Chapter 2 Learning to Use the Hardware and Software

Contents

Chapter 2
Learning to Use the Hardware and Software Tools by Generating a Sine Wave

The directory `C:\C6701\evmstart` contains two example files that you can use as a starting point for all your projects.

A Sample Linker Command File

```
/****************************************************************/
/*  File c:\c6701\evmstart\evmlink.cmd                          */
/*    This linker command file can be used as the starting      */
/*  point for linking programs for the TMS320C6701 EVM.  It     */
/*  assumes that memory MAP 1 has been selected.  Almost all    */
/*  sections have been allocated to internal memory. If         */
/*  more memory is needed sections can be mapped to external    */
/*  memory.                                                     */
/*     The external SBSRAM has been divided into program        */
/* (SBSRAM_PROG_MEM) and data (SBSRAM_DATA_MEM) memory. The     */
/* lengths of each type of memory can be changed as desired.    */
/****************************************************************/

-c
-heap  0x1000
-stack 0x800

MEMORY
{
  /* Internal program memory, 16K 32-bit instructions  */
  INT_PROG_MEM (RX)    : origin = 0x00000000 length = 0x00010000

  /* Internal data memory, 64K bytes */
  INT_DATA_MEM (RW)    : origin = 0x80000000 length = 0x00010000
```

A Sample Linker Command File (cont.)

```
  /* External synchronous burst static RAM. 256K bytes  */
  SBSRAM_PROG_MEM (RX) : origin = 0x00400000 length = 0x00020000
  SBSRAM_DATA_MEM (RW) : origin = 0x00420000 length = 0x00020000

  /* External synchronous dynamic RAM, 8M bytes  */
  SDRAM0_DATA_MEM (RW) : origin = 0x02000000 length = 0x00400000
  SDRAM1_DATA_MEM (RW) : origin = 0x03000000 length = 0x00400000
}

SECTIONS
{
  .vec:     load = 0x00000000 /* Interrupt vectors included */
                              /*by using dev6x.lib  */
/* Use SBSRAM_PROG_MEM for .text if it cannot fit in INT_PROG_MEM */
/*  .text:    load = SBSRAM_PROG_MEM */ /* Executable code */

  .text:    load = INT_PROG_MEM
  .const:   load = INT_DATA_MEM /* Initialized constants */
  .bss:     load = INT_DATA_MEM /* Global and static variables */
  .data:    load = INT_DATA_MEM /* Data from .asm programs */
  .cinit:   load = INT_DATA_MEM /* Tables for initializing */
                                /* variables and constants */
  .stack:   load = INT_DATA_MEM /* Stack for local variables */
  .far:     load = INT_DATA_MEM /* Global and static variables */
                                /* declared far */
  .sysmem:  load = SDRAM0_DATA_MEM /* Used by malloc, etc. */
  .cio:     load = INT_DATA_MEM /* Used for C I/O functions */
  .ipmtext: load = INT_PROG_MEM /* Used by dev6x.lib */
}
```

C Program to Use as a Starting Point

```
/*********************************************************/
/* Program: evmstart.c                                   */
/* This program can be used as the starting point for all  */
/* DSP programs for ENEE 428.  It initializes the C6701    */
/* EVM board, configures the McBSP0 serial port, and then  */
/* configures the stereo codec. It then enters an infinite */
/* loop that reads a sample from the codec A/D and loops it*/
/* back out to the codec D/A.  This loop should be replaced*/
/* by the code to achieve the goals of the experiment.     */
/*********************************************************/

#include <stdio.h>
#include <stdlib.h>

/*********************************************************/
/* For using the TI DSP support software. See             */
/*  TMS320C6201/6701 Evaluation Module Technical Reference,*/
/*     SPRU305, Chapter 3.                                 */
/*  TMS320C6x Peripheral Support Library Programmer's      */
/*     Reference, SPRU273B.                                */
/* These headers are in C:\C6701\evm6x\dsp\include         */
/*********************************************************/
#include <common.h>   /* macros for bit fields */
#include <board.h>    /* for components on EVM board */

#include <mcbspdrv.h> /* serial port drivers  */
#include <codec.h>    /* for codec configuration */
/********************************************************/

#include <math.h>

/* NOTE:  The TI compiler gives warnings if math.h is moved up
          under stdlib.h  */
```

evmstart.c (cont.)

```
#define  PORT_0 0
#define  sampling_rate 16000

void main(void){
   Mcbsp_dev      bspH;          /* handle to serial port  */
   Mcbsp_config   mcbspConfig;   /* config obj: mcbsp regs */
/*   int  act_samp_rate;  */
   /*****************************************************/
   /* The object module for evm_init() is located in:     */
   /*    C:\C6701\evm6x\dsp\lib\drivers\drv6x.lib         */
   /* It initializes the EVM for use by determining the   */
   /* C6x map mode, setting the default EMIF to the EVM   */
   /* configuration, and setting external peripheral base */
   /* addresses which are dependent on the map mode       */
   /*****************************************************/
   evm_init();
   mcbsp_drv_init(); /* init the MCBSPdrv library */
                     /* (in drv6x.lib)   */
   bspH = mcbsp_open(0); /* open mcbsp & return handle */
                         /* (prototyped in mcbspdrv.h) */
   mcbsp_reset(bspH);    /* reset the serial port */

/********************************************************/
/* configure serial port configuration structure        */
/********************************************************/
   memset(&mcbspConfig, 0, sizeof(mcbspConfig));
                /* Initialize all structure elements to 0 */
   mcbspConfig.loopback                = DLB_DISABLE;

   mcbspConfig.tx.update               = TRUE;
   mcbspConfig.tx.interrupt_mode       = INTM_RDY;
   mcbspConfig.tx.clock_polarity       = CLKX_POL_RISING;
   mcbspConfig.tx.frame_sync_polarity = FSYNC_POL_HIGH;
   mcbspConfig.tx.clock_mode           = CLK_MODE_EXT;
   mcbspConfig.tx.frame_sync_mode      = FSYNC_MODE_EXT;
   mcbspConfig.tx.phase_mode           = SINGLE_PHASE;
```

evmstart.c (cont.)

```
mcbspConfig.tx.frame_length1       = 0;
mcbspConfig.tx.frame_length2       = 0;
mcbspConfig.tx.word_length1        = WORD_LENGTH_32;
mcbspConfig.tx.word_length2        = 0;
mcbspConfig.tx.companding          = NO_COMPAND_MSB_1ST;
mcbspConfig.tx.frame_ignore        = NO_FRAME_IGNORE;
mcbspConfig.tx.data_delay          = DATA_DELAY0;

mcbspConfig.rx.update              = TRUE;
mcbspConfig.rx.interrupt_mode      = INTM_RDY;
mcbspConfig.rx.justification       = RXJUST_RJZF;
mcbspConfig.rx.clock_polarity      = CLKR_POL_FALLING;
mcbspConfig.rx.frame_sync_polarity = FSYNC_POL_HIGH;
mcbspConfig.rx.clock_mode          = CLK_MODE_EXT;
mcbspConfig.rx.frame_sync_mode     = FSYNC_MODE_EXT;
mcbspConfig.rx.phase_mode          = SINGLE_PHASE;
mcbspConfig.rx.frame_length1       = 0;
mcbspConfig.rx.frame_length2       = 0;
mcbspConfig.rx.word_length1        = WORD_LENGTH_32;
mcbspConfig.rx.word_length2        = 0;
mcbspConfig.rx.companding          = NO_COMPAND_MSB_1ST;
mcbspConfig.rx.frame_ignore        = NO_FRAME_IGNORE;
mcbspConfig.rx.data_delay          = DATA_DELAY0;

/* Now call mcbsp_config() to load the McBSP registers  */
/* according to the structure values.                   */

mcbsp_config( bspH, &mcbspConfig );

/* If you uncomment the next statement, you will have   */
/* to map .text to SBSRAM_PROG_MEM  because it will not */
/* fit in INT_PROG_MEM.                                 */

/* printf("McBSP0 configuration completed\n"); */
```

