# F2833x Multichannel Buffered Serial Port

Introduction

The **M**ulti **C**hannel **B**uffered **S**erial **P**ort (McBSP) is a synchronous serial data communication channel for high-speed data transfer between the F2833x and external serial devices. It is quite often used to directly connect Audio - or Video - Codec's to an F2833x system. The 2833x device provides up to two high-speed multichannel buffered serial ports (McBSPs).

An independent clock signal (CLK(R/X) for receiver and transmitter can be generated by the F2833x (master - mode) or by the external device (slave - mode).
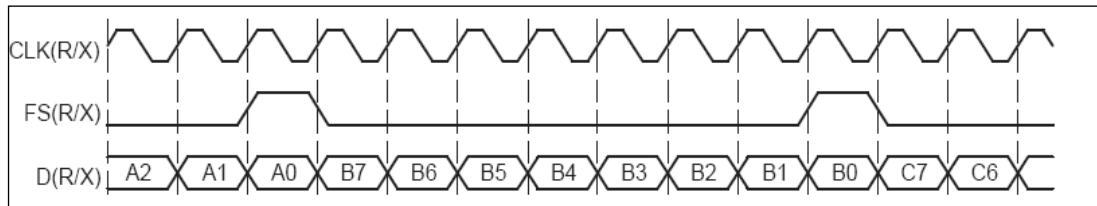

A frame sync signal (FS(R/X) indicates the beginning of a new data sequence (frame).

A frame contains between 1 and 128 words (or channels); a word is a number of bits (8, 12, 16, 20, 24 or 32).

The serial data streams are available on the "Data Transmit" (DX) and "Data Receive" (DR) pins.

A hardware compression and expanding technique by coding standards "A - law" (US and Japan) or "µ - law" (Europe) can be included into the transmission path. The specifications for µ-law and A-law log PCM are part of the CCITT G.711 recommendation.
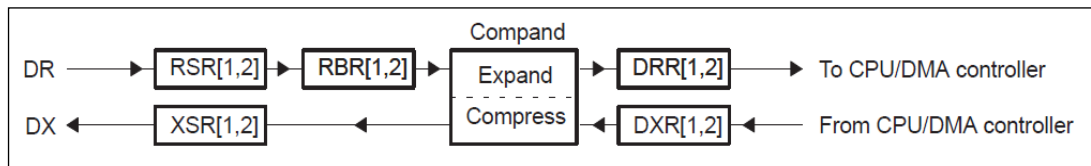
One of the ranges of operating modes of McBSP is a hardware emulation of an additional Serial Peripheral Interface (SPI).

The Peripheral Explorer Board provides interfaces to two McBSP related devices:

- A stereo audio codec TLV320AIC23B, connected to interface McBSP - A for the stereo in/out audio data stream and to interface SPI-A for the AIC23B control data stream.

- A serial EEPROM AT25C256K connected to interface McBSP - B.

Module Topics

F2833x McBSP Block diagram

Each McBSP - interface consists of six electrical signals:

Multichannel Buffered Serial Port (McBSP)

Introduction:

- **Two High – Speed multichannel synchronous serial ports (McBSP-A and McBSP-B)**

- **Maximum data rate: 20 MHz**

- **Each McBSP consists of a data - flow – path and a control - path**

- **Six Pins per channel**
    - **MDX: data transmit**
    - **MDR: data received**
    - **MCLKX: transmit clock**
    - **MCLKR: receive clock**
    - **MFSX: frame sync transmit**
    - **MFSR: frame sync receive**

13 - 2

The following slide explains the data flow:

McBSP Block Diagram
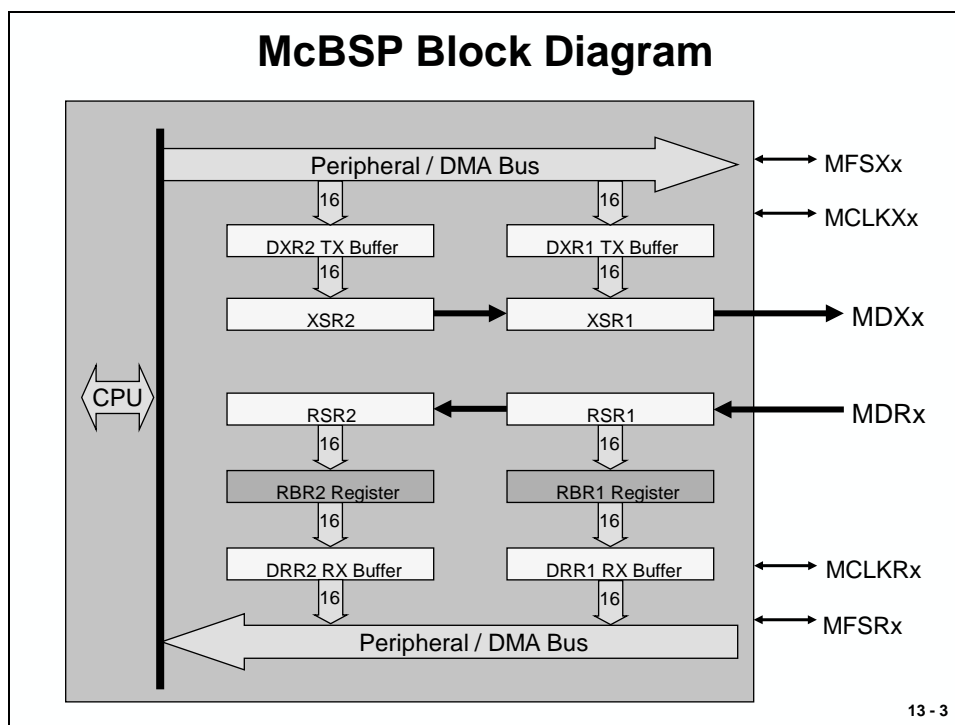
13 - 3

The electrical signals of a McBSP interface are listed in the following table. The direction of the clock signals depends on the setup of the McBSP as a master device (output) or as a slave device (input):

McBSP - A	McBSP - B	Signal Description
MCLKRA	MCLKRB	Receiver clock signal
MCLKXA	MCLKXB	Transmitter clock signal
MFSRA	MFSRB	Receiver Frame-Sync pulse signal
MFSXA	MFSXB	Transmitter Frame-Sync pulse signal
MDRA	MDRB	Data Input (Receiver)
MDXA	MDXB	Data Output (Transmitter)

The McBSP - interface is also connected to the CPU and to the DMA - unit of the F2833x via four internal event signals:
- MRINT: McBSP - Receive Interrupt signal
- MXINT: McBSP - Transmit Interrupt signal
- REVT: Receive Synchronization event to DMA - unit
- XEVT: Transmit Synchronization event to DMA - unit

Data values are communicated to devices interfaced to the McBSP via the data transmit (MDX) pin for transmission and via the data receive (MDR) pin for reception. Control information in the form of clocking and frame synchronization is communicated via the following pins: MCLKX (transmit clock), MCLKR (receive clock), MFSX (transmit frame synchronization), and MFSR (receive frame synchronization).

The CPU and the DMA controller communicate with the McBSP through 16-bit-wide registers, accessible via the internal peripheral bus. The CPU or the DMA controller writes the data to be transmitted to the data transmit registers (DXR1, DXR2). Data written to the DXRs is shifted out to DX via the transmit shift registers (XSR1, XSR2). Similarly, receive data on the DR pin is shifted into the receive shift registers (RSR1, RSR2) and copied into the receive buffer registers (RBR1, RBR2). The contents of the RBRs are then copied to the DRRs, which can be read by the CPU or the DMA controller. This allows simultaneous data movement of internal and external data communications.

DRR2, RBR2, RSR2, DXR2, and XSR2 are not used (written, read, or shifted) if the serial word length is 8 bits, 12 bits, or 16 bits. For larger word lengths, these registers are needed to hold the most significant bits.

Basic F2833x McBSP Features

Features of McBSP

- **Full - duplex communication**

- **Double-buffered transmission and triple-buffered reception, allowing a continuous data stream**

- **Independent clocking and framing for reception and transmission**

- **send interrupts to the CPU and send DMA events to the DMA - controller**

- **128 channels for transmission and reception**

- **Multichannel selection modes that enable or disable block transfers in each of the channels**

- **Direct interface to industry-standard CODECs, analogue interface chips (AICs), and other serially connected A/D and D/A devices**

- **Support for external generation of clock signals and frame - synchronization signals**

- **A programmable sample rate generator for internal generation and control of clock signals and frame - synchronization signals**

13 - 4

Features of McBSP

- **Direct interface to:**

  - **T1/E1 framers**

  - **IOM-2 compliant devices**

  - **AC97-compliant devices with multiphase frame capability**

  - **I2S compliant devices**

  - **SPI devices**

- **Variable data sizes: 8, 12, 16, 20, 24, and 32 bits**

- **A-law (Europe) and μ-law (US & Japan) hardware compression / expanding**

13 - 5

McBSP Data Frame Diagram

Since McBSP is a synchronous serial data communication interface, all bits are time synchronized by a clock signal ("CLK" in Slide 13-6). The receiver and transmitter part can use different timings (clock signals "MCLKX" and "MCLKR")

A McBSP data frame is started by a frame sync signal ("FS") shown in Slide 13-6. Again the transmitter and receiver can use independent frame sync signals (transmitter: "MFSX", receiver: "MFSR").
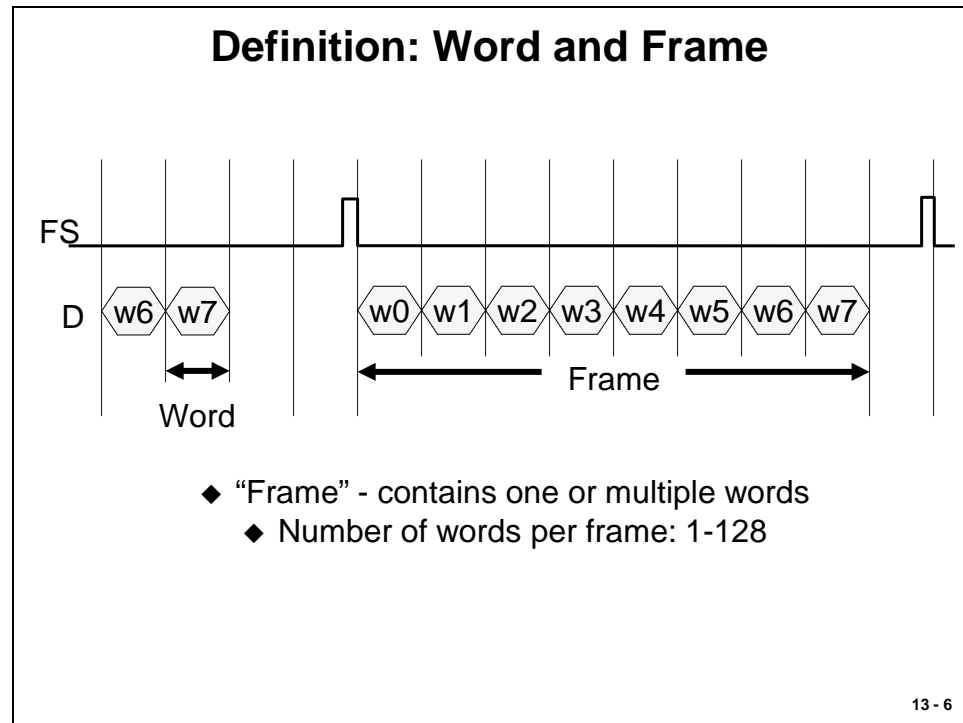
A frame consists of multiple "words". The number of words is programmable between 1 and 128 words.

A "word" is a certain number of bits, which can be programmed as:
- 8
- 12
- 16
- 20
- 24 or
- 32 bits per word.

