 frame       */
#define ELEMENT_COUNT  32     /* Define # of elements to transfer per page  */

/* Global variables used in interrupt ISR */
volatile int transfer_done = FALSE;

/* Let these buffers in the global section as they are also used     */
/* in the Interrupt Service Routines (ISRs)                          */
static Uint32 intPage1[BUFFER_SIZE]; /* define internal pages */
static Uint32 intPage2[BUFFER_SIZE];

/* Allocate space for extPage1, extPage2, extPage3 & extPage4 in         */
/*   MEMORY/SECTIONS section of the link command file, *.cmd. Note       */
/*   that the user also has to define  the origin and length of these    */
/*   external pages/memories in the cmd file                             */
#pragma DATA_ALIGN (extPage1, 128);  // Assign extPage1 to alignment boundary
#pragma DATA_SECTION(extPage1, ”external_data”);
#pragma DATA_ALIGN (extPage2, 128);  // Assign extPage2 to alignment boundary
#pragma DATA_SECTION(extPage2, ”external_data”);
```

```
#pragma DATA_ALIGN (extPage3, 128);  // Assign extPage3 to alignment boundary
#pragma DATA_SECTION(extPage3, ”external_data”);
#pragma DATA_ALIGN (extPage4, 128);  // Assign extPage4 to alignment boundary
#pragma DATA_SECTION(extPage4, ”external_data”);

static Uint32 extPage1[BUFFER_SIZE]; /* define external pages */
static Uint32 extPage2[BUFFER_SIZE];
static Uint32 extPage3[BUFFER_SIZE];
static Uint32 extPage4[BUFFER_SIZE];

/* These pointers are used to load starting address values of external */
/* and internal pages to the DMA Address Global Reload Registers       */
Uint32 *gbladdr1, *gbladdr2, *gbladdr3, *gbladdr4;

/* Declare CSL objects */
DMA_Handle hDma1;           /* Handle for DMA            */
Uint32 RegId1;              /* Global register IDs       */
Uint32 RegId2;
Uint32 dmaGblRegMsk;        /* DMA Global Register Mask  */

/* Select Global Address Reload Register A or Global Address Register B or C */
Uint32 dmaGblRegId = DMA_GBLCNTA | DMA_GBLADDRB | DMA_GBLADDRC;

/* External functions and function prototypes */
void set_interrupts_dma(void);

/* Include the vector table to call the IRQ ISRs hookup */
extern far void vectors();
/*---------------------------------------------------------------------------*/
/* main()                                                                    */
/*---------------------------------------------------------------------------*/

void main(void)
{
/* Declaration of local variables */
static int element_count, xfer_type, frame_count, page_size;

IRQ_setVecs(vectors); /* point to the IRQ vector table */
element_count = ELEMENT_COUNT;
xfer_type = XFER_TYPE;
frame_count = FRAME_COUNT;
page_size = element_count;
```

```
/* initialize the CSL library */
CSL_init();

switch (xfer_type) {
case DMA_TRANS:

DMA_reset(INV);    /* Reset all DMA channels */
/*---------------------------------------------------------------------------*/
/* DMA channel 1 config structure                                            */
/*---------------------------------------------------------------------------*/
/* DMA channel 1 */

/* Establish Global Register Values in the following configuration structure */
dmaGblRegMsk = DMA_globalAlloc(dmaGblRegId);
DMA_globalConfigArgs(dmaGblRegMsk,
      DMA_GBLADDR_GBLADDR_DEFAULT,  /* DMA global address registers A to D */
      DMA_GBLADDR_GBLADDR_OF((Uint32) extPage2),
      DMA_GBLADDR_GBLADDR_OF((Uint32) intPage2),
      DMA_GBLADDR_GBLADDR_DEFAULT,
      DMA_GBLIDX_RMK(               /* DMA global index registers A to B   */
         DMA_GBLIDX_FRMIDX_DEFAULT,
         DMA_GBLIDX_ELEIDX_DEFAULT
      ),
      DMA_GBLIDX_DEFAULT,
      DMA_GBLCNT_RMK(   /* config DMA global count reload register A */
         DMA_GBLCNT_FRMCNT_OF(frame_count),  /* frame count reload   */
         DMA_GBLCNT_ELECNT_OF(page_size)     /* element count reload */
      ),
      DMA_GBLCNT_DEFAULT