evmstart.c (cont.)

```
   /*****************************************************/
   /*  Configure the CODEC                              */
   /*    These routines reside in drv6x.lib             */
   /*****************************************************/
   codec_init();

   /* A/D 0 dB gain, no 20dB mic gain, sel (L/R)LINE input */
   codec_adc_control(LEFT,  0.0, FALSE, LINE_SEL);
   codec_adc_control(RIGHT, 0.0, FALSE, LINE_SEL);

   /* mute (L/R)LINE input to mixer                     */
   codec_line_in_control(LEFT,  MIN_AUX_LINE_GAIN, TRUE);
   codec_line_in_control(RIGHT, MIN_AUX_LINE_GAIN, TRUE);
  /* D/A 0.0 dB atten, do not mute DAC outputs         */
   codec_dac_control(LEFT,  0.0, FALSE);
   codec_dac_control(RIGHT, 0.0, FALSE);

   /* Set codec data format (mono=FALSE, stereo=TRUE)    */
   codec_audio_data_format(LINEAR_16BIT_SIGNED_LE, TRUE, BOTH);

   /* Set the sampling rate. The rate is quantized to the */
   /* nearest of the following values (in kHz):          */
   /* 5.5125, 6.6150, 8.0000, 96.000, 11.025, 16.0000,   */
   /* 189000, 22.0500, 27.4286, 32.0000, 33.0750, 37.8000,*/
   /* 44.1000, 48.0000                                   */

 /*  act_samp_rate = codec_change_sample_rate(sampling_rate,TRUE); */

   codec_change_sample_rate(sampling_rate,TRUE);

/* Disable Codec interrupts */
   codec_interrupt_disable();

/*  printf("The codec sampling rate is %d\n", act_samp_rate); */
```

evmstart.c (cont.)

```
/*********************************************************/
/* Turn on the serial port transmitter and receiver       */
/*********************************************************/

   MCBSP_ENABLE(PORT_0, MCBSP_RX|MCBSP_TX);

/*********************************************************/
/* Main program: Replace with your code                   */
/*********************************************************/

   for(;;){
      /* Poll XRDY bit, then read & write data   */
      while(!MCBSP_XRDY(PORT_0));
      MCBSP_WRITE(PORT_0, MCBSP_READ(PORT_0));
    }
}
```

Creating a CCS Project for evmstart.c

- Start CCS and click on Project, select New, and fill out the boxes as follows:

Project Name:	*give it a name*
Location:	*a directory in your private workspace*
Project type:	Executable (.out)
Target:	TMS320C67xx

- Add the following library files to the project:

```
C:\C6701\evm6x\dsp\lib\devlib\dev6x.lib
C:\C6701\evm6x\dsp\lib\drivers\drv6x.lib
C:\ti\c6000\cgtools\lib\rts6701.lib
```

- Add the linker command file:

```
C:\C6701\evmstart\evmlink.cmd
```

- Add the C source file:

```
C:\C6701\evmstart\evmstart.c
```

Creating a Project for evmstart.c
Setting the Build Options

Click on Project and select Build Options. Enter the following options:

Compiler → Basic

Target Version:	670x
Generate Debug Info:	Full Symbolic Debug (-g)
Opt Speed vs Size:	Speed Most Critical (ms)
Opt Level:	None
Program Level Opt:	None

Compiler → Preprocessor

Include Search Path (-i):	`C:\C6701\evm6x\dsp\include`
Define Symbols (-d):	`CHIP_6701`
Preprocessing:	None

Compiler → Files

Asm Directory:	*a directory in your workspace*
Obj Directory:	*a directory in your workspace*

Setting the Build Options (cont.)

Linker → Basic

Output Filename (-o):	evmstart.out (You can change this.)
Map Filename (-m):	evmstart.map (You can change this.)
Autoinit Model:	Run-time Autoinitialization

A Simple First Experiment

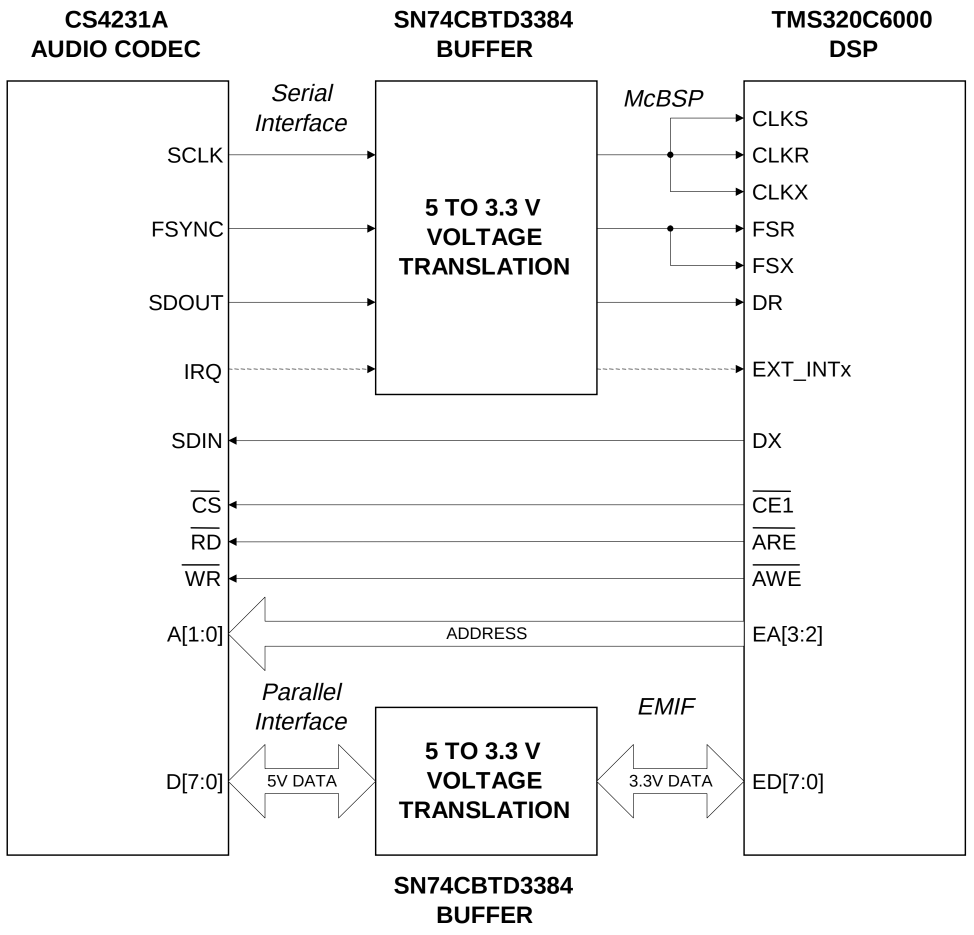
- When the project has been created, build the executable module by clicking on the Rebuild All icon or Project → Rebuild All.
- Load the program using File → Load Program

The program, evmstart, simply loops the A/D input back to the D/A output. Check this by doing the following:

A Simple First Experiment (cont.)

- Plug a stereo cable into the EVM line input and connect both channels to the same signal generator output. The program evmstart.out sets the codec to sample at 16000 Hz, so set the signal generator to output a sine wave of less than 8000 Hz.
- Plug a stereo cable into the EVM line output. Connect the left and right outputs to two different oscilloscope channels.
- Start the program running and check that the sine waves appear on the scope. Make sure the input level is small enough so that there is no clipping.
- Vary the sine wave frequency. What happens when it is more than 8000 Hz? Measure the amplitude response of the system by varying the input frequency and dividing the output amplitude by the input amplitude. Plot the response.