In addition, there is the option to initialize the McBSP to a "single" phase mode or in a "dual" phase mode. In the latter we can use different setups for "words per frame" and "bits per word" in phase 1 and in phase 2. In Lab exercise 13_3 we will use a dual phase setup to send different stereo signals to right channel (phase 1) and left channel (phase 2) of the AIC23B audio codec.

Definition: Bit and Word

CLK

FS

D — a1, a0, b7, b6, b5, b4, b3, b2, b1, b0

Bit

Word

- ◆ "Bit" - one data bit per serial clock period
- ◆ "Word" or "channel" contains number of bits (8, 12, 16, 20, 24, 32)

13 - 7
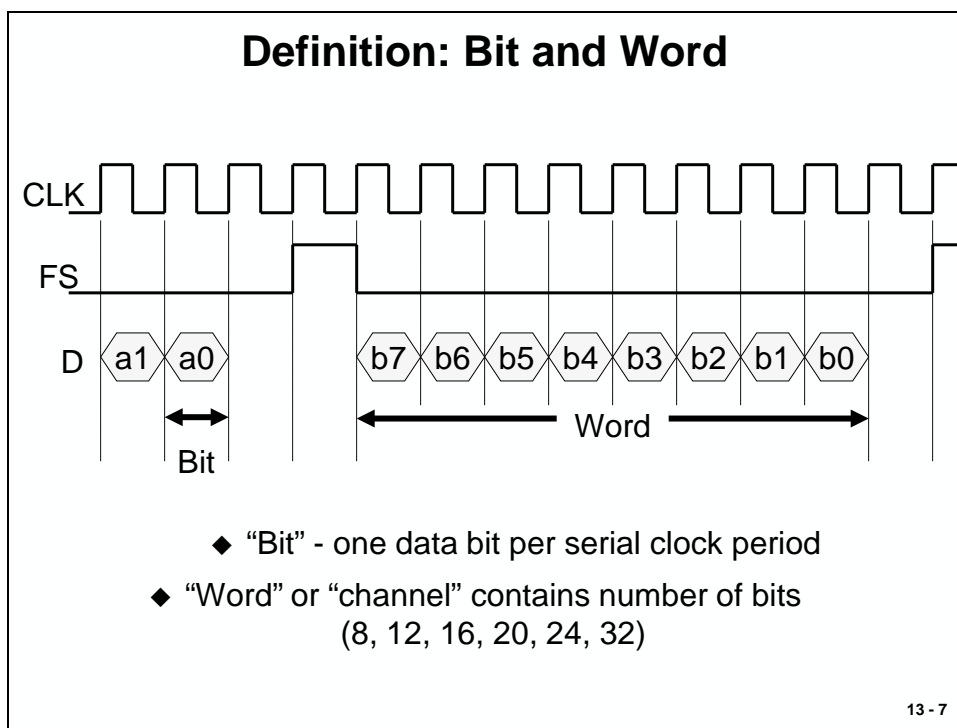
Another operating mode of the McBSP is called **"multi-channel" - mode**, in which we can time slice a frame and process only preselected words of this frame. The following slide is an example, in which a codec sends 32 words per frame, but the F2833x reads only words 0, 5 and 27 from each frame. Since we will not use this mode in our lab exercises, we will not go deeper into details of this operating mode for now.

Multi-Channel Selection

Frame   *TDM Bit Stream*   Multi-channel
Transmit & Receive only selected Channels

CODEC
0 — Ch31 ... Ch1 Ch0
1 — Ch31 ... Ch1 Ch0

McBSP

Ch0-0
Ch0-1

Ch5-0
Ch5-1

Ch27-0
Ch27-1

- ◆ Allows multiple channels (words) to be independently selected for transmit and receive (e.g. only enable Ch0, 5, 27 for receive, then process via CPU)

- ◆ The McBSP keeps time sync with all channels, but only "listens" or "talks" if the specific channel is enabled (reduces processing/bus overhead)

- ◆ Multi-channel mode controlled primarily via two registers:

Multi-channel Control Reg
MCR
(enables Mc-mode)

Rec/Xmt Channel Enable Regs
R/XCER (A-H)
(enable/disable channels)

- ◆ Up to 128 channels can be enabled/disabled

13 - 8

Companding (Compressing and Expanding)

Companding (COMpressing and exPANDing) hardware allows compression and expansion of data in either μ-law or A-law format. The companding standard employed in the United States and Japan is μ-law. The European companding standard is referred to as A-law. The specifications for μ-law and A-law log PCM are part of the CCITT G.711 recommendation.

The dynamic ranges of A-law and μ-law are 13 bits and 14 bits, respectively. Any values outside this range are set to the most positive or most negative value. Thus, for companding to work best, the data transferred to and from the McBSP via the CPU or DMA controller must be at least 16 bits wide. The μ-law and A-law formats both encode data into 8-bit code words. Companded data are always 8-bits wide; the appropriate word length bits (RWDLEN1, RWDLEN2, XWDLEN1 and XWDLEN2) must therefore be set to 0, indicating an 8-bit wide serial data stream.
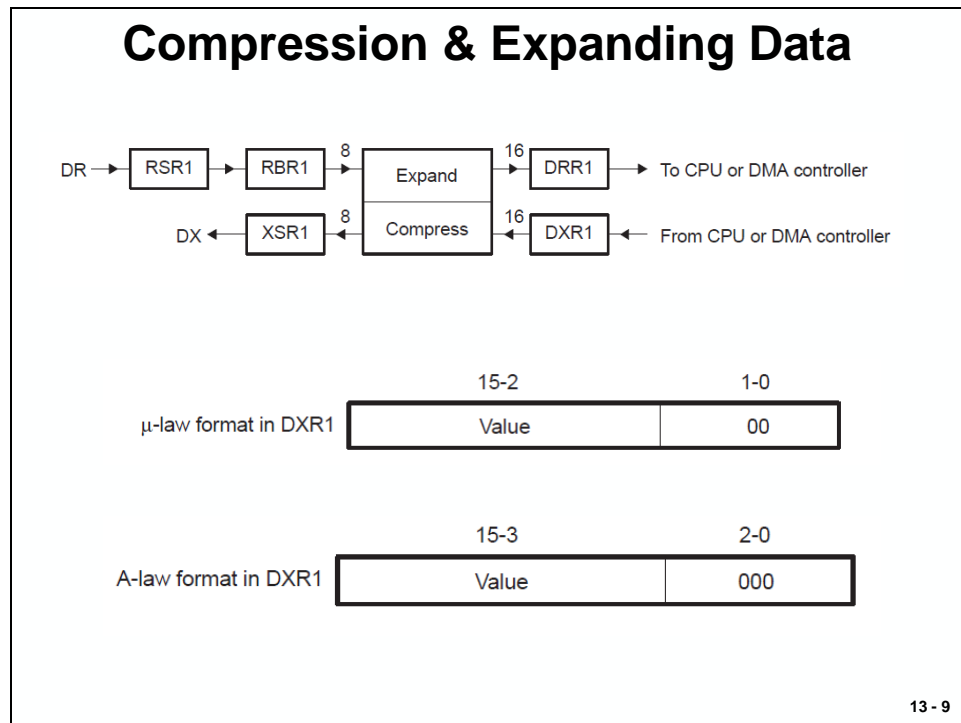
When companding is chosen for the transmitter, compression occurs during the process of copying data from DXR1 to XSR1. The transmit data is encoded according to the specified companding law (A-law or μ-law). When companding is chosen for the receiver, expansion occurs during the process of copying data from RBR1 to DRR1. The receive data values are decoded into twos-complement format.

McBSP - clocking

The figure shows "McBSP - Clocking" timing diagram with Internal CLK(R/X), Internal FS(R/X), and D(R/X) signals showing data bits A1, A0, B7, B6, B5, B4, B3, B2, B1, B0.

- Data is shifted one bit at a time
  - from the DR pin to the RSR(s) or
  - from the XSR(s) to the DX pin.

- The receive clock signal (CLKR) controls bit transfers from the DR pin to the RSR(s).

- The transmit clock signal (CLKX) controls bit transfers from the XSR(s) to the DX pin.

13 - 10

Data words are shifted one bit at a time from the DR pin to the RSR(s) or from the XSR(s) to the DX pin. The time for each bit transfer is controlled by the rising or falling edge of the receiver and transmitter clock signals.

The receive clock signal (CLKR) controls bit transfers from the DR pin to the RSR(s).

The transmit clock signal (CLKX) controls bit transfers from the XSR(s) to the DX pin.

CLKR or CLKX can be derived from a pin at the boundary of the McBSP (slave mode) or derived from inside the McBSP (master mode).

The polarities of CLKR and CLKX are programmable.

Please note: The McBSP cannot operate at a frequency faster than ½ the LSPCLK frequency. When driving CLKX or CLKR at the pin, choose an appropriate input clock frequency. When using the internal sample rate generator for CLKX and/or CLKR, choose an appropriate input clock frequency and divide down value (CLKDV) (i.e., be certain that CLKX or CLK do not exceed LSPCLK/2).

McBSP Frame Phases

The McBSP allows you to configure each frame to contain one or two phases. The number of words and the number of bits per word can be specified differently for each of the two phases of a frame, allowing greater flexibility in structuring data transfers. For example, you might define a frame as consisting of one phase containing two words of 16 bits each, followed by a second phase consisting of 10 words of 8 bits each. This configuration permits you to compose frames for custom applications or, in general, to maximize the efficiency of data transfers.

McBSP – Frame Phases

Single Phase Frame, 8 bits per word

Dual Phase Frame

Phase 1: 2 words of 12 bits each
Phase 2: 3 words of 8 bit each

13 - 11
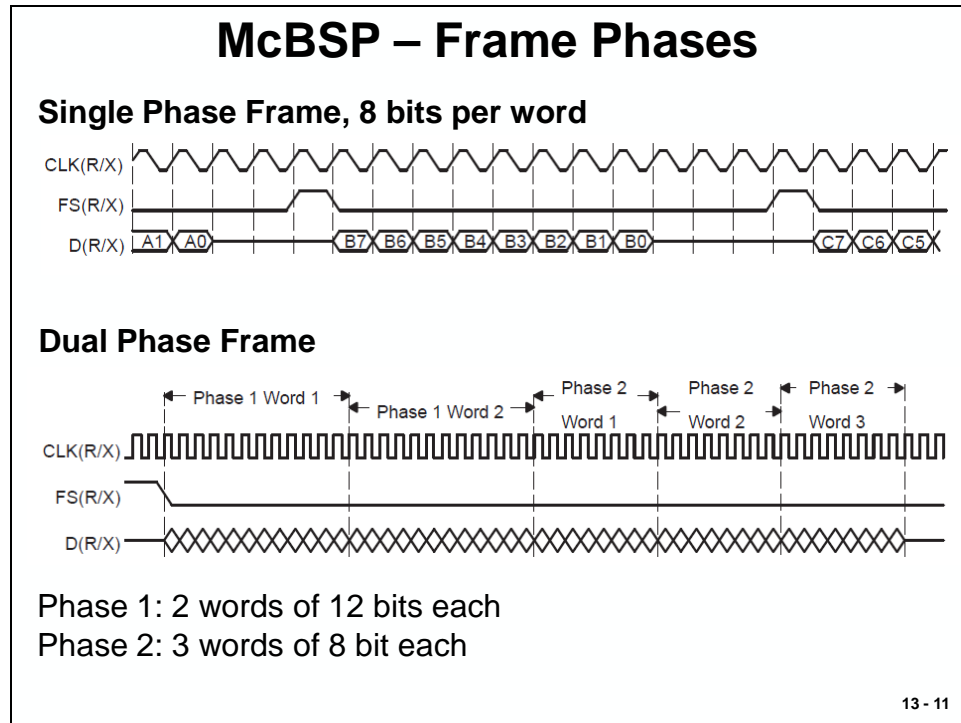
One clock pulse is generated by the master device for each data bit transferred. Due to a variety of different technology devices that can be connected to the I2C-bus, the levels of logic 0 (low) and logic 1 (high) are not fixed and depend on the associated level of V_{DD}. For details, see the data manual for your particular F2833x.

