![Texas Instruments logo]

TMS320C6000 McBSP: I²S Interface

Todd Hiers / Rebecca Ma   *Digital Signal Processing Solutions*

Abstract

This document describes how to use the multichannel buffered serial port (McBSP) in the Texas Instruments (TI™) TMS320C6000 digital signal processors (DSP) to interface with devices that conform to the Inter-IC Sound (I²S) specification. I²S is a protocol for transmitting two channels of digital audio data over a single serial connection.

The flexible McBSP in the TMS320C6000 supports the I²S specification by simple setup of the serial control registers. The McBSP is capable of generating all of the necessary clocking signals to act as the I²S master. The McBSP is also capable of receiving the clocking signals to be the I²S slave. The dual-phase transmit/receive mode of the McBSP is easy to configure for transmitting the left and right audio channels.

To make processing the audio data easier, it can be deinterleaved after reception or reinterleaved before transmission automatically by the DMA controller. Using autoincrementing and indexing with the proper index increment values, the DMA will fill or draw from separate left and right channel buffers for reception or transmission.

Contents

Figures

Tables

Design Problem

How can the multi-channel buffered serial port (McBSP) in a TMS320C6000 digital signal processor be used for transmitting audio data in I^{2}S format?
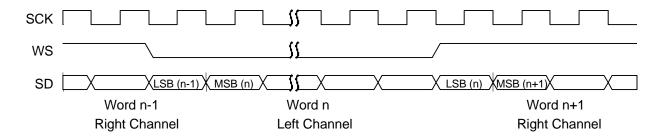
Overview

The Inter-IC Sound (I^{2}S) serial interconnect format is a popular standard for the exchange of stereo digital samples between two devices on a printed circuit board. The multi-channel serial port (McBSP) on the TMS320C6000 family of digital signal processors is flexible enough to talk to most devices supporting the I^{2}S standard.

The I^{2}S bus is a three-wire connection that exclusively handles two time-multiplexed data channels. Other information such as sub-coding and control are transferred separately. The three lines are the bit clock (SCK), the word select line (WS), and the serial data line (SD). The master device on the bus is responsible for generating appropriate SCK and WS signals, and the transmitting device (which may or may not be the master device) places the appropriate serial data on the bus. Data is transmitted MSB first, alternating left and right channel data words, with left channel data on the bus while WS is low and right channel data on the bus while the WS line is high. Figure 1 shows a timing diagram of bus data.

The I^{2}S specification is flexible enough that receivers and transmitters need not agree on a word size. Since data is transmitted MSB first and each new word is indicated by a transition in WS, the slave device can determine the appropriate word size by the master's signals on the fly. However, when using the 'C6000 as the slave device, this automatic word size detection is not possible.

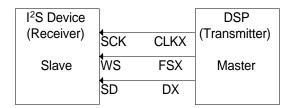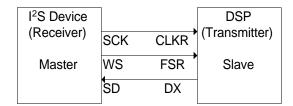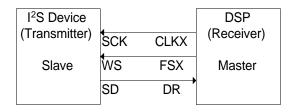*Figure 1. I^{2}S Timing Specification*

McBSP Setup

The McBSP conforms to the I^2S interface by using the frame syncronization signal (FSX or FSR) as the WS signal, the bit clock (CLKX, CLKR, or CLKS) as SCK, and the data pins (DR and DX) as the SD line. The actual signals used depend the DSP's role as master or slave and its role as transmitter or receiver. Only one device on the I^2S bus can be the master, since the master is responsible for generating the clocking signals. The McBSP can be both a transmitter and receiver simultaneously, allowing for continuous data exchange. This capability is possible when the I^2S device is similarly capable of simultaneous reception and transmission. In this setup, there are two SD lines, both of which are clocked by SCK and WS. One SD line handles the data transfer from the DSP to the I^2S device, while the other handles data from the I^2S device to the DSP. Figure 2 illustrates the different possible configurations of I^2S operation. The bottom configurations in Figure 2, which are supersets of the ones above, are described in this application report.

Figure 2. I^2S Modes of Operation

Master Mode

Pin Control Register (PCR)

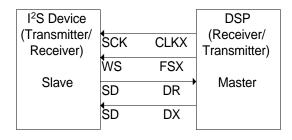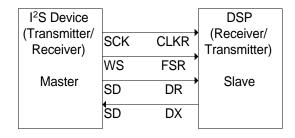As the I^{2}S master device, the DSP generates the serial clock (SCK) and word select (WS) signals. CLKX and CLKR are configured as output pins and are used as the serial clock. FSX and FSR are also configured as output pins so that either may be used as the WS signal. The frame synchronization polarity bits (FSXP/FSRP) in PCR determine if the frame begins with left channel data or right channel data. The FSXP and FSRP bits should have the same value. Table 1 and Figure 3 show the Pin Control Register settings for I^{2}S master mode.