The Codec to ’C6701 McBSP0 Interface



Brian G. Carlson and Vassos Soteriou, “TMS320C6201/6701 EVM: TMS320C6000 McBSP to Multimedia Audio Codec Interface,” SPRA477, Figure 1, p. 4, August 2001

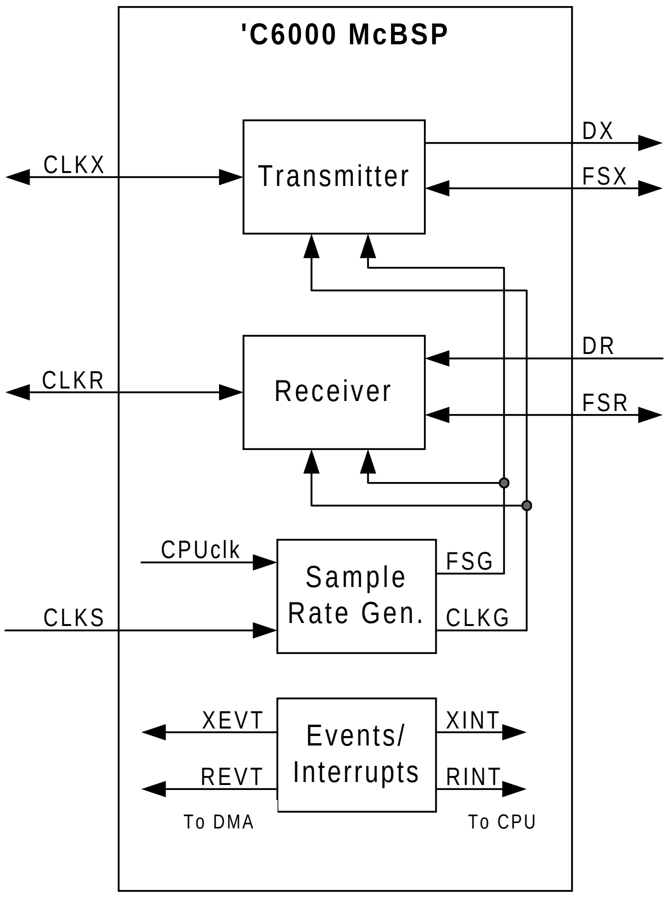
Multichannel Buffered Serial Port (McBSP) Properties

- Can generate shift clocks and frame sync signals internally, or use external signals (EVM uses external ones)
- μ-law and A-law hardware companding options
- Multichannel selection of up to 32 elements from a 128 element TDMA frame

CS4231A Codec Properties

- Stereo A/D and D/A converters with filters
- Fourteen possible sampling rates
- Serial sample and parallel control interface

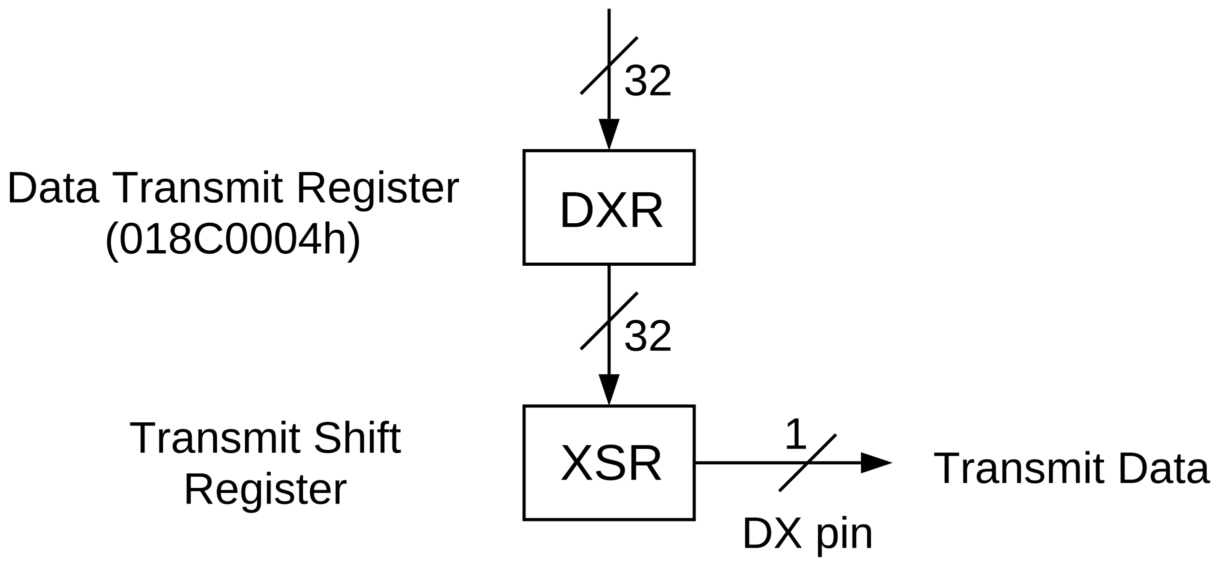
McBSP Block Diagram



DX/DR Serial transmit/receive data
FSX/FSR Transmit/receive frame sync
CLKX/CLKR Transmit/receive serial shift clock
XINT/RINT Transmit/receive interrupt to CPU
XEVT/REVT Transmit/receive interrupt to DMA
CLKS External clock for Sample Rate Gen.

Shaku Anjanaiah and Brad Cobb, “TMS320C6000 McBSP Initialization,” SPRA488A, September 2001, Figure 1, p. 2.

McBSP Transmitter Block Diagram



Serial Port Control Register (018C0008h)

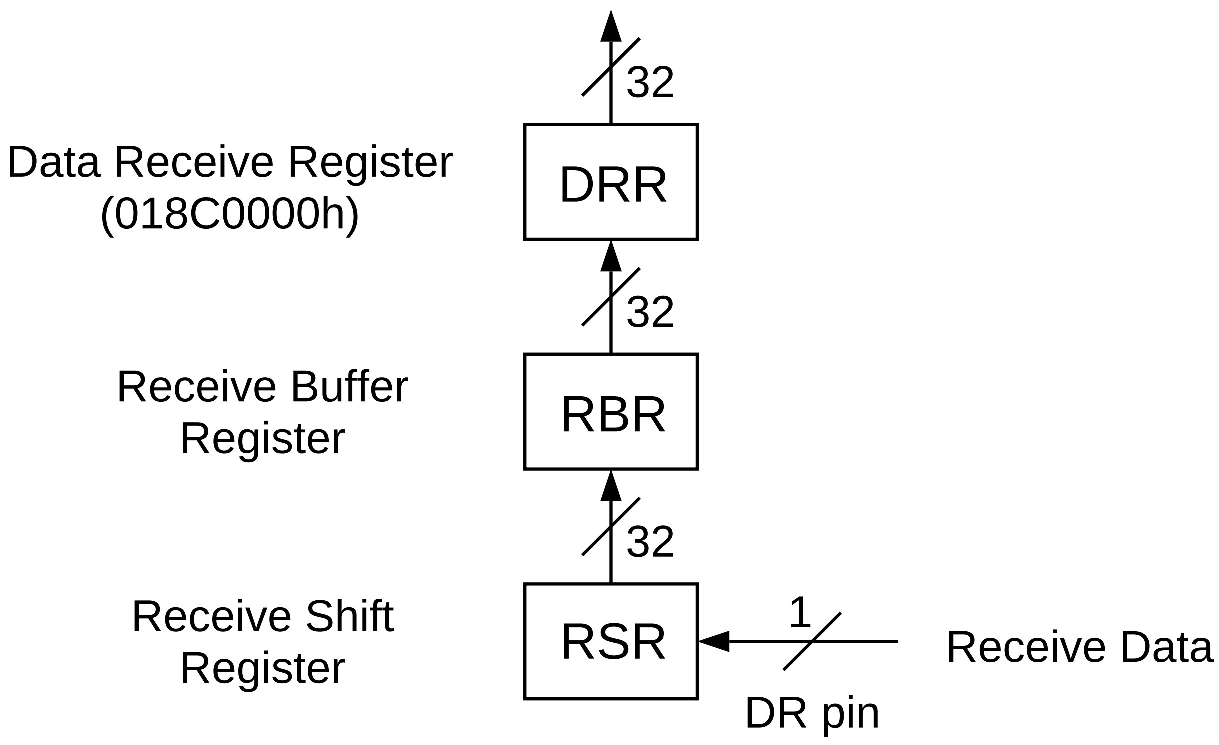
	XINTM		XEMPTY	XRDY	XRST		RJUST		RINTM		RFULL	RRDY	RRST
	21 20		18	17	16		14 13		5 4		2	1	0