McBSP Receive

The following process describes how data travels from the DR pin to the CPU or to the DMA controller:

McBSP – Reception

Reception physical data path

DR → RSR[1,2] → RBR[1,2] → Expand or justify and bit fill → DRR[1,2] → To CPU or DMA controller

Reception signal activity

CLKR
FSR
DR   A1 A0   B7 B6 B5 B4 B3 B2 B1 B0   C7 C6 C5
RRDY

RBR1 to DRR1 copy(A)   Read from DRR1(A)   RBR1 to DRR1 copy(B)   Read from DRR1(b)

DRR = Data Receive Register
RRDY = status bit "Receiver Ready"

13 - 12
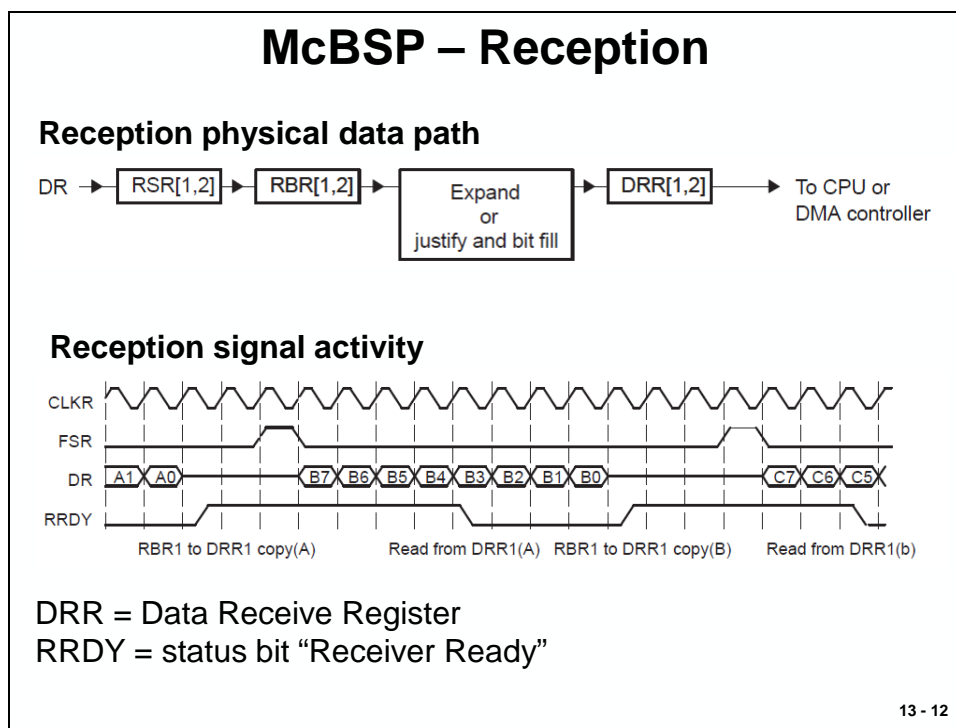
1. The McBSP waits for a receive frame-synchronization pulse on the internal FSR.
2. When the pulse arrives, the McBSP inserts the appropriate data delay that is selected with the RDATDLY bits of register RCR2 (with 1 data bit delay shown in Slide 13-12 above).
3. The McBSP accepts data bits on the DR pin and shifts them into the receive shift register(s). If the word length is 16 bits or smaller, only RSR1 is used. If the word length is larger than 16 bits, RSR2 and RSR1 are used; RSR2 contains the most significant bits.
4. When a full word is received, the McBSP copies the contents of the receive shift register(s) to the receive buffer register(s), provided that RBR1 is not full with previous data. If the word length is 16 bits or smaller, only RBR1 is used. If the word length is larger than 16 bits, RBR2 and RBR1 are used and RBR2 contains the most significant bits.
5. The McBSP copies the contents of the receive buffer register(s) into the data receive register(s), provided that DRR1 is not full with previous data. When DRR1 receives new data, the receiver ready bit (RRDY) is set in SPCR1. This indicates that received data is ready to be read by the CPU or the DMA controller. If the word length is 16 bits or smaller, only DRR1 is used. If the word length is larger than 16 bits, DRR2 and DRR1 are used and DRR2 contains the most significant bits. If companding is used during the copy (RCOMPAND = 10b or 11b in RCR2), the 8-bit compressed data value in RBR1 is expanded to a left-justified 16-bit value in DRR1. If companding is disabled, the data value copied from RBR[1,2] to DRR[1,2] is justified and bit filled according to the RJUST bits.
6. The CPU or the DMA controller reads the data from the data receive register(s). When DRR1 is read, RRDY is cleared and the next RBR-to-DRR copy is initiated.

McBSP Transmission

This section explains the fundamental process of transmission in the McBSP.

McBSP – Transmission

Transmission physical data path

DX ← XSR[1,2] ← Compress or do not modify ← DXR[1,2] ← From CPU or DMA controller

Transmission signal activity

CLKX

FSX

DX   A1 A0   B7 B6 B5 B4 B3 B2 B1 B0   C7 C6 C5

XRDY

DXR1 to XSR1 copy(B)   Write to DXR1(C)   DXR1 to XSR1 copy(C)   Write to DXR1

DXR = Data Transmit Register
XRDY = status bit "Transmitter Ready"

13 - 13
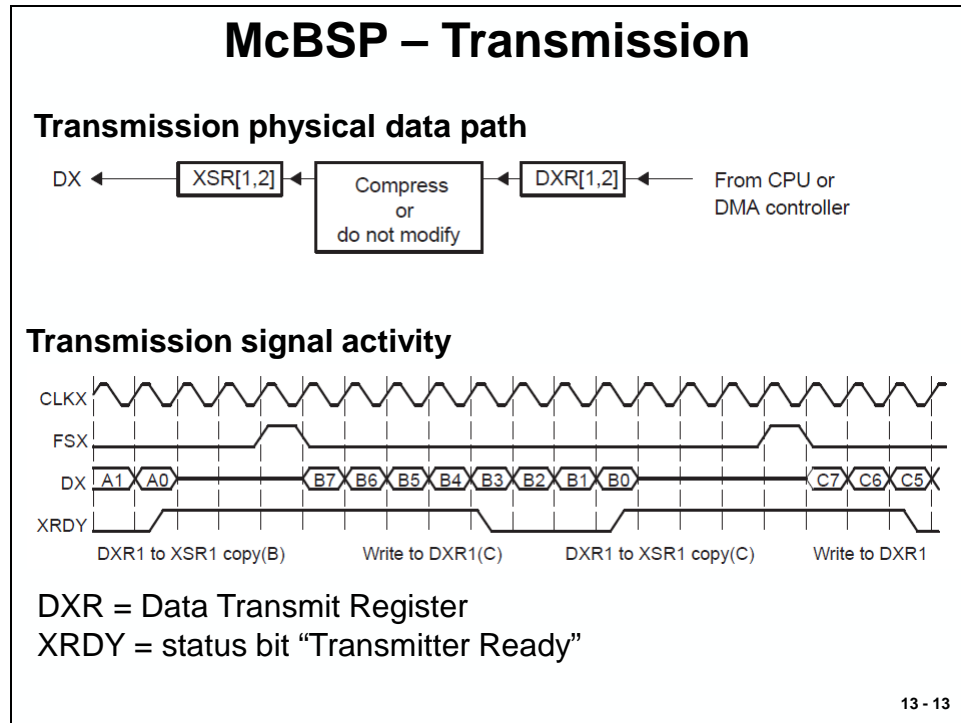
1. The CPU or the DMA controller writes data to the data transmit register(s). When DXR1 is loaded, the transmitter ready bit (XRDY) is cleared in SPCR2 to indicate that the transmitter is not ready for new data. If the word length is 16 bits or smaller, only DXR1 is used. If the word length is larger than 16 bits, DXR2 and DXR1 are used and DXR2 contains the most significant bits.

2. When new data arrives in DXR1, the McBSP copies the contents of the data transmit register(s) to the transmit shift register(s). In addition, the transmit ready bit (XRDY) is set. This indicates that the transmitter is ready to accept new data from the CPU or the DMA controller. If the word length is 16 bits or smaller, only XSR1 is used. If the word length is larger than 16 bits, XSR2 and XSR1 are used and XSR2 contains the most significant bits. If companding is used during the transfer (XCOMPAND = 10b or 11b in XCR2), the McBSP compresses the 16-bit data in DXR1 to 8-bit data in the μ-law or A-law format in XSR1. If companding is disabled, the McBSP passes data from the DXR(s) to the XSR(s) without modification.

3. The McBSP waits for a transmit frame-synchronization pulse on the internal FSX.

4. When the pulse arrives, the McBSP inserts the appropriate data delay that is selected with the XDATDLY bits of XCR2. In the preceding timing diagram, a 1-bit data delay is selected.

5. The McBSP shifts data bits from the transmit shift register(s) to the DX pin.

McBSP - Interrupts and DMA

The McBSP sends notification of important events to the CPU and DMA via the internal signals shown in Slide 13-14:

McBSP – Interrupts and DMA

Internal Signals from McBSP to CPU or DMA:

Internal Signal	Description
RINT	Receiver Interrupt from McBSP to CPU; based on a selected condition in the receiver
XINT	Transmitter Interrupt from McBSP to CPU; based on a selected condition in the transmitter
REVT	Receive synchronization event from McBSP to DMA; triggered when data has been received in DRR.
XEVT	Transmit synchronization event from McBSP to DMA; triggered when DXR is ready to accept new data.

13 - 14

McBSP Module Registers

A register summary of the set of McBSP - Registers is shown at the following two slides:

McBSP – Register Set

McBSP Control & Data Registers:

Register	Description
DRR2	Data Receive Register 2 (high)
DRR1	Data Receive Register 1 (low)
DXR2	Data Transmit Register 2 (high)
DXR1	Data Receive Register 1 (low)
SPCR2	Serial Port Control Register 2
SPCR1	Serial Port Control Register 1
RCR2	Receive Control Register 2
RCR1	Receive Control Register 1
XCR2	Transmit Control Register 2
XCR1	Transmit Control Register 1
SRGR2	Sample Rate Generator Register 2
SRGR1	Sample Rate Generator Register 1

13 - 15

McBSP – Register Set

McBSP Multi Channel Control Registers:

Register	Description
MCR2	Multichannel Control Register 2
MCR1	Multichannel Control Register 1
RCERx	Receive Channel Enable Register Partition x
XCERx	Transmit Channel Enable Register Partition x
PCR	Pin Control Register
XCERB	Transmit Channel Enable Register Partition B
PCR	Pin Control Register
MFFINT	Interrupt Enable Register

x = Partition A, B, C, D, E, F, G, H

Partition	Channels
A	0 - 1 5
B	16 – 31
C	32 – 47
D	48 – 63
E	64 – 79
F	80 – 95
G	96 – 111
H	112 - 127

13 - 16

Data Receive and Transmit Register

The CPU or the DMA controller reads received data from one or both of the data receive registers. If the serial word length is 16 bits or smaller, only DRR1 is used. If the serial length is larger than 16 bits, both DRR1 and DRR2 are used and DRR2 holds the most significant bits.

McBSP – Data Register

McBSP Data Receive Register (DRR2 and DRR1):

DDR2

15		0
	High part of receive data (for 20-, 24- or 32-bit data)	
	R/W-0	

DDR1

15		0
	Receive data (for 8-, 12-, or 16-bit data) or low part of receive data (for 20-, 24- or 32-bit data)	
	R/W-0	

McBSP Data Transmit Register (DXR2 and DXR1):

DXR2

15		0
	High part of transmit data (for 20-, 24- or 32-bit data)	
	R/W-0	

DXR1

15		0
	Transmit data (for 8-, 12-, or 16-bit data) or low part of receive data (for 20-, 24- or 32-bit data)	
	R/W-0	

13 - 17

If the serial word length is 16 bits or smaller, receive data on the MDRx pin is shifted into receive shift register 1 (RSR1) and then copied into receive buffer register 1 (RBR1). The content of RBR1 is then copied to DRR1, which can be read by the CPU or by the DMA controller. The RSRs and RBRs are not accessible by the user.