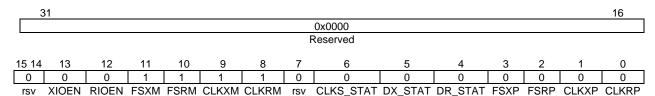*Figure 3.   Pin Control Register (PCR)*

31															16
0x0000															
Reserved															

15 14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0 0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
rsv	XIOEN	RIOEN	FSXM	FSRM	CLKXM	CLKRM	rsv	CLKS_STAT	DX_STAT	DR_STAT	FSXP	FSRP	CLKXP	CLKRP

Table 1.   Bit-Field Values for Pin Control Register

Register [bit-field #]	Bit-field Name	Value (in binary)	Function
PCR[11]	FSXM	1	FSX is output pin
PCR[10]	FSRM	1	FSR is output pin
PCR[9]	CLKXM	1	CLKX is output pin
PCR[8]	CLKRM	1	CLKR is output pin
PCR[1]	CLKXP	0	Data is clocked out on the rising edge of SCK
PCR[0]	CLKRP	0	Data is sampled in on the falling edge of SCK

Receive/Transmit Control Register

The McBSP is set to dual-phase frame mode by setting the appropriate bits (R/XPHASE) in the receive/transmit control registers (R/XCR). One phase is for left-channel data, and the other is for right-channel data. The data word length is set to the appropriate value for both phases (R/XWDLEN1/2 in R/XCR). The number of words in each phase of the frame is set to 1 (R/XFRLEN1/2 in R/XCR = 0). New data words begin one SCK cycle after a change in the WS line (R/XDATDLY in R/XCR = 1). Figure 4 and 5 and Table 2 show the Receive/Transmit Control Register settings with 32-bit data.

Figure 4. Receive Control Register (RCR)

31	30	24	23	21	20	19	18	17	16
1	000		101		00		1	1	
RPHASE	RFRLEN2		RWDLEN2		RCOMPAND		RFIG	RDATDLY	

15	14	8	7	5	4	0
0	000		101		0	
reserved	RFRLEN1		RWDLEN1		Reserved	

Figure 5. Transmit Control Register (XCR)

31	30	24	23	21	20	19	18	17	16
1	000		101		00		1	1	
XPHASE	XFRLEN2		XWDLEN2		XCOMPAND		XFIG	XDATDLY	

15	14	8	7	5	4	0
0	000		101		0	
reserved	XFRLEN1		XWDLEN1		Reserved	

Table 2. Bit-Field Values for Receive/Transmit Control Register

Register [bit-field #]	Bit-field Name	Value (in binary)	Function
RCR[31]	RPHASE	1	Dual Phase Receive
RCR[7-5]	RWDLEN1	101	32 bits Receive Word Length (Phase 1)
RCR[23-21]	RWDLEN2	101	32 bits Receive Word Length (Phase 2)
RCR[14-8]	RFRLEN1	0	1 word Receive Frame Length (Phase 1)
RCR[30-24]	RFRLEN2	0	1 word Receive Frame Length (Phase 2)
XCR[31]	XPHASE	1	Dual Phase Transmit
XCR[7-5]	XWDLEN1	101	32 bits Transmit Word Length (Phase 1)
XCR[23-21]	XWDLEN2	101	32 bits Transmit Word Length (Phase 2)
XCR[14-8]	XFRLEN1	0	1 word Transmit Frame Length (Phase 1)
XCR[30-24]	XFRLEN2	0	1 word Transmit Frame Length (Phase 2)

Sample Rate Generator Register (SRGR)

The frame sync signal is configured by setting the frame width field (FWID), the frame period field (FPER), and the frame synchronization bit (FSGM) in the sample rate generator register (SRGR). The FSGM bit forces the frame sync to be generated based on the serial clock regardless of the availability of data in DXR. In this mode, a new frame sync is generated every FPER + 1 serial clock cycles, and the signal is held active (either indicating a left or right channel word depending on the polarity bit) for FWID + 1 serial clock cycles. In this way, it is possible to create waveforms with various periods and duty cycles on the FSX pin.

For I^2S, the frame period must be twice the word length, and the frame width must be the word length. Therefore, FPER in SRGR should be set to (2 * word length – 1), and FWID in SRGR should be set to (word length – 1). This allows the FSX pin to be the WS line for transmission/reception of left and right channel data, as the WS line is held low for one word length, then high for one word length.

Finally, since the clock signals are generated by the McBSP, the serial clock must be set to an appropriate speed. The device being interfaced to determines the maximum speed at which the serial clock can run. Although other setups are possible, the most practical setup is to have the CPU clock derive the serial clock, with an appropriate divide-down factor. Alternatively, CLKS can be generated from an external oscillator that runs at a multiple of the sampling frequency. For example, if 48KHz sampling is desired and the left and right channels contain a single 32-bit word, CLKS can be driven by a 3.072MHz (32 x 2 x 48KHz) oscillator. The sample rate generator clock divider (CLKGDV) field of SRGR must be chosen such that (CPU clock frequency) / (2 * word length * (CLKGDV + 1)) is less than or equal to the I^2S device's maximum sampling/output frequency.