Note: Addresses are for McBSP0

Operation of Serial Port Transmitter

- The CPU or DMA writes a word into the Data Transmit Register (DXR). The XRDY flag is cleared whenever data is written to the DXR.
- After a word (32 bits in our case) is shifted out of Transmit Shift Register (XSR), a parallel transfer of the DXR into the XSR is performed. The XRDY flag is set when the transfer occurs.
- The serial port transmitter sends an interrupt request (XINT) to the CPU when the XRDY flag makes a transition from 0 to 1 if XINTM = 00b in the SPCR. It also sends a Transmit Event Notice (XEVT) to the DMA.

McBSP Receiver Block Diagram



Note: Addresses are for McBSP0

Operation of Serial Port Receiver

- RX bits shift serially into the Receive Shift Register (RSR).
- When a full element is received, the 32-bit RSR is transferred in parallel to the Receive Buffer Register (RBR) if it is empty.
- The RBR is copied to the Data Receive Register (DRR) if it is empty.
- The RRDY bit in SPCR is set to 1 when RSR is moved to DRR, and it is cleared when DRR is read.
- When RRDY transitions from 0 to 1, the McBSP generates a CPU interrupt request (RINT) if RINTM = 00b in the SPCR. A receive event (REVT) is also sent to the DMA controller.

More on the CS4231A Codec

The Fourteen Sampling Rates

Two different crystals are attached to the codec which allows the following sampling rates (in kHz) to be selected:

5.5125	6.6150	8.0000	9.6000
11.0250	16.0000	18.9000	22.0500
27.4286	32.0000	33.0750	37.8000
44.1000	48.0000		

Word Formats

- We configure the codec to generate 16-bit linear samples.
- Stereo Mode
  - Configure McBSP0 for a one phase frame containing one 32-bit word.
  - The most significant 16-bits are the left channel sample and the least significant 16-bits are the right channel sample.

Codec Word Formats (cont.)

- Mono Mode
  - Configure McBSP0 for a one phase frame containing one 32-bit word.
  - The right channel ADC input is set to zero.
  - The left channel output sample is sent to both DAC channels.

DAC and ADC Filters

See pages 45 and 46 of the CS4231A reference given below for plots of the DAC and ADC filter amplitude responses. They are lowpass filters with a cutoff frequency of half the sampling rate. The responses automatically scale with the sampling rate.

For complete details on the codec see:
Crystal Semiconductor Corp., "CS4231A Parallel Interface, Multimedia Audio Codec," DS139PP2, September, 1994.

C6x Peripheral Support Library

The C6x Peripheral Support Library (drv6x.lib and dev6x.lib) contains C callable functions and macros for configuring and interfacing with the EVM, codec, and DSP peripherals. (TI is not manufacturing the TMS320C6701 EVM any longer and has not included these libraries in the newest versions of CCS.)

The libraries are at:

- `C:\C6701\evm6x\dsp\lib\drivers\drv6x.lib`
- `C:\C6701\evm6x\dsp\lib\devlib\dev6x.lib`

For complete documentation see:

1. *TMS320C6202/6701 EVM Technical Reference*, SPRU305.
2. *TMS320C6x Peripheral Suport Library Programmer's Reference*, SPRU273.

Selected Codec and EVM Routines

`codec_init()`	Initialize to default config
`codec_reset()`	Reset the codec
`codec_adc_control()`	Control input A/D converters
`codec_dac_control()`	Control D/A converter output
`codec_audio_data_format()`	Set data format
`codec_playback_enable()`	Enable playback mode
`codec_playback_disable()`	Disable playback mode
`codec_capture_enable()`	Enable capture mode
`codec_interrupt_enable()`	Enable codec to gen interrupts
`codec_serial_port_enable()`	Enable SDOUT and SDIN
`codec_change_sample_rate()`	Change sampling rates
`evm_init()`	Initialize EVM for use
`evm_codec_enable()`	Enable the codec

Selected Serial Port Routines

`MCBSP_ENABLE()`	Enables McBSP tx, rcv, or both
`MCBSP_TX_RESET()`	Reset McBSP transmitter
`MCBSP_RX_RESET()`	Reset McBSP receiver
`MCBSP_READ()`	Read data value from DRR
`MCBSP_WRITE()`	Write data value to DXR
`MCBSP_RRDY()`	Returns McBSP RRDY bit
`MCBSP_XRDY()`	Returns McBSP XRDY bit
`mcbsp_init()`	Initializes McBSP registers
`mcbsp_open()`	Open McBSP for subsequent calls
`mcbsp_close()`	Releases port
`mcbsp_config()`	Configures MCBSP for operation
`mcbsp_reset()`	Reset MCBSP

Example C Code for Stereo Read

```
   int data;
   float left, right;

/* Wait for  RRDY to be set, then read
      input sample */
   while(!MCBSP_RRDY(0));
   data  = MCBSP_READ(0);

/* Shift right to move left ch 16 bits to
     bits 15 - 0 and extended sign into
     bits 31 - 16. Then convert to float. */
   left  =  data >> 16;

/* Shift left by 16 to lop off left ch and
     then right by 16 to sign extend.
     Convert to float. */
   right = data << 16 >> 16;
```

Example C Code for Stereo Write

```
   float left,  right;
   int   ileft, iright, sample;

/* Convert left and right values to integers */
   ileft  = (int) left;
   iright = (int) right;

/* Combine L/R samples into a 32-bit word */
   sample = (ileft<<16)|(iright & 0x0000FFFF);

/* Poll XRDY bit until DXR empty */
   while(!MCBSP_XRDY(0));
   MCBSP_WRITE(0, sample);
```

Experiment 2 (Part 1) Generating Sine Waves Using XRDY Polling

For Experiment 2 (Part 1) do the following:

1. Set the sampling rate to 8 kHz.
2. Set the codec to stereo mode.
3. Generate a 1 kHz sine wave on the left channel and a 2 kHz sine wave on the right channel. Remember that $|\sin(x)| \leq 1$ and that floats less than 1 become 0 when converted to ints. Therefore, scale your floating point sine wave samples to make them greater than 1 and fill reasonable part of the D/A dynamic range before converting them to ints.
4. Combine the left and right channel integer samples into a 32-bit integer and write the resulting word to the McBSP0 DXR using polling of the XRDY flag.

Experiment 2 (Part 1) (cont.)