If the serial word length is larger than 16 bits, receive data bits on the MDRx pin are shifted into both of the receive shift registers (RSR2, RSR1) and then copied into both of the receive buffer registers (RBR2, RBR1). The contents of the RBRs are then copied into both of the DRRs, which can be read by the CPU or by the DMA controller.

If companding is used during the copy from RBR1 to DRR1 (RCOMPAND = 10b or 11b), the 8-bit compressed data value in RBR1 is expanded to a left-justified 16-bit value in DRR1. If companding is disabled, the data copied from RBR[1,2] to DRR[1,2] is justified and bit filled according to the RJUST bits.

If the serial word length is 16 bits or fewer, data written to DXR1 is copied to transmit shift register 1 (XSR1). From XSR1, the data is shifted onto the DX pin one bit at a time. If the serial word length is more than 16 bits, data written to DXR1 and DXR2 is copied to both transmit shift registers (XSR2, XSR1). From the XSRs, the data is shifted onto the DX pin one bit at a time. If companding is used during the transfer from DXR1 to XSR1 (XCOMPAND = 10b or 11b), the McBSP compresses the 16-bit data in DXR1 to 8-bit data in the μ-law or A-law format in XSR1. If companding is disabled, the McBSP passes data from the DXR(s) to the XSR(s) without modification.

Serial Port Control Register 1 (SPCR1)

McBSP – Serial Port Control Register (SPCR1)

15	14	13	12	11	10		8
DLB	RJUST		CLKSTP		Reserved		
R/W-0	R/W-0		R/W-0		R-0		

7	6	5	4	3	2	1	0
DXENA	Reserved	RINTM		RSYNCERR	RFULL	RRDY	RRST
R/W-0	R/W-0	R/W-0		R/W-0	R-0	R-0	R/W-0

DLB: Digital Loopback Mode   0 = disabled; 1 = enabled
RJUST: Receive Justification Mode 0 = Right justify data and zero fill MSBs
                                      1 = Right justify data and sign – extend MSBs
                                      2 = Left justify data and zero fill LSBs
CLKSTP: Clock Stop Mode   0 and 1 = disabled
                                      2 = enabled, without clock delay (SPI – Mode)
                                      3 = enabled, with half-cycle clock delay (SPI – Mode)
DXENA: DX Delay Enable   0 = OFF
                                      1 = ON; extra delay for turn – ON - time
RINTM: Receiver Interrupt Mode   0 = INT when RRDY = 1
                                      1 = INT after 16 channels (multichannel mode)
                                      2 = INT of frame sync pulse
                                      3 = INT on Receive Frame Sync Error
RSYNCERR: Receive Frame Sync Error 0 = no error; 1 = error
RFULL: Receiver Full Status bit   1 = Receiver Full condition (RSR, RBR and DRR full)
RRDY: Receiver Ready Status bit   1 = Receiver Ready; new data in DRR
RRST: Receiver Reset Control Bit 0 = Reset Receiver; 1 = release Receiver from Reset

13 - 18

Serial Port Control Register 2 (SPCR2)

McBSP – Serial Port Control Register (SPCR2)

15					10	9	8
Reserved						FREE	SOFT
R-0						R/W-0	R/W-0

7	6	5	4	3	2	1	0
FRST	GRST	XINTM		XSYNCERR	XEMPTY	XRDY	XRST
R/W-0	R/W-0	R/W-0		R/W-0	R-0	R-0	R/W-0

FREE: Free Run JTAG Mode   0 = Stop at breakpoint; 1 = Free Run
Soft: Soft Stop JTAG Mode   0 = if FREE = 0, stop immediately in case of breakpoint
                                  1 = if FREE = 0, stop at end of frame

FRST: Frame Sync Logic Reset   0 = Reset; 1 = release Frame Logic from Reset
GRST: Sample Rate Generator Reset 0 = Reset; 1 = release SRG from Reset
XINTM: Transmit Interrupt Mode   0 = INT when XRDY = 1
                                  1 = INT after 16 channels (multichannel mode)
                                  2 = INT of frame sync pulse
                                  3 = INT on Transmit Frame Sync Error
XSYNCERR: Receive Frame Sync Error 0 = no error; 1 = error
XEMPTY: Transmitter Empty Status bit 0 = Transmitter empty (DXR1); 1 = not empty
XRDY: Transmitter Ready Status bit 1 = Transmitter ready (DXR1,2) to accept new data
XRST: Transmitter Reset Control Bit 0=Reset Transmitter; 1=release Transmitter from Reset

13 - 19

Receive Control Register 1 (RCR1)

McBSP – Receive Control Register 1 (RCR1)

15	14		8
Reserved		RFRLEN1	
R-0		R/W-0	

7	5	4	0
RWDLEN1		Reserved	
R/W-0		R-0	

RFRLEN1:   Receive Frame Length1(0…0x7F)
           Single Phase Frame: Number of Words in a frame (1…128)
           Dual Phase Frame: Number of Words in frame – phase 1 (1...128)

RWDLEN1:   Receive Word Length1 (0…7)
           Single Phase Frame: Number of Bits in a Word
           Dual Phase Frame: Number of Bits in a Word of frame - phase 1
           0 = 8 Bit
           1 = 12 Bit
           2 = 16 Bit
           3 = 20 Bit
           4 = 24 Bit
           5 = 32 Bit
           6 and 7: reserved

13 - 20

Receive Control Register 2 (RCR2)

McBSP – Receive Control Register 2 (RCR2)

15	14		8
RPHASE		RFRLEN2	
R/W-0		R/W-0	

7	5	4	3	2	1	0
RWDLEN2		RCOMPAND		RFIG	RDATDLY	
R/W-0		R/W-0		R/W-0	R/W-0	

RPHASE: Receive Phase Number   0 = Single Phase Frame
                                       1 = Dual Phase Frame

RFRLEN2:   Receive Frame Length 2 (0…0x7F)
           Single Phase Frame: don't care
           Dual Phase Frame: Number of Words in frame – phase 2 (1...128)

RWDLEN2:   Receive Word Length 2 (0…7)
           Single Phase Frame: don't care
           Dual Phase Frame: Number of Bits in a Word of frame - phase 2
           0 = 8 Bit, 1 = 12 Bit, 2 = 16 Bit, 3 = 20 Bit, 4 = 24 Bit, 5 = 32 Bit
RCOMPAND: Receive Companding Mode   0 = no companding, MSB received first
                                       1 = no companding, 8-bit-data, LSB first
                                       2 = µ-law; 8-bit-data, MSB received first
                                       3 = A-law; 8-bit-data, MSB received first
RFIG: Receive Frame Sync Ignore 1 = unexpected Frame Sync ignored
RDATDLY: Receive Data Delay   0 = 0 clock cycles delay after frame Sync
                                       1 = 1 cycle; 2 = 2 cycles; 3 =reserved

13 - 21

Transmit Control Register 1 (XCR1)

McBSP – Transmit Control Register 1 (XCR1)

15	14		8
Reserved		XFRLEN1	
R-0		R/W-0	

7	5	4	0
XWDLEN1		Reserved	
R/W-0		R-0	
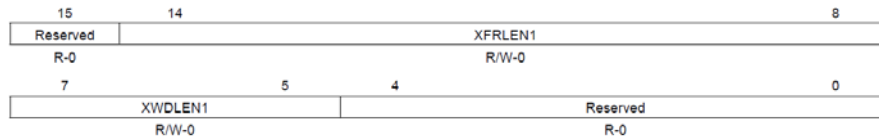
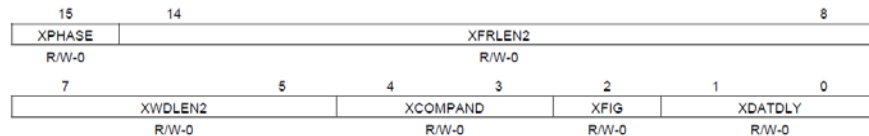XFRLEN1:   Transmit Frame Length1(0…0x7F)
                Single Phase Frame: Number of Words in a frame (1…128)
                Dual Phase Frame: Number of Words in frame – phase 1 (1...128)

XWDLEN1:   Transmit Word Length1 (0…7)
                Single Phase Frame: Number of Bits in a Word
                Dual Phase Frame: Number of Bits in a Word of frame - phase 1
                0 = 8 Bit
                1 = 12 Bit
                2 = 16 Bit
                3 = 20 Bit
                4 = 24 Bit
                5 = 32 Bit
                6 and 7: reserved

13 - 22

Transmit Control Register 2 (XCR2)

McBSP – Transmit Control Register 2 (XCR2)

15	14		8
XPHASE		XFRLEN2	
R/W-0		R/W-0	

7	5	4	3	2	1	0
XWDLEN2		XCOMPAND		XFIG	XDATDLY	
R/W-0		R/W-0		R/W-0	R/W-0	

XPHASE: Transmit Phase Number   0 = Single Phase Frame
                                            1 = Dual Phase Frame

XFRLEN2:   Transmit Frame Length 2 (0…0x7F)
               Single Phase Frame: don't care
               Dual Phase Frame: Number of Words in frame – phase 2 (1...128)

XWDLEN2:   Transmit Word Length 2 (0…7)
               Single Phase Frame: don't care
               Dual Phase Frame: Number of Bits in a Word of frame - phase 2
               0 = 8 Bit, 1 = 12 Bit, 2 = 16 Bit, 3 = 20 Bit, 4 = 24 Bit, 5 = 32 Bit

XCOMPAND: Transmit Companding Mode   0 = no companding, MSB transmitted first
                                           1 = no companding, 8-bit-data, LSB first
                                           2 = µ-law; 8-bit-data, MSB transmitted first
                                           3 = A-law; 8-bit-data, MSB transmitted first

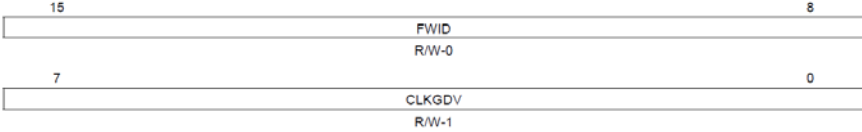XFIG: Transmit Frame Sync Ignore   1 = unexpected Frame Sync ignored
XDATDLY: Transmit Data Delay   0 = 0 clock cycles delay after frame Sync
                                           1 = 1 cycle; 2 = 2 cycles; 3 =reserved **13 - 23**

Sample Rate Generator Register 1 (SRGR1)

McBSP – Sample Rate Generator (SRGR1)

15					8
		FWID			
		R/W-0			

7					0
		CLKGDV			
		R/W-1			

FWID: Frame Sync Pulse Width   0…255
Pulse Width of Frame Sync Signal in McBSP clock cycles

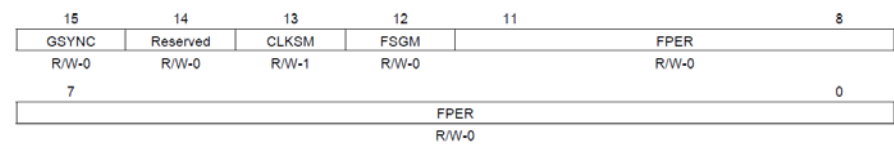CLKGDV: Divide Down Value for Clock-Generator   0…255

CLKG frequency = (Input clock frequency)/ (CLKGDV + 1)
The input clock is selected by the SCLKME (Register PCR) and CLKSM (Register SRGR) bits:

SCLKME	CLKSM	Input Clock Source
0	0	Reserved
0	1	LSPCLK
1	0	Signal on pin MCLKR
1	1	Signal on pin MCLKX