);  /* end of DMA global control register configuration structure */

hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);  /* Handle to DMA channel 1 */
DMA_configArgs(hDma1,
    DMA_PRICTL_RMK(
        DMA_PRICTL_DSTRLD_C,  //Use DMA Global Addr. Register C as dest .reload
        DMA_PRICTL_SRCRLD_B,  // Use DMA Global Addr. Register B as src. reload
        DMA_PRICTL_EMOD_HALT, // DMA cahnnel pauses during emulation halt
        DMA_PRICTL_FS_RSYNC,  // RSYNC event used to synchronize entire frame
        DMA_PRICTL_TCINT_ENABLE,    /* TCINT =1 */
        DMA_PRICTL_PRI_DMA,       /* DMA high priority  */
        DMA_PRICTL_WSYNC_DEFAULT,
        DMA_PRICTL_RSYNC_EXTINT4,    /* Set sync. event EXT_INT4=00100 */
        DMA_PRICTL_INDEX_DEFAULT,
        DMA_PRICTL_CNTRLD_A,   /* Reload with DMA global reload counter A  */
        DMA_PRICTL_SPLIT_DISABLE,
         DMA_PRICTL_ESIZE_32BIT, /* Element size is 32 bits long */
         DMA_PRICTL_DSTDIR_INC,  /* Increment destination by element size */
        DMA_PRICTL_SRCDIR_INC,  /* Increment destination by element size     */
        DMA_PRICTL_START_DEFAULT
         ),
     DMA_SECCTL_RMK(
         DMA_SECCTL_WSPOL_NA,    /* only available for 6202 and 6203 devices */
         DMA_SECCTL_RSPOL_NA,    /* only available for 6202 and 6203 devices */
         DMA_SECCTL_FSIG_NA,     /* only available for 6202 and 6203 devices */
         DMA_SECCTL_DMACEN_DEFAULT,
         DMA_SECCTL_WSYNCCLR_DEFAULT,
         DMA_SECCTL_WSYNCSTAT_DEFAULT,
         DMA_SECCTL_RSYNCCLR_DEFAULT,
         DMA_SECCTL_RSYNCSTAT_DEFAULT,
         DMA_SECCTL_WDROPIE_DEFAULT,
         DMA_SECCTL_WDROPCOND_DEFAULT,
         DMA_SECCTL_RDROPIE_DEFAULT,
         DMA_SECCTL_RDROPCOND_DEFAULT,
         DMA_SECCTL_BLOCKIE_ENABLE, /* BLOCK IE=1 enables DMA channel int */
         DMA_SECCTL_BLOCKCOND_DEFAULT,
         DMA_SECCTL_LASTIE_DEFAULT,
         DMA_SECCTL_LASTCOND_DEFAULT,
         DMA_SECCTL_FRAMEIE_DEFAULT,
         DMA_SECCTL_FRAMECOND_DEFAULT,
         DMA_SECCTL_SXIE_DEFAULT,
         DMA_SECCTL_SXCOND_DEFAULT
         ),
     DMA_SRC_RMK((Uint32) extPage1),  /* source page      */
     DMA_DST_RMK((Uint32) intPage1),  /* destination page */
    DMA_XFRCNT_RMK(
        DMA_XFRCNT_FRMCNT_OF(frame_count), /* set xfer frame count */
        DMA_XFRCNT_ELECNT_OF(page_size)    /* set xfer page size   */
        )
);
```

```
/* Initialize the interrupts                                  */
/*    Enable the interrupts after the DMA channels are opened */
/* as the DMA_OPEN_RESET clears and disables the channel      */
/* interrupt once specified and clears the corresponding      */
/* interrupt bits in the IER. */
set_interrupts_dma();

DMA_autoStart(hDma1);  /* Start DMA channel 1 with autoinitialization    */
} /* end of switch here */
/* To flag an interrupt to the CPU when DMA transfer/receive is done   */
while (!transfer_done);

DMA_close(hDma1); /* close the channel when the transfer is complete */
} /* end main, program ends here */

/*----------------------------------------------------------------------------*/
/* set_interrupts_dma()                                                       */
/*----------------------------------------------------------------------------*/
void                          /* Set the interrupts */
set_interrupts_dma(void)
{
 IRQ_nmiEnable();
 IRQ_globalEnable();
 IRQ_disable(IRQ_EVT_DMAINT1);  /* INT9 */
 IRQ_clear(IRQ_EVT_DMAINT1);
 IRQ_enable(IRQ_EVT_DMAINT1);
 return;
}
/*---------------------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISR                                       */
/*---------------------------------------------------------------------------*/
interrupt void       /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)        /* DMA ch1                          */
{
     transfer_done = TRUE;
     /* Allocate DMA Global Address Registers B & C */
     RegId1 = DMA_allocGlobalReg(DMA_GBL_ADDRRLD, 0x0184003C);
     RegId2 = DMA_allocGlobalReg(DMA_GBL_ADDRRLD, 0x0184006C);

     DMA_freeGlobalReg(RegId1);  /* Free DMA Clobal Registers that were */
     DMA_freeGlobalReg(RegId2);  /* previously allocated                */
     gbladdr1 = extPage3; /* Get addresses of extPage3,4 and IntPage1,2 */
     gbladdr2 = intPage1;
     gbladdr3 = extPage4;
     gbladdr4 = intPage2;

     /* Set GLBADDRB to point to starting address of extPage 3 */
     DMA_setGlobalReg(RegId1, (Uint32) gbladdr1);
     /* Set GLBADDRC to point to starting address of intPage 1 */
     DMA_setGlobalReg(RegId2, (Uint32) gbladdr2);

     /* Set GLBADDRB to point to starting address of extPage 4 */
     DMA_setGlobalReg(RegId1, (Uint32) gbladdr3);
     /* Set GLBADDRC to point to starting address of intPage 2 */
     DMA_setGlobalReg(RegId2, (Uint32) gbladdr4);

     return;
}
/*-----------------------End of paging_dma.c---------------------------------*/
```