5. Observe the left and right channel outputs on two oscilloscope channels.
6. Verify that the sine wave frequencies observed on the scope are the desired values by measuring their periods.
7. Use the signal generator to measure the frequencies. Set the EXT-INT switch to EXT and attach a channel of the codec output to the EXT COUNTER input connector.
8. When you have verified that your program is working, change the left channel frequency to 7 kHz and the right channel frequency to 6 kHz. Measure the D/A output frequencies. Explain your results. (Hint: Look up "aliasing" in any reference on digital signal processing.)

Generating Samples of a Sine Wave

Continuous Time Sine Wave

$$s(t) = \sin 2\pi f_0 t$$

Sampled Sine Wave

Let $f_s = 1/T$ be the sampling rate where T is the sampling period.

$$\begin{aligned} s(nT) &= \sin 2\pi f_0 nT = \sin 2\pi \frac{f_0}{f_s} n \\ &= \sin n\Delta\theta \end{aligned}$$

where $\Delta\theta = 2\pi f_0 / f_s$

Recursive Angle Generation

Let

$$\theta(n) = n\Delta\theta$$

Then

$$\theta(n+1) = (n+1)\Delta\theta = n\Delta\theta + \Delta\theta = \theta(n) + \Delta\theta$$

Sample Program Segment for Polling

```
#include <math.h>
#define  pi  3.141592653589
int sample = 0;
float fs = 8000.;
float f0 = 1000.;
float delta = 2.*pi*f0/fs;
float twopi = 2.0*pi;
float angle = 0;
float left;

for(;;){        /*  Infinite loop       */
   left = 15000.0*sin(angle);
                      /* Scale for D/A  */
   sample = ((int) left) <<16;
                     /* Put in top half */
   while(!MCBSP_XRDY(0)); /* Poll XRDY  */
   MCBSP_WRITE(0, sample);/* Write to DXR*/
   angle += delta;
   if( angle >= twopi) angle - = twopi;
     /*  Keep angle from overflowing   */
}
```

Some Important Information

- Remember to include `math.h` in your C program.
- The back of the EVM has stereo input and output jacks. The lab has stereo cables to convert from the stereo plug to the 3 wires: ground, left channel and right channel. The banana connectors can be used to connect the stereo cables to the oscilloscope. The 2 alligator clips to BNC cables can also be used.
- Stereo cable wires
  - **Left Channel:** White wire
  - **Right Channel:** Red wire
  - **Ground:** Bare wire

Method 2 for Generating a Sine Wave – Using Interrupts

Almost all the time in the polling method is spent sitting in a loop waiting for the XRDY flag to get set. A much more efficient approach is to let the DSP perform all sorts of desired tasks *in the background* and have the serial port interrupt these background tasks when it needs a sample to transmit. The *interrupt service routine* is called a *foreground task.*

The TMS320C6701 contains a vectored priority interrupt controller.

- The highest priority interrupt is RESET which cannot be masked.
- The next priority interrupt is the Non-Maskable Interrupt (NMI) which is used to alert the DSP of a serious hardware problem.

Using Interrupts (cont. 1)

- There are two reserved interrupts and 12 additional maskable CPU interrupts. The peripherals, such as the timers, serial ports, and DMA controller, plus the four external interrupt pins present a set of 16 interrupt sources. The 16 TMS320C6701 interrupt sources are shown in the table on Slide 33.
- The interrupt system includes a multiplexer to select the CPU interrupt sources and map them to the 12 maskable prioritized CPU interrupts.
- When CPU interrupt n occurs, program execution jumps to byte offset $4 \times 8 \times n = 32n$ in an *interrupt service table* (IST). The IST contains 16 *interrupt service fetch packets* (ISFP), each consisting of eight 32-bit instruction words. An ISFP may contain an entire interrupt service routine or may branch to a larger service routine.

TMS320C6701 Available Interrupt Sources

Sel Num	**Acronym**	**Description**
00000b	DSPINT	Host Processor to DSP interrupt
00001b	TINT0	Timer 0 interrupt
00010b	TINT1	Timer 1 interrupt
00011b	SD_INT	EMIF SDRAM timer interrupt
00100b	EXT_INT4	External interrupt pin 4
00101b	EXT_INT5	External interrupt pin 5
00110b	EXT_INT6	External interrupt pin 6
00111b	EXT_INT7	External interrupt pin 7
01xb	DMA_INTx	DMA channel x interrupt
01100b	XINT0	McBSP 0 transmit interrupt
01101b	RINT0	McBSP 0 receive interrupt
01110b	XINT1	McBSP 1 transmit interrupt
01111b	RINT1	McBSP 1 receive interrupt

Example of an Interrupt Service Fetch Packet

An ISFP for RESET for C programs is shown below.

```
mvkl _c_int00, b0; load lower 16 bits of _c_init
mvkh _c_int00, b0; load upper 16 bits of _c_init
b    b0          ; branch to C initialization
mvc  PCE1, b0    ; get base of IST
mvc  b0, ISTP    ; load pointer to IST base
nop  3           ; do 3 NOP's for branch latency 5
nop              ; add two words to fetch packet
nop              ;   to make a total of 8 words
```

We will normally start the interrupt service table (IST) at location 0. It can be relocated and the Interrupt Service Table Pointer register (ISTP) points to its starting address which must be a multiple of 256 words. The organization of the IST is shown in Slide 35.

Interrupt Service Table Structure

Priority	Byte Offset	ISFP	IER Bit
Highest	000h	$\overline{\text{RESET}}$	0
	020h	NMI	1
	040h	Reserved	2
	060h	Reserved	3
	080h	INT4	4
	0A0h	INT5	5
	0C0h	INT6	6
	0E0h	INT7	7
	100h	INT8	8
	120h	INT9	9
	140h	INT10	10
	160h	INT11	11
	180h	INT12	12
	1A0h	INT13	13
	1C0h	INT14	14
Lowest	1E0h	INT15	15

Interrupt Control Registers

	Name	**Description**
CSR	Control status reg	Globaly set or disable ints
IER	Int enable reg	Enable interrupts
IFR	Int flag reg	Shows status of interrupts
ISR	Int set reg	Manualy set flags in IFR
ICR	Int clear reg	Manualy clear flags in IFR
ISTP	Interrupt service table pointer	Pointer to the beginning of the interrupt service table
NRP	Nonmaskable int return pointer	Return address used on return from a nonmaskable interrupt
IRP	Interrupt return ptr	Return address used on return from a maskable interrupt

Conditions for an Interrupt

The following conditions must be met to process a maskable interrupt:

- The *global interrupt enable bit* (GIE) which is bit 0 in the control status register (CSR) is set to 1. When GIE = 0, no maskable interrupt can occur.
- The NMIE bit in the *interrupt enable register* (IER) is set to 1. No maskable interrupt can occur if NMIE = 0.
- The bit corresponding to the desired interrupt is set to 1 in the IER.
- The desired interrupt occurs, which sets the corresponding bit in the *interrupt flags register* (IFR) to 1 and no higher priority interrupt flags are 1 in the IFR

What Happens When an Interrupt Occurs

- The corresponding flag in the IFR is set to 1.
- If GIE = NMIE = 1 and no higher priority interrupts are pending, the interrupt is serviced:
  - GIE is copied to PGIE and GIE is cleared to preclude other interrupts.
  - The flag bit in the IFR is cleared.
  - The return address is put in the *interrupt return pointer* (IRP).
  - Execution jumps to the corresponding fetch packet in the interrupt service table (IST).
  - The service routine must save the CPU state on entry and restore it on exit.
  - A return from a maskable interrupt is accomplished by the assembly instructions

```
B   IRP;  return, moves PGIE to GIE
NOP 5  ;  delay slots for branch
```