13 - 24

Sample Rate Generator Register 2 (SRGR2)

McBSP – Sample Rate Generator (SRGR2)

15	14	13	12	11		8
GSYNC	Reserved	CLKSM	FSGM	FPER		
R/W-0	R/W-0	R/W-1	R/W-0	R/W-0		

7						0
			FPER			
			R/W-0			

GSYNC: Clock Sync Mode   only used, if clock source is external
           1 = Clock Synchronization;
                CLKG is adjusted to MCLKR / MCLKX
           0 = no clock sync;
                CLKG free running, FSG every FPER-cycles
CLKSM: Sample Clock Mode

SCLKME	CLKSM	Input Clock Source
0	0	Reserved
0	1	LSPCLK
1	0	Signal on pin MCLKR
1	1	Signal on pin MCLKX

FSGM: Frame Sync Mode;   Frame Pulse from pin FSX (if FSXM = 0)
0:          if FSXM = 1, generate frame pulse when DXR is copied into XSR
1:          if FSXM = 1, generate frame pulse based on FPER and FWID
FPER: Frame Sync Period   (1…4096); Number of CLKG cycles between frame pulses

13 - 25

Pin Control Register (PCR)

McBSP – Pin Control Register (PCR)

15			12	11	10	9	8
Reserved				FSXM	FSRM	CLKXM	CLKRM
R-0				R/W-0	R/W-0	R/W-0	R/W-0

7	6	5	4	3	2	1	0
SCLKME	Reserved	DXSTAT	DRSTAT	FSXP	FSRP	CLKXP	CLKRP
R/W-0	R-0	R/W-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0

FSXM: Transmit Frame Sync Mode 0 = Frame Pulse is supplied externally via pin FSX
1 = Frame Pulse generated internally by Sample Rate Generator (bit FSGM of register *SRGR2)*

FSRM: Receive Frame Sync Mode 0 = Frame Pulse is supplied externally via pin FSR
1 = Frame Pulse generated internally by Sample Rate Generator (bit FSGM of register *SRGR2)*

CLKXM: Transmit Clock Mode if CLKSTP = 0 or 1:
0 = external transmit clock from pin MCLKX
1 = internal transmit clock; MCLKX is output
if CLKSTP = 2 or 3:
0 = McBSP is slave in SPI – Protocol; MCLKX is input
1 = McBSP is master in SPI – Mode; MCLKX is output

SCLKME: Sample Rate Generator Input Mode (see CLKSM in Register SRGR2)
DXSTAT: DX pin Status Bit 1 = drive DX pin high; 0 = DX pin low (GPIO mode)
DRSTAT: DR pin Status Bit 1 = drive DR pin high; 0 = DR pin low (GPIO mode)
FSXP: Transmit Frame Sync Polarity 0 = active high; 1 = active low
FSRP: Receive Frame Sync Polarity 0 = active high; 1 = active low
CLKXP: Transmit Clock Polarity data valid on rising (0) or falling (1) edge of CLKX
CLKRP: Receive Clock Polarity data sampled on rising (1) or falling (0) edge

13 - 26

Interrupt Enable Register (MFFINT)

McBSP – Interrupt Enable Register (MFFINT)

15		8
Reserved		
R-0		

7		3	2	1	0
Reserved			RINT ENA	Reserved	XINT ENA
R-0			R/W-0	R-0	R/W-0

RINT ENA: 0 = disable McBSP Receive Interrupts
1 = enable McBSP Receive Interrupts

RRDY   00
EOBR condition 01   RINT
FSR detected 10   RINTENA   MRINT
RSYNCERR   11

RINTM bits

XINT ENA: 0 = disable McBSP Transmit Interrupts
1 = enable McBSP Transmit Interrupts

XRDY   00
EOBX condition 01   XINT
FSX detected 10   XINTENA   MXINT
XSYNCERR   11

XINTM bits

13 - 27

Multichannel Mode Enable Registers (RCERx, XCERx)

We will not use the multichannel operation mode in our next lab exercises. Therefore we will not discuss these registers here. For more information about these registers and the multi-channel operation mode please refer to user manual "TMS320F2833x Multichannel Buffered Serial Port (McBSP) Reference Guide", literature number: SPRUFB7A at www.ti.com.

In the next pages we will discuss the first external device of the Peripheral Explorer Board, which is connected via McBSP - A:

Stereo Audio Codec TLV320AIC23B

Stereo Audio Codec TLV320AIC23B

Here is what the data sheet says:

"The TLV320AIC23B is a high-performance stereo audio codec with highly integrated analogue functionality. The analogue-to-digital converters (ADCs) and digital-to-analogue converters (DACs) within the TLV320AIC23B use multi-bit sigma-delta technology with integrated oversampling digital interpolation filters. Data-transfer word lengths of 16, 20, 24, and 32 bits, with sample rates from 8 kHz to 96 kHz, are supported. The ADC sigma-delta modulator features third-order multi-bit architecture with up to 90-dBA signal-to-noise ratio (SNR) at audio sampling rates up to 96 kHz, enabling high-fidelity audio recording in a compact, power-saving design. The DAC sigma-delta modulator features a second-order multi-bit architecture with up to 100-dBA SNR at audio sampling rates up to 96 kHz, enabling high-quality digital audio-playback capability, while consuming less than 23 milliwatts during playback only. The TLV320AIC23B is the ideal analogue input/output (I/O) choice for portable digital audio-player and recorder applications, such as MP3 digital audio players."

Wow, sounds impressive, doesn't it? But the question is: how can we get this device to work? And, more importantly, if it is running, can we explain why? So let us try to use it in a simple application first. At the end of this chapter we will use the internal DAC to "synthesize" an audio stereo output signal, based on the TMS320F2833x internal sine wave lookup table. With a headphone plugged into J25 of the Peripheral Explorer Board we can make this signal audible (or we use a scope to measure it).

To start with, let us start with a brief discussion of the functionality of this device. Here are the main features:

Stereo Audio Codec TLV320AIC23B

Main Features:

- 90dB SNR Multibit Sigma-Delta ADC
- 100-dB SNR Multibit Sigma-Delta DAC
- 8 kHz …96 kHz Sampling-Frequency
- SPI- Interface for Control Channel Compatible Serial-Port Protocols
- 2 - Phase Audio - Data Input/Output via McBSP
- Standard I2S, MSB, or LSB Justified-Data Transfers
- 16/20/24/32-Bit Audio Data Word Length
- Volume Control With Mute on Input and Output
- ADC Multiplexed Input for Stereo-Line Inputs and Microphone
- Highly Efficient Linear Headphone Amplifier (30 mW into 32 Ohm from a 3.3-V Analogue Supply Voltage

13 - 28

Functional Block Diagram

Stereo Audio Codec TLV320AIC23B

Functional Block
Diagram:

Signals:
RHPOUT =
    right headphone out
LHPOUT =
    left headphone out

© Document Number : SLWS106H,
    page 1-3 (www.ti.com)

13 - 29
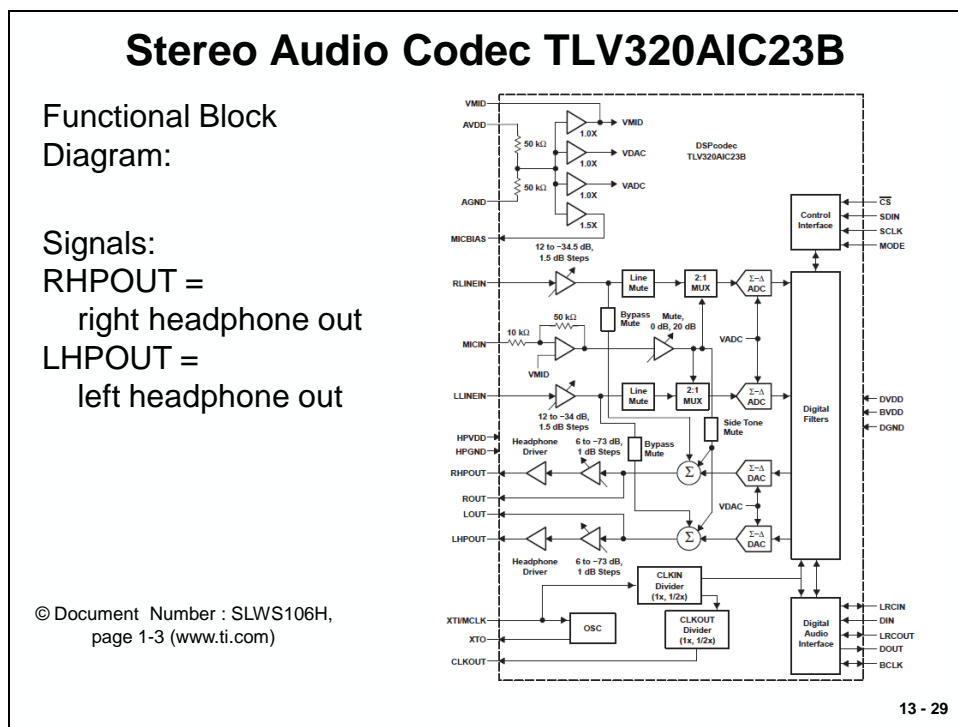
The AIC23 is connected to the F2833x by the following signal lines:

GPIO -MUX	F2833x - Function	Description	AIC23-signal
GPIO20 = 2	McBSPA - MDXA	Audio data out	DIN
GPIO21 = 2	McBSPA - MDRA	Audio data in	DOUT
GPIO22 = 2	McBSPA - MCLKXA	Transmit clock	BCLK
GPIO23 = 2	McBSPA - MFSXA	Transmit frame sync	LRCIN
GPIO58 = 1	McBSPA - MCLKRA	Receive Clock	BCLK
GPIO59 = 1	McBSPA - MFSRA	Receive frame sync	LRCOUT
GPIO16 = 1	SPIA - SPISIMO	Control data out	SDIN
GPIO18 = 1	SPIA - SPICLK	Control data clock	SCLK
GPIO19 = 1	SPIA - SPISTE	Slave trans. enable	/CS

Description:

- MXDA: serial audio data output stream from F2833x to AIC23

- MRDA: serial audio data input stream from AIC23 to F2833x

- MCLKXA: McBSPA Transmit Bit Clock.
            Generated by AIC23 (signal: "BCLK"; 12 MHz).

- MCLKRA: McBSPA Receive Bit Clock.
            Generated by AIC23 (signal: "BCLK"; 12 MHz).

- MFSXA: McBSPA Frame Sync Transmit.
            Generated by AIC23 (signal: "LRCIN"; 44.1 kHz).

- MFSRX: McBSPA Frame Sync Transmit.
            Generated by AIC23 (signal: "LRCIN"; 44.1 kHz).

- SPISIMO: SPIA "Slave In - Master Out" - signal.
            Generated by F2833x

- SPICLK: SPIA - clock signal.
            Generated by F2833x (1.0 MHz).

- SPISTE: SPIA - "Slave Transmit Enable".
            Generated by F2833x.

Signal "BCLK" is the data clock signal for the audio data transmission between the AIC23 and the F2833x. It defines the bit rate of the digital audio data stream. After a successful initialization of the AIC23, this 12 MHz - signal should look like:

Signal "LRCIN" is a 44.1 kHz - signal that starts the transmission of a new McBSP - frame. After a successful initialization it will look like this:
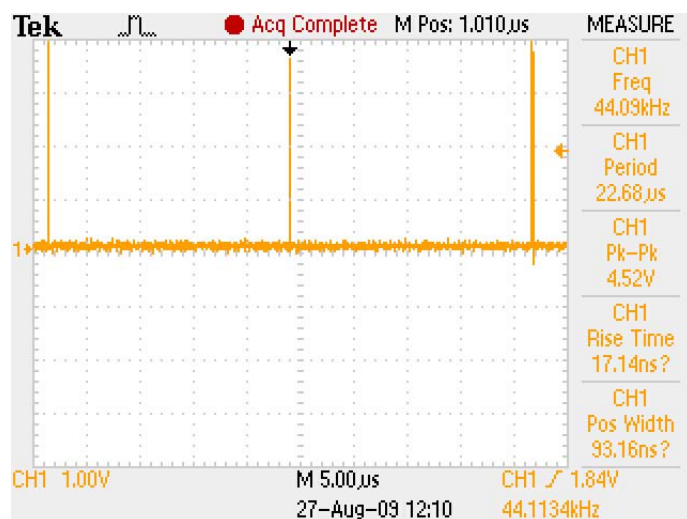
To generate the signal waveforms from above and to use the DAC of the codec to produce a sinusoidal stereo audio signal, we have to initialize the codec. The control channel of the AIC23 is connected to SPI-channel A of the F2833x (for SPI - details see also Chapter 10).