Figures 6 and Table 3 show the SRGR settings for I^2S master mode with the serial clock as the CPU clock divided by 71.
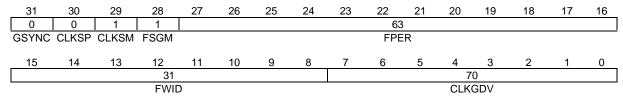
Figure 6.   Sample Rate Generator Register (SRGR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	1	1						63						
GSYNC	CLKSP	CLKSM	FSGM						FPER						

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			31								70				
			FWID								CLKGDV				

Table 3.   Bit-Field Values for McBSP Registers

Register [bit-field #]	Bit-field Name	Value (in binary)	Function
SRGR[28]	FSGM	1	Frame Sync generated by sample rate generator
SRGR[27-16]	FPER	111111 (63)	64 Cycle Frame Period
SRGR[15-8]	FWID	11111 (31)	32 Cycle Frame Active Duration
SRGR[7-0]	CLKGDV	1000110 (70)	CLKX = CPU clock divided by 71

Slave Mode

I^2S slave mode is simpler to implement than master mode because the slave is not responsible for generating any frame or clock signals. The McBSP must simply accept the incoming clock and frame sync signals generated by the master device. In this case the CLKX, CLKR, FSX, and FSR pins are configured as input pins (which is the default setting). Clearing bits CLKXM, CLKRM, FSXM, and FSRM in PCR will set those pins as inputs.

The R/XCR settings are the same as for master mode, since the same data scheme is implemented. The SRGR does not need to be set up since external clocks and frames are provided. Dual-phase mode with one word per phase and the appropriate word size should be specified by setting the appropriate fields in R/XCR. Specifically, set R/XPHASE to 1, R/XFRLEN1/2 to 0, and R/XWDLEN1/2 to the appropriate value for the chosen word size.

Since the WS signal is understood by the McBSP as a frame sync signal and not a true word select signal, it must know the data word size ahead of time. If the McBSP is not properly set up with the correct word size, incorrect data will be sent and received. For this reason the McBSP does not conform 100% to the I²S specification for slave devices. However, as long as the data word size is correct, the McBSP will function properly in slave mode.

Figures 7 through 9 and Table 4 show the McBSP register settings for I²S slave mode with 32 bit data.
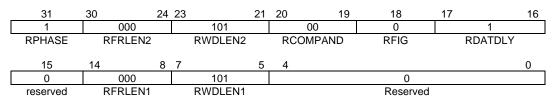
Figure 7. Receive Control Register (RCR)

31	30 24	23 21	20 19	18	17 16
1	000	101	00	0	1
RPHASE	RFRLEN2	RWDLEN2	RCOMPAND	RFIG	RDATDLY

15	14 8	7 5	4 0
0	000	101	0
reserved	RFRLEN1	RWDLEN1	Reserved

Figure 8. Transmit Control Register (XCR)

31	30 24	23 21	20 19	18	17 16
1	000	101	00	0	1
XPHASE	XFRLEN2	XWDLEN2	XCOMPAND	XFIG	XDATDLY

15	14 8	7 5	4 0
0	000	101	0
reserved	XFRLEN1	XWDLEN1	Reserved

Figure 9. Pin Control Register (PCR)

31	16
0x0000
Reserved

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
rsv	XIOEN	RIOEN	FSXM	FSRM	CLKXM	CLKRM	rsv	CLKS_STAT	DX_STAT	DR_STAT	FSXP	FSRP	CLKXP	CLKRP

Table 4. Bit-Field Values for McBSP Registers

Register [bit-field #]	Bit-field Name	Value (in binary)	Function
RCR[31]	RPHASE	1	Dual Phase Receive
RCR[7-5]	RWDLEN1	101	32 bits Receive Word Length (Phase 1)
RCR[23-21]	RWDLEN2	101	32 bits Receive Word Length (Phase 2)
RCR[14-8]	RFRLEN1	0	1 word Receive Frame Length (Phase 1)
RCR[30-24]	RFRLEN2	0	1 word Receive Frame Length (Phase 2)
XCR[31]	XPHASE	1	Dual Phase Transmit
XCR[7-5]	XWDLEN1	101	32 bits Transmit Word Length (Phase 1)
XCR[23-21]	XWDLEN2	101	32 bits Transmit Word Length (Phase 2)
XCR[14-8]	XFRLEN1	0	1 word Transmit Frame Length (Phase 1)
XCR[30-24]	XFRLEN2	0	1 word Transmit Frame Length (Phase 2)
PCR[11]	FSXM	0	FSX is input pin
PCR[10]	FSRM	0	FSR is input pin
PCR[9]	CLKXM	0	CLKX is input pin
PCR[8]	CLKRM	0	CLKR is input pin

Receiving/Transmitting I^{2}S Data

The I^{2}S serial format transmits interleaved data by alternating between left and right channel data. Once the McBSP is properly set up to handle I^{2}S serial format, the DSP must also be programmed to effectively handle the interleaved data. Since most DSP algorithms expect continuous non-interleaved data, the received data must be sorted into left and right data buffers before processing can occur, and then the data must be re-interleaved before it can be transmitted.