The Interrupt Selector

- The 16 maskable interrupt sources are shown in Slide 33. The interrupt selector chooses and prioritizes which 12 the CPU will use.
- Any interrupt source can be mapped to any CPU interrupt by setting the INTSELn field of MUXL/MUXH to the desired selection number given in Slide 33

Interrupt Multiplexer Low Register 019C0004h

4–0	INTSEL4
9–5	INTSEL5
14–10	INTSEL6
15	Reserved
20–16	INTSEL7
25–21	INTSEL8
30–26	INTSEL9
31	Reserved

Interrupt Multiplexer High Register 019C0000h

4–0	INTSEL10
9–5	INTSEL11
14–10	INTSEL12
15	Reserved
20–16	INTSEL13
25–21	INTSE14
30–26	INTSEL15
31	Reserved

C Interrupt Service Routines
TI Extension to Standard C

- Declare the function to be an ISR by using the `interrupt` keyword:
  ```
  interrupt void your_isr_name(){}
  ```
  or use the interrupt pragma:
  ```
  #pragma INTERRUPT(your_isr_name)
  ```
- The C compiler will generate code to:
  1. Save the CPU registers used by the ISR on the stack. If the ISR calls another function, all registers are saved.
  2. Restore the registers before returning with a `B IRP` instruction.
- You cannot pass parameters to, or return values from an interrupt service routine.

Using the Peripheral Support Library

To write and build programs using the TI C interrupt extensions and the Peripheral Support Library:

- Include the following header files in your C program

  `C:\C6701\evm6x\dsp\include\intr.h`

  `C:\C6701\evm6x\dsp\include\regs.h`

- Link in the following libraries
  `C:\C6701\evm6x\dsp\lib\drivers\drv6x.lib`

  `C:\C6701\evm6x\dsp\lib\devlib\dev6x.lib`

- When using `dev6x.lib`, the interrupt service table is generated in a section called `.vec`. The sample beginning linker command file `evmlink.cmd` loads the `.vec` section starting at absolute address 0.

Selected Peripheral Support Library Interrupt Functions

`INTR_CHECK_FLAG(bit)`	Returns value of bit in IFR
`INTR_CLR_FLAG(bit)`	Clears int by writing 1 to ICR
`INTR_ENABLE(bit)`	Sets bit in IER
`INTR_DISABLE(bit)`	Clears bit in IER
`INTR_GLOBAL_ENABLE(bit)`	Sets GIE bit in CSR
`INTR_GLOBAL_DISABLE(bit)`	Clears GIE bit in CSR
`intr_hook(*fp,cpu_intr)`	Place func ptr into isr jump table at location for cpu intr
`intr_init()`	Init ISTP with addr of global label vec_table (resolved at link)
`intr_map(cpu_intr, isn)`	Maps int source to the CPU int
`intr_isn(cpu_intr)`	Returns int src num for CPU int

Installing a C ISR

- Initialize ISTP using label `vec_table` created by `dev6x.lib`
  ```
  intr_init();
  ```
- Map the interrupt source number to a CPU interrupt number.
  ```
  intr_map(CPU_INT15, ISN_XINT0);
  ```
- Clear the interrupt flag to make sure none is pending.
  ```
  INTR_CLR_FLAG(CPU_INT15);
  ```
- Hook the ISR to the CPU interrupt. Let the ISR be `my_isr()`.
  ```
  intr_hook(my_isr, CPU_INT15);
  ```
- Enable the NMI interrupt.
  ```
  INTR_ENABLE(CPU_INT_NMI);
  ```
- Enable the CPU interrupt in the IE register.
  ```
  INTR_ENABLE(CPU_INT15);
  ```
- Set the GIE bit in the CSR.
  ```
  INTR_GLOBAL_ENABLE();
  ```

Experiment 2 (Part 2)

Generating Sine Waves by Using Interrupts

Repeat the steps for Experiment 2 (Part 1) but now use a C interrupt service routine to generate the sine wave samples and write them to the McBSP0 data transmit register (DXR). No polling of the XRDY flag is needed because samples are transmitted only when interrupts occur at the codec's sampling rate.

The `main()` function should:

- initialize McBSP0
- initialize the codec, 8 kHz sampling rate
- map CPU INT15 to McBSP0 XINT0
  Note: The choice of INT15 was arbitrary. Any of INT4 – INT15 can be used.
- hook CPU INT15 to your ISR
- enable interrupts
- go into an infinte interruptable loop

Sample Program Segment for Interrupts

```
#include <stdlib.h>   /* Standard C library  */
#include <common.h>   /* macros for Ti lib's */
#include <board.h>    /* EVM drivers         */
#include <intr.h>     /* Interrupt functions */
#include <mcbspdrv.h> /* Serial port drivers */
#include <codec.h>    /* Codec drivers       */
#include <math.h>     /* C math functions    */
     .
     .
     .
  #define PI     3.141592653589
  #define FS     8000.
  #define LFREQ  1000.
  #define RFREQ  2000.
  float Ldelta = 2.*PI*LFREQ/FS;
  float Rdelta = 2.*PI*RFREQ/FS;

  void main(void){
/* Initialize EVM, McBSP0, and Codec first  */
      .
      .
      .
```

Sample Program for Ints (cont. 1)

```
/* Install interrupt service routine  */
   intr_init(); /* Disables all interrupts  */
  /* McBSP0 transmit int */
   intr_map(CPU_INT15, ISN_XINT0);
  /* Hook our ISR to INT15 */
   intr_hook(tx_isr, CPU_INT15);
  /* Clear old interrupts */
   INTR_CLR_FLAG(CPU_INT15);

/* Enable interrupts  */
  /* NMI must be enabled for  */
  /*    other ints to occur   */
   INTR_ENABLE(CPU_INT_NMI);
  /* Set INT15 bit in IER     */
   INTR_ENABLE(CPU_INT15);
  /* Turn on enabled ints     */
   INTR_GLOBAL_ENABLE();
/* Note: Functions in capital letters are  */
/* macros defined in the .h files          */
   MCBSP_WRITE(0,0);/*Write a word to start*/
                    /*  transmitter        */
   for(;;); /* Infinite interruptable loop */
}
```

Sample Program for Ints (cont. 2)

```
interrupt void tx_isr(void){
   int Lsample, Rsample;
   int LRsample;

/* WARNING:  Langle and Rangle must maintain
   their values between ISR calls.  This can
   be done by making them static as below or
   by making them global variables.    */

   static float Langle = 0. Rangle = 0.;

/* Generate left and right sine wave samples.
   Scale them up to use the D/A dynamic range
   and convert them to integers Lsample and
   Rsample.  Combine Lsample and Rsample into
   a single 32-bit int LRsample and send it
   to the codec.  Increment Langle and Rangle
   by Ldelta and Rdelta modulo 2 pi for the
   next samples and return to the infinite
   loop.                                    */
}
```

Chapter 2 (Part 3)
Direct Memory Access (DMA)

The Direct Memory Access (DMA) Controller is another important internal 'C6000 peripheral. The DMA controller transfers data between any locations in the DSP's 32-bit address space independently of the CPU. See *TMS320C6000 Peripherals Reference Guide*, SPRU190D, Chapter 4 for complete details.

Some features of the controller are:

- Four independent programmable channels
- A fifth (auxiliary) channel services requests from the host port interface (HPI)
- Can transfer 8-bit bytes, 16-bit halfwords, or 32-bit words
- Each block transfer can consist of multiple frames.

DMA Features (cont.)

- Split-channel mode where a single channel can perform both the receive and transmit element transfers to or from a peripheral simultaneously as if it were two DMA channels.
- After a transfer, addresses can stay the same, be incremented or decremented by one element, or incremented or decremented by the value in a global index register.
- Autoinitialization at the end of a block transfer
- Read, write, or frame transfers may be initiated (synchronized) by selected events like word receptions or transmissions by a McBSP.
- Can send an interrupt to the CPU at the end of a block transfer so the CPU can take some desired action.