Initialization of SPI - channel A

The SPI communication between the F2833x and the AIC23 will be used to initialize the operating mode of the AIC23. For the SPI-A setup, we only need to perform a simple initialization:

- Register SPICCR:
    - SPICHAR: 16 bit character transmission
    - CLKPOLARITY: Clock polarity: Output on falling edge
    - SPILBK: no loopback mode
    - SPISWRESET: release from RESET
- Register SPIBRR:
    - SPI-clock = LSPCLK / (BRR +1)
- Register SPICTL:
    - MASTER_SLAVE: Master
    - CLK_PHASE: no
    - Disable interrupts
    - Enable Talk

Initialization of the AIC23

To initialize the AIC23, we have to send a series of SPI - commands. Let us inspect the command structure of the AIC23. A control word consists of 16 bits, starting with the MSB. The control word is divided into two parts. The first part (bits 15 - 9) is the AIC23 register address block; the second part (bits 8 - 0) is the data block.

The TLV320AIC23B has the following set of registers, which are used to program the modes of operation:

ADDRESS	REGISTER
0000000	Left line input channel volume control
0000001	Right line input channel volume control
0000010	Left channel headphone volume control
0000011	Right channel headphone volume control
0000100	Analog audio path control
0000101	Digital audio path control
0000110	Power down control
0000111	Digital audio interface format
0001000	Sample rate control
0001001	Digital interface activation
0001111	Reset register

When we initialize the AIC23 to accept a stereo audio data stream, we have to initialize this list of registers. This requires the merging of the 7-bit register address (15-9) with the 8-bit data (8-0), before it will be transmitted as a 16 - bit result via SPI. Please try to follow the given sequence:

1. Reset Register

Clear all 8 data bits to zero.

Stereo Audio Codec TLV320AIC23B

Reset Register:

Reset Register (Address: 0001111)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	RES	RES	RES	RES	RES	RES	RES	RES	RES
Default	0	0	0	0	0	0	0	0	0

RES   Write 000000000 to this register triggers reset

Power Down Control Register:

Power Down Control (Address: 0000110)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	OFF	CLK	OSC	OUT	DAC	ADC	MIC	LINE
Default	0	0	0	0	0	0	1	1	1

LINE = Line Input
MIC = Microphone Input
ADC = Internal ADC
DAC = Internal DAC
OUT = Output Signals   **0 = ON; 1 = OFF**
OSC = Oscillator
CLK = CLOCK
OFF = Device Power

13 - 31

2. Power Down Control Register

Switch OFF: LINE, MIC, ADC, DAC and OUT. Switch ON: OSC, CLK and OFF.

3. Left Headphone Volume Control Register

- Set Left Headphone Volume (LHV) to 0dB.
- Enable Left Zero Crossing (LZC) to update volume on zero crossing only.

Stereo Audio Codec TLV320AIC23B

Left Channel Headphone Volume Control Register:

Left Channel Headphone Volume Control (Address: 0000010)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	LRS	LZC	LHV6	LHV5	LHV4	LHV3	LHV2	LHV1	LHV0
Default	0	1	1	1	1	1	0	0	1

LRS = Left / Right simultaneous update volume (0 = OFF, 1 = ON)
LZC = Left channel zero cross (0 = OFF, 1 = ON). If ON, volume updates only at zero crossings
LHV = Left Headphone Volume (0x7F = +6dB; 0x79 = 0dB; 0x30 = -73dB (mute)

Right Channel Headphone Volume Control Register:

Right Channel Headphone Volume Control (Address: 0000011)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	RLS	RZC	RHV6	RHV5	RHV4	RHV3	RHV2	RHV1	RHV0
Default	0	1	1	1	1	1	0	0	1

RLS = Right / Left simultaneous update volume (0 = OFF, 1 = ON)
RZC = Right channel zero cross (0 = OFF, 1 = ON). If ON, volume updates only at zero crossings
RHV = Right Headphone Volume (0x7F = +6dB; 0x79 = 0dB; 0x30 = -73dB (mute)

13 - 32

4. Right Headphone Volume Control Register

- Set Right Headphone Volume (RHV to 0dB).
- Enable Right Zero Crossing (RZC) to update volume on zero crossing only.

5. Analogue Audio Path Control Register

- Switch ON the DAC.
- Mute the Microphone.
- Microphone boost = 0dB.
- Audio Input Select = Line.
- Disable ADC and DAC Bypass mode.
- Switch OFF the Microphone Side Tone

Stereo Audio Codec TLV320AIC23B

Analogue Audio Path Control Register:

Analog Audio Path Control (Address: 0000100)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	STA2	STA1	STA0	STE	DAC	BYP	INSEL	MICM	MICB
Default	0	0	0	0	0	1	0	1	0

MICB = Microphone boost (0 = 0dB; 1 = 20dB)
MICM = Microphone mute (0 = normal; 1 = muted)
INSEL = Input Select for Audio (0 = line; 1 = Microphone)
BYP = Bypass (0 = disabled; 1 = enabled (line in to line out))
DAC = DAC select (0 = DAC OFF; 1 = DAC ON)
STE = Added Side Tone (0 = OFF; 1 = ON)
STA = Side Tone Volume (If STE = ON, MIC is routed both to headphone & line out).

STE	STA2	STA1	STA0	ADDED SIDETONE
1	1	X	X	0 dB
1	0	0	0	−6 dB
1	0	0	1	−9 dB
1	0	1	0	−12 dB
1	0	1	1	−18 dB
0	X	X	X	Disabled

13 - 33

6. Digital Audio Path Control Register

- Enable ADC High Pass Filter, Disable De - emphasis, disable DAC soft mute.

Stereo Audio Codec TLV320AIC23B

Digital Audio Path Control Register:

Digital Audio Path Control (Address: 0000101)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	X	X	X	X	DACM	DEEMP1	DEEMP0	ADCHP
Default	0	0	0	0	0	1	0	0	0

ADCHP = ADC High Pass Filter (0 = enabled; 1 = disabled)
DEEMP = De-emphasis control (0 = disabled, 1 = 32kHz, 2 = 44.1kHz, 3 = 48kHz)
DACM = DAC soft mute (0 = disabled; 1 = enabled)

13 - 34

7. Digital Audio Interface Format Register

Stereo Audio Codec TLV320AIC23B

Digital Audio Interface Format Register:

Digital Audio Interface Format (Address: 0000111)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	X	MS	LRSWAP	LRP	IWL1	IWL0	FOR1	FOR0
Default	0	0	0	0	0	0	0	0	1

FOR = Data Format
    0 = MSB first, right aligned
    1 = MSB first, left aligned
    2 = I2S – Format, MSB first, left -1 aligned
    3 = DSP – Format; Frame sync followed by 2 words
IWL = Input word length 0 = 16 bit
    1 = 20 bit
    2 = 24 bit
    3 = 32 bit
LRP = DAC left /right phase   0 = right channel on and LRCIN = high
    1 = right channel on and LRCIN = low
LRSWAP = DAC left / right swap (0 = NO, 1 = YES)
MS = Master Mode (0 = Slave; 1 = Master)

13 - 35

- FOR: set to DSP-Mode
- IWL: set to 32 - bit mode
- LRP: Right Channel with LRCIN = low
- LRSWAP: no swap
- MS: set to Master

8. Sample Rate Control Register

- Set USB - Mode.
- Set BOSR to 272 fs.
- Set SR to 44.1 kHz for ADC and DAC

Stereo Audio Codec TLV320AIC23B

Sample Rate Control Register:

Sample Rate Control (Address: 0001000)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	CLKOUT	CLKIN	SR3	SR2	SR1	SR0	BOSR	USB/Normal
Default	0	0	0	1	0	0	0	0	0

USB = Clock Mode Select (0 = Normal, 1 = USB)
BOSR = Base Oversampling Rate
                    USB – Mode: 0 = 250fs, 1 = 272fs;
                    Normal – Mode: 0 = 256fs; 1 = 384fs

SR = Sampling Rate In the USB mode, the following ADC and DAC sampling rates are available:

SAMPLING RATE†		FILTER TYPE	SAMPLING-RATE CONTROL SETTINGS				
ADC (kHz)	DAC (kHz)		SR3	SR2	SR1	SR0	BOSR
96	96	3	0	1	1	1	0
88.2	88.2	2	1	1	1	1	1
48	48	0	0	0	0	0	0
44.1	44.1	1	1	0	0	0	1
32	32	0	0	1	1	0	0
8.021	8.021	1	1	0	1	1	1
8	8	0	0	0	1	1	0
48	8	0	0	0	0	1	0
44.1	8.021	1	1	0	0	1	1
8	48	0	0	0	1	0	0
8.021	44.1	1	1	0	1	0	1

CLKIN = Clock Input Divider (0 = MCLK; 1 = MCLK/2)
CLKOUT = Clock Output Divider (0 = MCLK; 1 = MCLK/2)

13 - 36

---

9. Digital Interface Activation Register

- Activate Interface

Stereo Audio Codec TLV320AIC23B

Digital Interface Activation Register:

Digital Interface Activation (Address: 0001001)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	RES	RES	X	X	X	X	X	ACT
Default	0	0	0	0	0	0	0	0	0

ACT = Activate Interface (0 = NO, 1 = YES)
RES = reserved

13 - 37

10. Power Down Control Register

- As a final step, activate everything except Microphone.

Power Down Control (Address: 0000110)

BIT	D8	D7	D6	D5	D4	D3	D2	D1	D0
Function	X	OFF	CLK	OSC	OUT	DAC	ADC	MIC	LINE
Default	0	0	0	0	0	0	1	1	1

1= OFF, 0 = ON

Lab Exercise 13_1: single audio tone

Objective

After the previous extensive (and probably boring) description of control registers for McBSP and the stereo audio codec AIC23B it is time for experiments. In Lab13_1 we will use the DAC of the AIC23B to synthesize a signal of 200 Hz on both the left and right head-phone channels. In Lab 13_2 we will improve the code to use interrupts. Finally, in Lab13_3 we will generate two independent signals on the left and right stereo channels. Since we will generate audio signals, they do have to be **sinusoidal** - we have to update the DAC periodically - exactly at 44.1 kHz. To do so, we will initialize the codec to operate at this frequency - a typical frequency for audio devices, such as CD - Players or MP3 - devices.

The question is: How do we generate a sinusoidal output value? One answer could be: "Use the trigonometric function y = sin(x)." However, the call to a trigonometric function is a time - consuming task and in time - critical applications we would need to make this function faster. A better solution is to use a "look-up-table". In such a table, we pre-calculate all the values, we will need in the code. For example, for y = sin(x), we could calculate 360 sine - values for a 360° unit circle, to obtain 360 points for interpolation.

Fortunately Texas Instruments already implemented a **sine - wave look - up table** in the boot - ROM memory of the F2833x! It consists of 512 entries for a unit circle of 360°, which will give us a next value at 360 degree / 512 = 0.7 degree. The values of this table are 32 bit wide and stored in fractional I2Q30 - format (range -2.0 ... +1.9999). The table starts at address 0x3FE000 with value y = sin (0), followed by value y = sin (0.7) and so on. We will use this table for the labs in this chapter.