You can set up a DMA channel to service the serial port and sort the data. By using properly set up index registers to do the destination autoincrement, the DMA sorts the received data into two separate buffers in memory. Similarly, by using index registers to do the source autoincrement, the DMA automatically re-interleaves data from two memory buffers for transmission. DMA split mode operation is possible, with some restrictions.

To do this sorting, the DMA must be set up to do the required transfer as two-word frames for the left and right channel data. The autoincrement of the source/destination register is set to use an index register that has the appropriate element and frame increment values set. The following formulas compute the element and frame index:

- B = Buffer size (total number of elements in each buffer)
- S = Element size in bytes

    ❑　　　ELEMENT INDEX = B x S
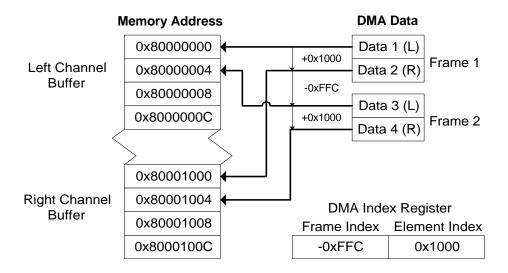   ❑　　　FRAME INDEX = S – (B x S)

The element index is the difference in memory locations (in bytes) of the right channel buffer and the left channel buffer. The frame index is the data word size (in bytes) minus the difference in buffer memory locations. Note that this result should be a negative number. With this setup, the first element of the DMA frame goes into the first buffer. Then, the destination increments by the difference between the buffers, so the second element goes into the same relative position in the second buffer. The destination register then increments by the word size minus the memory location difference (a negative number), pointing it to the next relative location in the first buffer. The next frame can then be received and sorted as the previous one. Figure 10 shows how DMA sorting works. In Figure 10, the element size $S = 4$ bytes, and the buffer size $B = 0x400$ elements in each buffer. The element index is $B \times S = 400h \times 4 = 1000h$. The frame index is $S - (B \times S) =$

The DMA autoinitialization feature allows continuous data transfer. To enable DMA autoinitialization, write START=11b in the channel's primary control register, and set the corresponding DMA global count reload register and the DMA global address register to the desired reload values. If you want to modify the reload register values before the next DMA autoinitialization, set TCINT=1 in the channel's primary control register and LAST IE=1 in the channels' secondary control register to enable last frame condition interrupt. Upon receiving the DMA channel last frame condition, the interrupt service routine can modify the reload register values before the next autoinitialization. This feature is especially useful when the buffer size is smaller than the amount of I^2S data received. You can reuse the same buffer space for different data blocks simply by setting the DMA global address register to the beginning of the buffer address.

This same setup works for the source autoincrement as well, allowing the automatic re-interleaving of data for transmission. Split mode may be used to allow one DMA channel to do both sorting and unsorting only if the difference between the receive left and right buffers' memory locations is the same as the difference between the transmit left and right buffers' memory locations. This restriction is because the same index register must be used for both. See the *TMS320C6000 DMA Example Applications* application report for details and examples on how to set up the DMA for sorting.

Figure 10. DMA Sorting

Conclusion

The Multichannel Buffered Serial Port on the TMS320C6000 Digital Signal Processor is flexible enough to interface to devices that conform to the Inter-IC Sound specification. The DSP is capable of being either the master or the slave device on the I^{2}S bus. Proper configuration of the McBSP is simple for both master and slave modes, and the DMA controller can be configured to handle data deinterleaving and reinterleaving automatically.

Sample C Functions

This C code shows how to set up the McBSP and DMA to interface to an I^{2}S device, specifically a Crystal CS4226 Surround Sound Codec. Since both the CS4226 and the McBSP can act as either master or slave, the code allows for either configuration. The 'C6000 McBSP's role is controlled by use of preprocessor defines. By default, the McBSP is the master and the CS4226 is the slave, but defining SLAVE causes the McBSP to be the slave and the CS4226 to be the master.

In this code, McBSP 0 simultaneously receives and transmits audio data in I^{2}S format. McBSP 1 interfaces to the CS4226's control port to program the device. DMA channel 0 in split mode services the data transfer to and from McBSP 0.