DMA Control Registers

Mnemonic	**Name**
SRCx	DMA channel x source address
DSTx	DMA channel x destination address
XFRCNTx	DMA channel x transfer count
PRICTLx	DMA channel x primary control
SECCTLx	DMA channel x secondary control
AUXCTL	DMA auxiliary control register
GBLADDRy	DMA global address register y
GBLCNTz	DMA global count reload register z
GBLIDXz	DMA global index register z

$x \in \{0, 1, 2, 3\}$, $y \in \{\text{A, B, C, D}\}$, $z \in \{\text{A, B}\}$

Purpose of DMA Register

- **SRC*x***: Address for next read transfer
- **DST*x***: Address for next write transfer
- **PRICTL*x***: Used to control transfer
- **SECCTL*x***: Used to enable interrupts and monitor activity
- **AUXCTL**: Controls auxiliary channel
- **XFRCNT*x***: Number of frames to transfer & number of elements per frame
- **GBLADDR*y***: Peripheral address used in split transfer mode
- **GBLCNT*z***: Used to reload XFRCNT after last element transfer
- **GBLIDX*z***: Used to control address updates during transfer

DMA Channel Primary Control Register (PRICTL)

DST reload	SRC reload	Emod	FS	TCINT	PRI	WSYNC	RSYNC	INDEX	CNT reload	SPLIT	ESIZE	DST DIR	SRC DIR	Status	Start
31 30	29 28	27	26	25	24	23 19	18 14	13	12	11 10	9 8	7 6	5 4	3 2	1 0

DMA Channel Secondary Control Register (SECCTL)

Rsvd	DMAC	WSYNC CLR	WSYNC STAT	RSYNC CLR	RSYNC STAT	WDROP IE	WDROP COND	RDROP IE	RDROP COND	BLOCK IE	BLOCK COND	LAST IE	LAST COND	FRAME IE	FRAME COND	SX IE	SX COND
31 19	18 16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

DMA Channel Transfer Count Register (XFRCNT)

FRAME COUNT	ELEMENT COUNT
31 16	15 0

DMA Global Count Reload Register (GBLCNT)

FRAME COUNT RELOAD	ELEMENT COUNT RELOAD
31 16	15 0

DMA Global Index Register (GBLIDX)

FRAME INDEX	ELEMENT INDEX
31 16	15 0

DMA Global Address Register (GBLADDR)

SPLIT ADDRESS	Reserved
31 3	2 0

Major Steps for Setting Up a DMA Transfer

1. Halt the DMA by setting START = 00b in the Primary Control Register (PRICTL).
2. In the Secondary Control Register (SECCTL) set WSYNC CLR=1 and RSYNC CLR=1 to clear write sync status (WSYNC STAT) and read sync status (RSYNC STAT).
3. Load source and destination address registers and the SRC DIR and DST DIR fields in PRICTL to specify how address are modified after a transfer.
4. Load the element size field (ESIZE) in PRICTL and transfer count register (XFRCNT).
5. Set read (RSYNC) and write (WSYNC) synchronization sources in PRICTL.
6. Start the DMA transer by writing 01b for no autoinitialization or 11b for autoinitialization to the START field of the PRICTL register.

Address Control Fields in PRICTL

ESIZE	Element size
	0: 32-bit 01: 16-bit 01: 8-bit
DST/SRC DIR	Dest/Source address modification after transfer
	0: No modification 01: Increment by one element size 01: Decrement by one element size 11: Adjust using GBLIDX selected by INDEX
INDEX	Selects GBLIDX to use as programmable value
	0: Use GBLIDX register A 1: Use GBLIDX register B

START and STATUS Fields in PRICTL

STATUS (PRICTL[3:2])	
00	Stopped
01	Running without autoinitialization
10	Paused
11	Running with autoinitialization

START (PRICTL[1:0])	
00	Stop
01	Start without autoinitialization
10	Pause
11	Start with autoinitialization

Note: Once START is modified, PRICTL should not be modified again until STATUS = START.

DMA Autoinitialization

- Without autoinitialization, the DMA stops when the programmed number of elements is transferred and it generates a CPU interrupt if TCINT (PRICTL[25]) is set. Then an ISR can operate on the transferred block and set up the DMA for the next transfer.
- With autoinitialization, the DMA controller can automatically reinitialize itself when a block transfer is completed and begin a new transfer. The transfer count, source address, and destination address are reloaded from special registers which can be modified for the next transfer while the current one is in progress.
- Setting START = 11 in PRICTL starts the DMA in the autoinitialization mode.

DMA Reload Register Control

Successive transfers must be similar because reload values only available for registers modified during transfers (XFRCNT, SRC, DST).

Destination Address Reload Control
DST RELOAD (PRICTL[31:30])

00	Reload disabled
01	Use DMA GBLADDR reg B as reload
10	Use DMA GBLADDR reg C as reload
11	Use DMA GBLADDR reg D as reload

Source Address Reload Control
SRC RELOAD (PRICTL[29:28])

00	Reload disabled
01	Use DMA GBLADDR reg B as reload
10	Use DMA GBLADDR reg C as reload
11	Use DMA GBLADDR reg D as reload

Transfer Count Reload for Autoinitialization
CNT RELOAD (PRICTL[12])

0	Reload XFRCNT with GBLCNT A
1	Reload XFRCNT with GNLCNT B

Triggering DMA Transfers

- **Synchronization by Triggering Events**

No sync:	Reads and writes occur as fast as possible
Read sync:	Wait for event to occur before each read
Write sync:	Wait for event to occur before each write
Frame sync:	Wait for event to occur before each frame xfer

- Triggering events selected by RSYNC, WSYNC, and FS fields in PRICTL.

- DMA triggering events are similar to interrupts to the CPU.
- When FS = 1, the event selected in RSYNC enables an entire frame transfer.

Event Selection Bits in PRICTL

Num	**Acronym**	**Description**
00000b	None	No synchronization
00001b	TINT0	Timer 0 interrupt
00010b	TINT1	Timer 1 interrupt
00011b	SD_INT	EMIF SDRAM timer interrupt
0yb	EXT_INTy	External int pin y ($y =$ 4,5,6,7)
01xb	DMA_INTx	DMA channel x int ($x =$ 0,1,2,3)
01100b	XEVT0	McBSP 0 transmit event
01101b	REVT0	McBSP 0 receive event
01110b	XEVT1	McBSP 1 transmit event
01111b	REVT1	McBSP 1 receive event
10000b	DSPINT	Host Processor to DSP interrupt

Split Channel Operation

Split-channel operation allows a single DMA channel to service both the input (receive) and output (transmit) streams from a peripheral with a fixed address like a McBSP. This makes the DMA channel equivalent to two channels.

Split-channel operations consist of transmit element transfers and receive element transfers, each of which consists of a read and write transfer.

- **Transmit element transfer**
  - **Transmit read transfer:** Data is read from SRC address, SRC is adjusted as configured, and transfer count is decremented. This event is not synchronized.
  - **Transmit write transfer:** Data from read is written to split destination address. The write is synchronized according to WSYNC.

Split Channel Operation (cont.)

- **Receive element transfer**
  - **Receive read transfer:** Data is read from from the split source address. The read is synchronized according to the RSYNC field.
  - **Receive write transfer:** Data from the receive read is written to the destination address. The destination address is then adjusted as configured. This event is not synchronized.

In split mode, RSYNC and WSYNC must be nonzero. The element and frame count must be the same for the transmitted and received data. Also, frame synchronization must be disabled in split-channel operation.