But let us first inspect this sine look-up table (note: Q30 is currently not working in CCS4.1):

- Open a memory window (**View ➔ Memory**)
- In the top left corner of the memory window, enter address **0x3FE000**
- In the bottom left corner, change the data type to: **32 Bit Signed Integer**
- Open the properties of memory window (right mouse click) and change **Q Value to 30**:
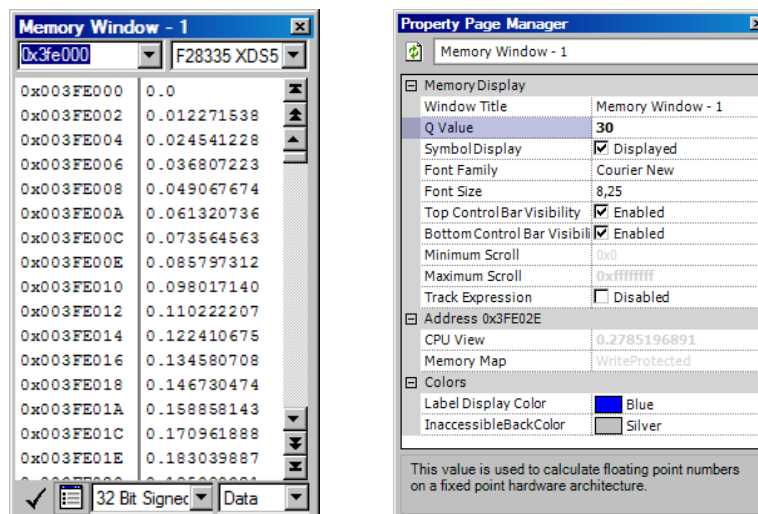
Preface

Before we start the laboratory procedure, it would be helpful to summarize the necessary steps to initialize the audio codec. The control channel of the AIC23B is connected to interface SPI-A. Of the F2833x and the audio data stream is interfaced to McBSP. Here is what we have to do:

SPI - A - Initialization

To initialize the control channel of the AIC23B, we have to set up the SPI - A unit of the F2833x. Since we will not use SPI - interrupts or SPI - FIFO units for this first exercise, the initialization sequence is quite simple:

For Register **"SPICCR"**:

- First hold SPI-A in Reset (Bit "SPIWRESET")
- Select16 - bit Mode (Bit field "SPICHAR")
- Select output data on falling edge of clock (Bit "CLKPOLARITY")
- Disable loopback Mode (Bit "SPILBK")
- Finally, release SPI - A from Reset (Bit "SPIWRESET")

For Register **"SPICTL":**

- Select Master Mode (Bit "MASTER_SLAVE")
- Select no clock delay (Bit "CLK_PHASE")
- Enable SPI - transmissions (Bit "TALK")
- Disable Interrupts (Bits "SPIINTENA" and "OVERRUNINTENA")

For Register **"SPIBRR":**

- Although the data rate of this channel is not really critical, because SPI-A is used for initialization purpose only, let us agree to initialize it to 1 MBit/s. Assuming that you have not changed the provided function "InitSysCtrl()" from Texas Instruments Header Files, your F2833xControlCard will run at 150 MHz (SYSCLKOUT) and with a LSPCLK = 37.5 MHz The SPI - data rate is calculated as:

$$SPI - data - rate = \frac{LSPCLK}{SPIBRR + 1}$$

- Load Register SPIBRR with the calculated value.

In preparation for Lab13_1, we can write a new function "SPIA_init()", which covers the initialization steps discussed above.

AIC23B Control Command Sequence

Next, we need to prepare the control sequence for the AIC23B. Please refer to pages 13-27 of this textbook module, where we discussed the AIC23B registers. Now we have to define the initialization sequence:

1. Apply a Reset - command to the Reset Register.
   Hint: The address of the Reset-Register is 0001111 (see Slide 13-31). A reset command consists of 9 zero bits 0000 0000 0. The resulting 16 - bit value is 0x1E00. Send this value as first command via SPI-A to the AIC23B.

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

2. Address the Power - Down - Control Register (Slide 13-31). For now switch OFF "Line", "Mic", "ADC", "DAC", and "OUT". Switch to ON bits for "OSC", "CLK" and "OFF". Fill in the values for all 16 bits in the table below:

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

3. Address the Left Headphone Control Register (Slide 13-32). Switch ON "LZC" and set "LHV" to volume = 0 db. LZC is the left zero crossing. When ON, it changes volume levels only at the zero crossing of the sinusoidal signal.

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

4. Address the Right Headphone Control Register (Slide 13-32). Switch ON "RZC" and set "RHV" to volume = 0 db. RZC is the right zero crossing. When ON, it changes volume levels only at the zero crossing of the sinusoidal signal.

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

5. Address the Analogue Audio Path Control Register (Slide 13-33). Enable the DAC and mute the microphone. Disable bypass and side tone. Set Audio Input to "Line".

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

6. Address the Digital Audio Path Control Register (Slide 13-34). Set all 9 data bits to zero.

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

7. Address the Digital Audio Interface Format Register (Slide 13-35). Set the AIC23B to Master Mode (MS). Set the data format to DSP (FOR). Set the input word length to 32 bit (IWL). Set LRP to 1 (right channel when LRCIN = low). Do not swap the DAC - channels (LRSWAP).

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

8. Address the Sample Rate Control Register (Slide 13-36). Select USB-Mode (USB). Select 44.1 kHz both for ADC and DAC (SR and BOSR). Set CLKIN and CLKOUT to MCLK.

9. Address the Digital Interface Activation Register (Slide 13-37). Activate the interface (ACT).

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

10. Address the Power - Down - Control Register (Slide 13-31). Turn on everything except the microphone (MIC).

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

McBSP - A Initialization Sequence

To be able to send audio data to the AIC23B we also have to initialize McBSP-A.

1. Register Serial Port Control 1 (**SPCR1**) - see Slide 13-18:

   - Reset Receiver. We don't use the McBSP-Receiver for Lab13_1

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

2. Serial Port Control 2 Register (**SPCR2**) - see Slide 13-19:
   - Do not stop McBSP on JTAG - Breakpoints (Bit "FREE")
   - Request Transmit Interrupt Service on Frame Sync Pulse (Bit field "XINTM")
   - Release Frame Sync Logic from Reset (Bit "FRST")
   - Release Sample Rate Generator from Reset (Bit "GRST")
   - Set all remaining bits to zero.

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

3. Interrupt Enable Register (**MFFINT**) - see Slide 13 - 27:
   - Enable Transmit Interrupts (XINT)
   - Disable Receive Interrupts (RINT)

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

4. Transmit Control Register 1 (**XCR1**) - see Slide 13 - 22:

   - Select "32 - bit" data in phase 1 (XWDLEN1)
   - Select 1 word in phase 1 (XFRLEN1)

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

5. Transmit Control Register 2 (**XCR2**) - see Slide 13 - 23:

   - Select "Dual Phase" - Frame (XPHASE). We will use phase 1 for the left audio signal and phase 2 for the right signal
   - Select "32 - bit" data in phase 2 (XWDLEN2)
   - Select 1 word in phase 2 (XFRLEN2)
   - Set data delay to 1 clock cycle after frame sync (XDATDLY)
   - Disable Compand, MSB first (XCOMPAND)

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

6. Sample Rate Generator Register 2 (SRGR2) - see Slide 13 - 25:

    - Select pin MCLKX as source for sample clock (bit "CLKSM").

    - Set all remaining bits to zero.

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

7. Pin Control Register (PCR) - see Slide 13 - 26:
    - Select external pulses for Receive and Transmit Frame Sync (Bits "FSXM" and "FSRM").
    - Select external clock from pin MCLKX (Bit "CLKXM")
    - Select external clock from pin MCLKR (Bit "CKLRM")
    - Set Bit "SCLKME" = 1 to select MCLKX as clock source for sample rate generator
    - Select Frame Sync Polarity as "active low" (Bits "FSXP" and "FSRP")
    - Select Transmit Clock polarity to "rising edge" (Bit "CLKXP")
    - Select Receive Clock Polarity to "rising edge" (Bit "CLKRP")

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

AIC23B Exercises:

Lab13_1:
- Initialize SPI-A as control channel for AIC23B
- Initialize McBSP-A as data channel for AIC23B
- AIC23B is master and sends the 12MHz base clock
- AIC23B sends a 44.1 kHz frame sync signal to McBSP
- Send a sinusoidal signal, based on the BOOT-ROM look-up table to the DAC of the AIC23B; sample rate is 44.1 kHz

Lab13_2:
- Send two different signals to left and right audio channel
- Add volume control

Lab13_3:
- Improvement of Lab13_2; reduce Interrupt Service time

13 - 38

Procedure

Now we are ready to code a new project for Lab13_1:

Open Files, Create Project File

1. Using Code Composer Studio, create a new project called **Lab13.pjt** in C:\DSP2833x_V4\Labs (or in another path that is accessible by you; ask your teacher or a technician for an appropriate location!).

2. A good point to start with is the source code of Lab6.c, which produces a hardware based time period using CPU core timer 0. Open file Lab6.c from C:\DSP2833x_V4\Labs\Lab6 and save it as Lab13_1.c in C:\DSP2833x_V4\Labs\Lab13.

3. Define the size of the C system stack. In the project window, right click at project "Lab13" and select "Properties". In category "C/C++ Build", "C2000 Linker", "Basic Options" set the C stack size to 0x400.

Link some of the source code files, provided by Texas Instruments, to the project:

4. In the C/C++ perspective, right click at project "Lab13" and select "**Link Files to Project**". Go to folder "*C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\source*" and link:

    - **DSP2833x_GlobalVariableDefs.c**

    From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source* link:
    - **DSP2833x_PieCtrl.c**
    - **DSP2833x_PieVect.c**
    - **DSP2833x_DefaultIsr.c**
    - **DSP2833x_CpuTimers.c**
    - **DSP2833x_SysCtrl.c**
    - **DSP2833x_CodeStartBranch.asm**
    - **DSP2833x_ADC_cal.asm**
    - **DSP2833x_usDelay.asm**

    From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\cmd* link:

    - **DSP2833x_Headers_nonBIOS.cmd**

Project Build Options

5. We have to extent the search path of the C-Compiler for include files. Right click at project "Lab13" and select "Properties". Select "C/C++ Build", "C2000 Compiler", "Include Options". In the box: "Add dir to #include search path", add the following lines:

    C:\tidcs\C28\dsp2833x\v131\DSP2833x_headers\include

C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\include

Note: Use the "Add" Icon to add the new paths:
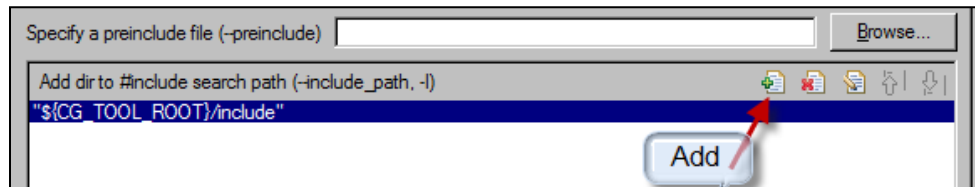
Close the Property Window by Clicking <**OK>**.

Preliminary Test

6. So far we only created a new project "Lab13.pjt", with the same functionality as in Lab6. A good step would be to rebuild Lab13, load the code into the controller and verify the binary counter at LEDs LD1 to LD4 of the Peripheral Explorer Board.

   The LEDs should display the counter at 100 milliseconds time steps.

   If not: Debug!

Change the GPIO - Multiplex Registers

7. Open file "Lab13_1.c" and edit the function "Gpio_select()":

   - The control channel of the AIC23B is connected to the SPI-A interface. Change the setup for multiplex register "GPAMUX2" to use SPI-A at pins GPIO16 (SPISIMO), 18(SPICLK) and 19(SPISTE).