The mcbsp.c file sets up the serial ports and DMA channel, and dma_int.c provides the necessary interrupt service routines. The mcbsp.c file also periodically copies the receive buffer to the transmit buffer to keep continuous data flow through the serial port. Although no digital signal processing is done on the data, the code indicates where any DSP algorithms operate. The net effect of the code is that the I^{2}S data fed into McBSP 0 is received in, sorted, unsorted, and then transmitted out a short while later.

```
/****************************************************************/
/*      mcbsp.c                                                 */
/****************************************************************/

#include <mcbsp.h>
#include <dma.h>
#include <time.h>
#include <stdlib.h>

/* Definitions */
#define MEM_SRC         0x80000000
#define MEM_DST         0x80001000

/* Uncomment the following for C6000 as I2S slave */
//#define SLAVE

/* Global variables */
int RECV_done = 0;
int XMIT_done = 0;
int DMA_done[4] = {0, 0, 0, 0};

/* Prototypes */
extern void set_interrupts(void);
void config_serial(void);
void start_cs4226(void);
```

```
void start_sp_dma(void);




void
start_sp_dma(void)
{
unsigned int    dma_pri_ctrl    = 0;
unsigned int    dma_sec_ctrl    = 0;
unsigned int    dma_src_addr    = 0;
unsigned int    dma_dst_addr    = 0;
unsigned int    dma_tcnt        = 0;
unsigned int    dma_index       = 0;

        /* Clear completion flag */
        DMA_done[0] = 0;

        /* Reset DMA Control Registers  */
        /* use DMA Ch0 to service McBSP */
        dma_reset();
        DMA_RSYNC_CLR(0);
        DMA_WSYNC_CLR(0);
        dma_reset();


        /* Set up DMA Channel to perform a block transfer of       */
        /* XFER_SIZE elements                                      */
        /*    from MEM_SRC to McBSP */
        /* Set up DMA Primary Control Register */


        LOAD_FIELD(&dma_pri_ctrl, DMA_RELOAD_GARC, DST_RELOAD,DST_RELOAD_SZ);
        LOAD_FIELD(&dma_pri_ctrl, DMA_RELOAD_GARB, SRC_RELOAD,SRC_RELOAD_SZ);
        LOAD_FIELD(&dma_pri_ctrl, DMA_DMA_PRI    , PRI        , 1             );
        LOAD_FIELD(&dma_pri_ctrl, SEN_XEVT0      , WSYNC      , WSYNC_SZ     );
        LOAD_FIELD(&dma_pri_ctrl, SEN_REVT0      , RSYNC      , RSYNC_SZ     );
        LOAD_FIELD(&dma_pri_ctrl, DMA_SPLIT_GARA , SPLIT      , SPLIT_SZ     );
        LOAD_FIELD(&dma_pri_ctrl, DMA_ESIZE32    , ESIZE      , ESIZE_SZ     );
        LOAD_FIELD(&dma_pri_ctrl, DMA_ADDR_INDX, DST_DIR      , DST_DIR_SZ    );
        LOAD_FIELD(&dma_pri_ctrl, DMA_ADDR_INDX, SRC_DIR      , SRC_DIR_SZ    );
        SET_BIT(&dma_pri_ctrl,EMOD);            /* Halt DMA with emu halt    */
        SET_BIT(&dma_pri_ctrl,TCINT);           /* Allow Ch to interrupt CPU */

        /* Set up DMA Secondary Control Register */
        LOAD_FIELD(&dma_sec_ctrl, DMAC_BLOCK_COND, DMAC_EN   , DMAC_EN_SZ  );
        SET_BIT(&dma_sec_ctrl, BLOCK_IE);

        /* Set up DMA Tranfer Count Register */
        LOAD_FIELD(&dma_tcnt, 0x100     , FRAME_COUNT  , FRAME_COUNT_SZ  );
        LOAD_FIELD(&dma_tcnt, 2         , ELEMENT_COUNT, ELEMENT_COUNT_SZ);

        /* Set up DMA Index Register */
        LOAD_FIELD(&dma_index, -0x7FC , FRAME_INDEX  , FRAME_INDEX_SZ  );
        LOAD_FIELD(&dma_index, 0x800  , ELEMENT_INDEX, ELEMENT_INDEX_SZ);

        /* Set up Source and Destination Address Registers */
        dma_src_addr = (unsigned int)MEM_SRC;
```

```
        dma_dst_addr = (unsigned int)MEM_DST;


        DMA_GADDR_A = (unsigned int)MCBSP_DRR_ADDR(0);
        DMA_GADDR_B = (unsigned int)MEM_SRC;
        DMA_GADDR_C = (unsigned int)MEM_DST;
        DMA_GCR_A = dma_tcnt;
        DMA_GNDX_A = dma_index;