Split Address Generation

The global address register (GBLADDR) selected by the SPLIT field in PRICTL determines the address of the peripheral for a split transfer:

- **Split source address:** The selected GBLADDR register contains the address for the input stream.
- **Split destination address:** The address for the output data stream is assumed to be one word address (four byte addresses) greater than the split source address. For example, for McBSP0:
  DRR byte address = 0x018C0000
  DXR byte address = 0x018C0004

SPLIT (PRICTL[11:10]) Field Values

00	Split mode disabled
01	Enabled; use GBLADDR reg A
10	Enabled; use GBLADDR reg B
11	Enabled; use GBLADDR reg C

Priorities

DMA vs. CPU Priority

Each DMA channel can be independently configured in high-priority mode by setting the PRI bit in the associated PRICTL register. Each resource uses its own scheme for resolving conflicts. Two examples are:

- If the CPU and DMA try to access the same internal data memory block at the same time, PRI determines the priority.
- The internal program memory always gives the CPU priority over the DMA.

Priority Between DMA Channels

The DMA controller gives Channel 0 the highest priority and Channel 3 the lowest priority. The auxiliary channel can be assigned a priority anywhere in this range.

Peripheral Support Library Functions for DMA

To use the Peripheral Support Library routines:

- Include the following header files in your C program
  `C:\C6701\evm6x\dsp\include\dma.h`
  `C:\C6701\evm6x\dsp\include\regs.h`
- Include the following libraries in your project:
  `C:\C6701\evm6x\dsp\lib\drivers\drv6x.lib`
  `C:\C6701\evm6x\dsp\lib\devlib\dev6x.lib`

If interested, you can find the source code in the `*.src` files. You can extract the individual modules by using the archiver `ar6x.exe`.

Some Peripheral Support Library DMA functions are listed in the next two slides. For more details see: *TMS320C6x Peripheral Support Library Programmer's Reference*, SPRU273B.

Peripheral Support Library DMA Routines

- `dma_global_init(auxcr, gcra, gcrb, gndxa, gndxb, gaddra, gaddrb, gaddrc, gaddrd)`: Set global dma registers
- `dma_init(ch, prictl, secctl, src_ad, dst_ad, tfrctr)` Set registers for the selected channel
- `dma_reset()`: Reset all DMA registers to default values
- `DMA_AUTO_START(ch)`: Begin autoinitialization operation on ch
- `DMA_START(ch)`: Begin DMA operation on selected channel
- `DMA_Stop(ch)`: Stop DMA operation on selected channel
- `DMA_Pause(ch)`: Pause DMA operation on selected channel
- `DMA_SRC_ADDR_ADDR(ch)`: Returns the address of the DMA source address register on the selected channel

Peripheral Support Library DMA Routines (cont.)

- `DMA_DEST_ADDR_ADDR(ch)`: Returns address of dest address reg
- `DMA_PRIMARY_CTRL_ADDR(ch)`: Returns address of PRICTL reg
- `DMA_SECONDARY_CTRL_ADDR(ch)`: Returns address of SECCTL
- `DMA_XFER_COUNTER_ADDR(ch)`: Returns address of XFRCNT
- `DMA_RSYNC_CLR(ch)`: Clear RSYNC in SECCTL so no sync
- `DMA_RSYNC_SET(ch)`: Set RSYNC in SECCTL selecting sync
- `DMA_WSYNC_CLR(ch)`: Clear WSYNC in SECCTL so no sync
- `DMA_WSYNC_SET(ch)`: Set WSYNC in SECCTL selecting sync

Experiment 2 (Part 3)
Generating a Sine Wave Using the DMA Controller

To learn how to use the DMA controller, do the following:

- Configure McBSP0 to transmit 32-bit words.
- Configure the codec for 16-bit linear, stereo mode, with an 8 kHz sampling rate.
- Generate a 512 word integer array, `table[512]`, where the upper 16 bits are the samples for 64 cycles of a 1 kHz sine wave for the left channel, and the lower 16 bits are the samples for 128 cycles of a 2 kHz sine wave for the right channel. Of course, the left and right channel sine wave samples must be scaled to use a large part of the DAC's dynamic range and must be converted to 16-bit integers before being combined into 32-bit words.

Experiment 2 (Part 3), DMA (cont.)

- Configure the DMA controller to read the entire array of 512 samples and write them to the Data Transmit Register (DXR) of McBSP0. Synchronize the transfers with the XRDY event to get the 8 kHz sampling rate. Use autoinitialization so the read address is automatically reset to the start of the sample array after the last word has been transmitted and the array is continually retransmitted.
- Observe the codec left and right channel outputs on the oscilloscope and verify that they are sine waves with the desired frequencies.

An example code segment is shown in the following slides to help you get started. The function `LOAD_FIELD(addr,val,bit,length)` used in the code is defined in `regs.h`.

Example DMA Code Segment

```
#define SZ_TABLE 512
int table[512];
void create_table(void);
void main(void){

/* DMA global and channel specific registers initialized  */
/*  to default values */
   unsigned int dma_gcr =0,   dma_gndxa=0,    dma_gaddra=0,
                dma_gcra=0,   dma_gndxb=0,    dma_gaddrb=0,
                dma_gcrb=0,   dam_gaddrc=0,   dma_gaddrd=0,
                dma_tcnt=0,   dma_pri_ctrl=0, dma_sec_ctrl=0,
                dma_src_addr, dma_dst_addr;

   MCBSP_ENABLE(0, MCBSP_TX ); /* enable mcbsp for Tx */
   create_table();    /* creates table of sine values */
   dma_reset();       /* reset DMA config registers   */
```

Example DMA Code Segment (cont. 1)

```
/* Set source addr, dest addr, and transfer count */
   dma_src_addr = (unsigned int) table;
   dma_dst_addr = MCBSP_DXR_ADDR(0);
   dma_tcnt     = SZ_TABLE;

/* Reload: no dest reload, src from GARB, tfr cnt from GCRA */
   LOAD_FIELD(&dma_pri_ctrl, DMA_RELOAD_GARB,
              SRC_RELOAD, SRC_RELOAD_SZ);
   LOAD_FIELD(&dma_pri_ctrl, DMA_RELOAD_NONE,
              DST_RELOAD, DST_RELOAD_SZ);
   dma_gaddrb = dma_src_addr;
   dma_gcra   = dma_tcnt;

/* Set sync info: No sync for reading sample and */
/* XEVT0 (mcbsp 0 Tx) as sync source for write   */
   LOAD_FIELD(&dma_pri_ctrl, SEN_NONE, RSYNC, RSYNC_SZ);
   LOAD_FIELD(&dma_pri_ctrl, SEN_XEVT0, WSYNC, WSYNC_SZ);
```

Example DMA Code Segment (c$^{\text{ont.}}$ 2)

```
 /* Addr adjust: src inc and dest unchanged after tfr */
   LOAD_FIELD(&dma_pri_ctrl, DMA_ADDR_INC, SRC_DIR,
              SRC_DIR_SZ);
   LOAD_FIELD(&dma_pri_ctrl, DMA_ADDR_NO_MOD, DST_DIR,
              DST_DIR_SZ);

 /* Load DMA channel registers */
   dma_global_init( dma_gcr, dma_gcra, dma_gcrb,
      dma_gndxa, dma_gndxb, dma_gaddra, dma_gaddrb,
      dam_gaddrc, dma_gaddrd);
   dma_init( DMA_CH1, dma_pri_ctrl, dma_sec_ctrl,
      dma_src_addr, dma_dst_addr, dma_tcnt);

 /* Start DMA channel 1in AUTO mode. */
  DMA_AUTO_START(DMA_CH1);
  while(1);
}
```