        /* Store DMA Control registers */
        dma_init(0,
              dma_pri_ctrl,
              dma_sec_ctrl,
              dma_src_addr,
              dma_dst_addr,
              dma_tcnt);

        /* Start DMA Transfer */
        DMA_AUTO_START(0);
        DMA_RSYNC_CLR(0);
        DMA_WSYNC_CLR(0);

} /* end start_sp_dma */

void
config_serial(void)
{
unsigned int spcr = 0;
unsigned int rcr  = 0;
unsigned int xcr  = 0;
unsigned int srgr = 0;
unsigned int mcr  = 0;
unsigned int rcer = 0;
unsigned int xcer = 0;
unsigned int pcr  = 0;


        /* Set up Pin Control Register */
#ifndef SLAVE
        LOAD_FIELD(&pcr, FSYNC_MODE_INT  , FSXM , 1);
        LOAD_FIELD(&pcr, FSYNC_MODE_INT  , FSRM , 1);
        LOAD_FIELD(&pcr, CLK_MODE_INT    , CLKXM, 1);
        LOAD_FIELD(&pcr, CLK_MODE_INT    , CLKRM, 1);
#else
        LOAD_FIELD(&pcr, FSYNC_MODE_EXT  , FSXM , 1);
        LOAD_FIELD(&pcr, FSYNC_MODE_EXT  , FSRM , 1);
        LOAD_FIELD(&pcr, CLK_MODE_EXT    , CLKXM, 1);
        LOAD_FIELD(&pcr, CLK_MODE_EXT    , CLKRM, 1);
#endif
        LOAD_FIELD(&pcr, FSYNC_POL_HIGH  , FSXP , 1);
        LOAD_FIELD(&pcr, FSYNC_POL_HIGH  , FSRP , 1);
        LOAD_FIELD(&pcr, CLKX_POL_RISING , CLKXP, 1);
        LOAD_FIELD(&pcr, CLKR_POL_FALLING, CLKRP, 1);

        /* Set up Receive Control Register */
        LOAD_FIELD(&rcr, DUAL_PHASE     , RPHASE, 1);
        LOAD_FIELD(&rcr, FRAME_IGNORE   , RFIG , 1);
        LOAD_FIELD(&rcr, DATA_DELAY1    , RDATDLY, RDATDLY_SZ);
        LOAD_FIELD(&rcr, 0             , RFRLEN1, RFRLEN1_SZ);
```

```
        LOAD_FIELD(&rcr, 0                  , RFRLEN2, RFRLEN2_SZ);
        LOAD_FIELD(&rcr, WORD_LENGTH_32    , RWDLEN1, RWDLEN1_SZ);
        LOAD_FIELD(&rcr, WORD_LENGTH_32    , RWDLEN2, RWDLEN2_SZ);
        LOAD_FIELD(&rcr, NO_COMPAND_MSB_1ST    , RCOMPAND, RCOMPAND_SZ);


        /* Set up Transmit Control Register */
        LOAD_FIELD(&xcr, DUAL_PHASE     , XPHASE, 1);
        LOAD_FIELD(&xcr, FRAME_IGNORE    , XFIG , 1);
        LOAD_FIELD(&xcr, DATA_DELAY1     , XDATDLY, XDATDLY_SZ);
        LOAD_FIELD(&xcr, 0                 , XFRLEN1, XFRLEN1_SZ);
        LOAD_FIELD(&xcr, 0                 , XFRLEN2, XFRLEN2_SZ);
        LOAD_FIELD(&xcr, WORD_LENGTH_32   , XWDLEN1, XWDLEN1_SZ);
        LOAD_FIELD(&xcr, WORD_LENGTH_32   , XWDLEN2, XWDLEN2_SZ);
        LOAD_FIELD(&xcr, NO_COMPAND_MSB_1ST, XCOMPAND, XCOMPAND_SZ);


        /* Set up Sample Rate Generator Register */
#ifndef SLAVE
        SET_BIT(&srgr, CLKSM);                 /* CLKG derived from CPU clock*/
        LOAD_FIELD(&srgr, FSX_FSG, FSGM, 1);
        LOAD_FIELD(&srgr, 70, CLKGDV, CLKGDV_SZ);
        LOAD_FIELD(&srgr, 63, FPER, FPER_SZ);
        LOAD_FIELD(&srgr, 31, FWID, FWID_SZ);
#endif
        /* Store McBSP 0 registers */
        mcbsp_init(0, spcr, rcr, xcr, srgr, mcr, rcer, xcer, pcr);

        /* Bring McBSP out of reset */
        MCBSP_SAMPLE_RATE_ENABLE(0);    /* Start Sample Rate Generator */
        MCBSP_FRAME_SYNC_ENABLE(0);     /* Enable Frame Sync pulse      */

} /* End config_serial */


void start_cs4226(void)
{
unsigned int spcr = 0;
unsigned int rcr  = 0;
unsigned int xcr  = 0;
unsigned int srgr = 0;
unsigned int mcr  = 0;
unsigned int rcer = 0;
unsigned int xcer = 0;
unsigned int pcr  = 0;
clock_t ctemp;

        /* Set up Pin Control Register */
        LOAD_FIELD(&pcr, FSYNC_MODE_INT  , FSXM , 1);
        LOAD_FIELD(&pcr, FSYNC_MODE_INT  , FSRM , 1);
        LOAD_FIELD(&pcr, CLK_MODE_INT    , CLKXM, 1);
        LOAD_FIELD(&pcr, CLK_MODE_INT    , CLKRM, 1);
        LOAD_FIELD(&pcr, FSYNC_POL_LOW   , FSXP , 1);
        LOAD_FIELD(&pcr, FSYNC_POL_LOW   , FSRP , 1);
        LOAD_FIELD(&pcr, CLKX_POL_FALLING , CLKXP, 1);
        LOAD_FIELD(&pcr, CLKR_POL_RISING, CLKRP, 1);

        /* Set up Receive Control Register */
        LOAD_FIELD(&rcr, SINGLE_PHASE    , RPHASE, 1);
        LOAD_FIELD(&rcr, FRAME_IGNORE    , RFIG , 1);
```

```
        LOAD_FIELD(&rcr, DATA_DELAY1       , RDATDLY, RDATDLY_SZ);
        LOAD_FIELD(&rcr, 0                 , RFRLEN1, RFRLEN1_SZ);
        LOAD_FIELD(&rcr, WORD_LENGTH_24    , RWDLEN1, RWDLEN1_SZ);
        LOAD_FIELD(&rcr, NO_COMPAND_MSB_1ST    , RCOMPAND, RCOMPAND_SZ);

        /* Set up Transmit Control Register */
        LOAD_FIELD(&xcr, SINGLE_PHASE     , XPHASE, 1);
        LOAD_FIELD(&xcr, FRAME_IGNORE     , XFIG , 1);
        LOAD_FIELD(&xcr, DATA_DELAY1      , XDATDLY, XDATDLY_SZ);
        LOAD_FIELD(&xcr, 0                , XFRLEN1, XFRLEN1_SZ);
        LOAD_FIELD(&xcr, WORD_LENGTH_24   , XWDLEN1, XWDLEN1_SZ);
        LOAD_FIELD(&xcr, NO_COMPAND_MSB_1ST, XCOMPAND, XCOMPAND_SZ);

        /* Set up Serial Port Control Register */
        LOAD_FIELD(&spcr, INTM_RDY        , XINTM, XINTM_SZ );
        LOAD_FIELD(&spcr, INTM_RDY        , RINTM, RINTM_SZ );
        LOAD_FIELD(&spcr, 2, CLKSTP, 1);

        /* Set up Sample Rate Generator Register */
        SET_BIT(&srgr, CLKSM);                  /* CLKG derived from CPU clock*/
        LOAD_FIELD(&srgr, FSX_DXR_TO_XSR, FSGM, 1);
        LOAD_FIELD(&srgr, 70, CLKGDV, CLKGDV_SZ);

        /* Store McBSP 1 registers */
        mcbsp_init(1, spcr, rcr, xcr, srgr, mcr, rcer, xcer, pcr);

        /* Bring McBSP out of reset */
        MCBSP_SAMPLE_RATE_ENABLE(1);    /* Start Sample Rate Generator */
        MCBSP_FRAME_SYNC_ENABLE(1);     /* Enable Frame Sync pulse      */
        MCBSP_ENABLE(1, MCBSP_RX);      /* Bring Receive out of reset   */
        MCBSP_ENABLE(1, MCBSP_TX);      /* Bring Transmit out of reset */


        /* Program CS4226 registers */

        XMIT_done=0; /* wait for McBSP to initialize */
        while(!XMIT_done);

#ifndef SLAVE
        XMIT_done=0; /* Clock is PLL driven by LRCK at 1 Fs, CLKOUT = 1 Fs */
        MCBSP1_DXR = 0x200162;
#else
        XMIT_done=0; /* Clock is ext oscillator, CLKOUT = 1 Fs    */
        MCBSP1_DXR = 0x200160;
#endif
        while(!XMIT_done);

        XMIT_done=0;    /* Mute all DACs but 1 & 2 (stereo pair 1) */
        MCBSP1_DXR = 0x2003FC;
        while(!XMIT_done);

        XMIT_done=0; /* No DAC attenuation */
        MCBSP1_DXR = 0x200400;
        while(!XMIT_done);
        XMIT_done=0;        /* No DAC attenuation */
        MCBSP1_DXR = 0x200500;
        while(!XMIT_done);
        XMIT_done=0; /* No DAC attenuation */
```

```
        MCBSP1_DXR = 0x200600;
        while(!XMIT_done);
        XMIT_done=0; /* No DAC attenuation */
        MCBSP1_DXR = 0x200700;
        while(!XMIT_done);
        XMIT_done=0; /* No DAC attenuation */
        MCBSP1_DXR = 0x200800;
        while(!XMIT_done);
        XMIT_done=0; /* No DAC attenuation */
        MCBSP1_DXR = 0x200900;
        while(!XMIT_done);

#ifndef SLAVE
        XMIT_done=0; /* C4226 is slave, I2S format, 64 bit clocks per Fs */
        MCBSP1_DXR = 0x200ECC;
#else
        XMIT_done=0; /* C4226 is Master, I2S format, 64 bit clocks per Fs */
        MCBSP1_DXR = 0x200EEC;
#endif
        while(!XMIT_done);

        XMIT_done=0;      /* Pull device out of reset */
        MCBSP1_DXR = 0x200200;
        while(!XMIT_done);

        /* Wait 90ms for PLL to lock onto the LRCK (FSX) */
        ctemp=clock();
        while(clock() < ctemp + 90);


} /* End start_cs4226 */



/* McBSP verification test code.                              */
void
main (void)
{
int i;

        set_interrupts();
        start_sp_dma();
        config_serial();
        start_cs4226();

        MCBSP_ENABLE(0, MCBSP_RX);      /* Bring Receive out of reset  */
        MCBSP_ENABLE(0, MCBSP_TX);      /* Bring Transmit out of reset */

        while(1){

                /* Set up DMA reload registers for the next block */
          DMA_GADDR_B = (unsigned int)MEM_SRC + 0x400;
          DMA_GADDR_C = (unsigned int)MEM_DST + 0x400;

                /* Wait for current block to finish */
                while(!DMA_done[0]);
                DMA_done[0]=0;
```

```
        /* Transfer the recently completed block from the input buffer */
        /* to the output buffer. Here is where any algorithms to do DSP*/
        /* on the data would go.              */
                for ( i = 0; i < 0x100; i++){
                    *(int *) (MEM_SRC + 4*i) =
                        *(int *) (MEM_DST + 4*i);

                    *(int *) (MEM_SRC + 4*i + 0x800) =
                        *(int *) (MEM_DST + 4*i + 0x800);
                }

                /* Set up DMA reload registers for the next block */
        DMA_GADDR_B = (unsigned int)MEM_SRC;
        DMA_GADDR_C = (unsigned int)MEM_DST;

                /* Wait for current block to finish */
                while(!DMA_done[0]);
                DMA_done[0]=0;

        /* Transfer the recently completed block from the input buffer */
        /* to the output buffer. Here is where any algorithms to do DSP*/
        /* on the data would go.              */
                for ( i = 0; i < 0x100; i++){
                    *(int *) (MEM_SRC + 4*i + 0x400) =
                        *(int *) (MEM_DST + 4*i + 0x400);

                    *(int *) (MEM_SRC + 4*i + 0xC00) =
                        *(int *) (MEM_DST + 4*i + 0xC00);
                }

        }

} /* end main */


/********************************************************************/
/*    dma_int.c                                                     */
/********************************************************************/

#include <intr.h>
#include <dma.h>

/* Global variables */
extern int DMA_done[4];
extern int RECV_done;
extern int XMIT_done;

/* Prototypes */
interrupt void DMA_Ch0_ISR(void);
interrupt void RINT_ISR(void);
interrupt void XINT_ISR(void);
void set_interrupts(void);

/* DMA Ch0 ISR used to clear block condition and flag when the      */
/* transfer has completed.                                          */
interrupt void
DMA_Ch0_ISR(void)
{
```

```
unsigned int sec_ctrl = 0x50000;

        sec_ctrl = REG_READ(DMA0_SECONDARY_CTRL_ADDR);
        if (GET_BIT(&sec_ctrl, BLOCK_COND)){
                DMA_done[0] = 1;
                RESET_BIT(&sec_ctrl, BLOCK_COND);
        }
        REG_WRITE(DMA0_SECONDARY_CTRL_ADDR, sec_ctrl);

} /* End DMA_Ch0_ISR */

interrupt void
XINT_ISR(void)
{

                XMIT_done=1;

}

interrupt void
RINT_ISR(void)
{

                RECV_done=1;

}
/* Routine to enable DMA and Timer interrupt service routines      */
void
set_interrupts(void)
{
        intr_init();
        intr_map(CPU_INT8, ISN_DMA_INT0);
        intr_hook(DMA_Ch0_ISR, CPU_INT8);
        intr_map(CPU_INT11, ISN_XINT1);
        intr_hook(XINT_ISR, CPU_INT11);
        intr_map(CPU_INT12, ISN_RINT1);
        intr_hook(RINT_ISR, CPU_INT12);

        INTR_GLOBAL_ENABLE();
        INTR_ENABLE(CPU_INT_NMI);
        INTR_ENABLE(CPU_INT8);
        INTR_ENABLE(CPU_INT11);
        INTR_ENABLE(CPU_INT12);

} /* End set_interrupts */
```

References

TMS320C6201 Digital Signal Processor data sheet, Texas Instruments, SPRS051C, March 1998.

TMS320C6201/C6701 Peripherals Reference Guide, Texas Instruments, SPRU190B, March 1998.

TMS320C6000 Peripheral Support Library Programmer's Reference, Texas Instruments, SPRU273B, July 1998.

TMS320C6000 DMA Example Applications Application Report, Texas Instruments, SPRA529.

IMPORTANT NOTICE

.