   - The digital audio data channel to and from the AIC23B uses the McBSP-A interface. Change the setup for multiplex register "GPAMUX2" to use McBSP-A functions on pins GPIO20 (MDXA), 21(MDRA), 22 (MCLKXA) and 23 (MFSXA). With the help of register "GPAQSEL2", set all four lines to "asynchronous".

   - For register "GPBMUX2", select two more McBSP-A signals at pin 58 (MCLKRA) and pin 59 (MFSRA).

Remove code from Lab6

8. In the function "main()", remove the local variable "counter" and the all the code inside the while(1) -loop that is related to the variable "counter".

   Please note: although we do not need CPU Timer 0 and its interrupt service routine for lab exercise 13_1, we keep this unit in our code, (a) as an exercise to have multiple interrupt sources and (b) as a placeholder for further extensions to this lab exercise.

Add SPI-A Initialzation Code

9.   At the end of "Lab13_1", add a function "SPIA_Init()" to initialize the interface for SPI-A. In the preface of this exercise (pages 13-34 to 13-37), we discussed all necessary initialization steps.

10. At the beginning of "Lab13_1", add a function prototype for "SPIA_Init()".

11. In "main()", after the function call to "Gpio_select()", add a function - call to "SPIA_Init()".

Add McBSP-A Initialzation Code

12. At the end of "Lab13_1", add the function "McBSPA_Init()" to initialize the interface for McBSP-A. In the preface of this exercise (pages 13-36 to 13-37), we discussed all the necessary initialization steps.

13. At the beginning of "Lab13_1", add a function prototype for "McBSPA_Init()".

14. In "main()", after the function call to "SPIA_Init()", add a function - call to "McBSPA_Init()".

Initialize the codec AIC23B

15. At the end of "Lab13_1" add the function "AIC23_Init()" to initialize the AIC23B stereo audio codec. The initialization process is a sequence of SPI - A commands that must be transmitted from the F2833x to the AIC23. In the Preface for this lab we discussed the necessary commands (pages 13-35 and 13-36).

The code in function "AIC23_Init()" must consist of a series of write commands of 16-bit words into register "SPITXBUF", followed by a wait construction for the end of the transmission and a dummy read of SPIRXBUF to clear the interrupt flag:

```
SpiaRegs.SPITXBUF = command_1;
while (SpiaRegs.SPISTS.bit.INT_FLAG != 1);
i = SpiaRegs.SPIRXBUF;
SpiaRegs.SPITXBUF = command_2;
while (SpiaRegs.SPISTS.bit.INT_FLAG != 1);
i = SpiaRegs.SPIRXBUF;
```

16. At the beginning of "Lab13_1", add a function prototype for "AIC23_Init()".

17. In "main()", after the function call to "McBSPA_Init()" add a function - call to "AIC23_Init()".

Change the Interrupt Structure for Lab13_1

18. We will use the McBSP - Transmit Frame - Sync signal, which is generated by the AIC23B at a frequency of 44.1 kHz, to request an interrupt service. This interrupt service routine will be used to send the next pair of stereo amplitude values for the sinusoidal audio signal to the AIC23B. In main, search for the line, that we used to

overload the PieVectTable with the address of function "cpu_timer0_isr()". After that line add:

> **PieVectTable.MXINTA = &McBSP_A_TX_isr;**

19. By adding a new code - line enable the PIE - interrupt - line McBSP-A (which is PIE line 6, interrupt number 6):

> **PieCtrlRegs.PIEIER6.bit.INTx6 = 1;**

20. Change the line to enable interrupt lines in Register **IER**. Now we have to enable line INT1 (CPU - Timer 0) and line INT6 (McBSP-A).

21. At the beginning of "Lab13_1", add a function prototype for the interrupt function "McBSP_A_TX_isr()".

Add global variables and IQ-Math

22. At the beginning of "Lab13_1.c", add two global variables:

> **float volume = 0.2;   // relative sound volume control (0...1)**
> **unsigned int fsin= 220;   // audio stereo sine frequency in Hertz**

23. Also at the beginning of "Lab13_1.c" add a global array variable "sine_table[512]" to obtain access to the ROM - sine - value lookup-table. This table consists of 512 values for a unit circle (see page 13-33) in I2Q30 - format. A "pragma DATA_SECTION" directive will connect this variable to a linker symbol "IQmathTables", which is defined in file "28335_RAM_lnk.cmd":

> **#pragma DATA_SECTION(sine_table,"IQmathTables");**
> **_iq30 sine_table[512];   // lookup-table 512 values in I2Q30 (+1...-1)**

24. The new data type "_iq30" is defined in another header file, provided by Texas Instruments and called "IQmathLib.h". Include this file in your source code "Lab13_1.c":

> **#include "IQmathLib.h"**

25. We have to extent the include search path. Right click at project "Lab13" and select "Properties". Select "C/C++ Build", "C2000 Compiler", "Include Options". In the box: "Add dir to #include search path", add the following line:

> **C:\tidcs\c28\IQmath\v15a\include**

Note: Depending on the installation on your PC, the IQ-Math library can be installed at a different location. If the IQ-math - library is not installed at all, search for "sprc087" on Texas Instruments website (www.ti.com) and download it from there.

26. Link the IQ-Math - Library to your project. From C:\tidcs\c28\IQmath\v15a\lib, link:

> **IQmath.lib**

Calculate new DAC - Value

Before we go into the coding details, we should discuss how to calculate the next amplitude based on the sine-lookup-table in boot-ROM.

The sine table consists of 512 entries for 360 degrees. If we would read the next value each time we get a frame synch pulse (f_{frame} = 44.1 kHz), we would generate a resulting signal of

$$f_{signal} = \frac{f_{frame}}{512} = \frac{44.1\ kHz}{512} = 86\ Hz \qquad\qquad [1]$$

To produce a higher signal frequency f_{signal}, we have to reduce the number of sample points taken from the table. For example, if we were to take every other sample point (Step_Size = 2), we would obtain:

$$f_{signal} = \frac{f_{frame}}{512} = \frac{44.1\ kHz}{\frac{512}{Step\_Size}} = \frac{StepSize * 44.1\ kHz}{512} = \frac{2 * 44.1\ kHz}{512} = 172\ Hz \qquad [2]$$

Rearranging this formula to Step_Size gives:
$$Step\_Size = \frac{f_{signal} * 512}{f_{frame}} \qquad\qquad [3]$$

For example to produce a signal of f_{signal} = 100 Hz we would need a Step_Size of 1.16.

To calculate the next index into the look-up - table (*lut_idx*) we have to multiply the linear index by Step_Size:

$$lut\_idx = linear\_idx * Step\_Size \qquad\qquad [4]$$

The linear index "linear_idx" (0…511) is incremented with each 44.1 kHz - Interrupt. The look-up - table index "lut_idx" is the resulting integer index into the sine - table. The final equation to calculate the next index is:

$$lut\_idx = linear\_idx * \frac{f_{signal} * 512}{f_{frame}} \qquad\qquad [5]$$

This formula will be used to read the next lookup table value, which is a value between +1 and -1 in I2Q30 - Format. We will multiply this value with the relative value "volume" (0…1) for volume control. A conversion function "_IQ30()" can be used to convert a float type "volume" into an I2Q30 type. The multiply is done using function "_IQ30mpy()":

amplitude = _IQ30mpy(sine_table[lut_idx],_IQ30(volume));   [6]

The result of the multiply operation is an I2Q30 - number. The AIC23B expects it as a 'per unit' value or, in other words an I1Q31 - number. When we left shift the amplitude by 1 bit, we have the final value for the AIC23B:

amplitude <<= 1;   [7]

Now let us resume the lab procedure!

Add McBSP - Transmit Interrupt Service

27. At the beginning of "Lab13_1.c", add a function prototype for a new interrupt service function:

interrupt void McBSP_A_TX_isr(void);

28. At the end of "Lab13_1.c", add a new interrupt service function "McBSP_A_TX_isr()". This function will be called by the frame sync signal (MFSXA), which is produced by the AIC23B as signal "LCRIN" at a frequency of 44.1 kHz. With this pulse, the AIC23B requests a new pair of amplitude values for the right and left stereo audio channel of its DAC. We have to include the following steps in the function "McBSP_A_TX_isr()":

- First, we need three local variables:

  ### static unsigned int linear_idx=0;
  ### unsigned int lut_idx;
  ### long amplitude;

- Next, calculate "lut_idx" according Equation [5] from page 3-42.

- If the result of the calculation for "lut_idx" is greater than 511, reset both "linear_idx" and "lut_idx" to zero.

- Calculate the new amplitude for the audio signal according equation [6] and [7].

- Load register McspaRegs.DXR2 with the upper 16 bits of "amplitude".

- Load register McspaRegs.DXR1 with the lower 16 bits of "amplitude".

- Increment variable "linear_idx". If the value exceeds 511, reset it to zero.

- Finally, acknowledge PIE - Interrupt - Group 6:

  ### PieCtrlRegs.PIEACK.bit.ACK6 = 1;

Build, Load and Run

29. Click the "Rebuild Active Project " button or perform:

Project → Rebuild All (Alt +B)

30. Load the output file in the debugger session:

Target → Debug Active Project

and switch into the "Debug" perspective.

31. Verify that in the debug perspective the window of the source code "Lab13_1.c" is high-lighted and that the blue arrow for the current Program Counter position is placed under the line "void main(void)".

32.   Perform a real time run.

Target ➜ Run

33.   Connect headphones to J25 (Headphone - Out) of the Peripheral Explorer Board. A signal of 220 Hz should be audible on the right and left ear pieces. Or use an oscilloscope to measure the signal amplitudes on J25 left and right channels:
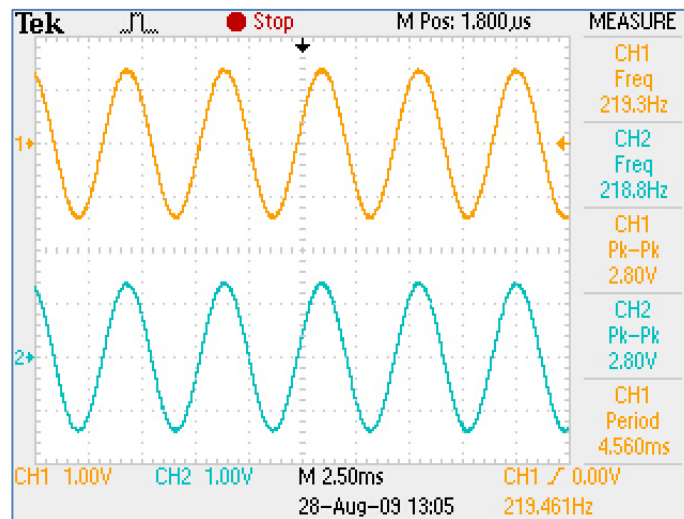
Right (yellow) and left (blue) stereo audio signal at headphone connector J25

34.   Open a Watch Window and modify the values for variables "fsin" and "volume" to change the signal on the headphones:

Note: If you use "real-time debug mode", which we already used in previous chapters, you can change the variables whilst the code is running!

Lab Exercise 13_2: Dual audio tone

Objective

In Lab13_1, we generated a single frequency on both audio channels. Now we will generate two different frequencies for the left and right channels. Recall that we initialized the AIC23B in dual phase operating mode. In this mode each frame sync signal (MFSXA) starts the transmission of two 32 Bit words from McBSP to the AIC23B. The first 32-bit word (phase 1) is the new data for the left channel and the 2nd 32-bit word (phase 2) is for the right channel.

In Lab13_1 we loaded the 32 - bit register-pair DX2/DX1 only once. As a result, the McBSP transmitted the same word twice, for the left and right channels. For the new lab, we have to load a 2nd 32-bit number into DX2/DX1, after the first one has been transmitted to drive the two channels with different signals.

Procedure

Open Project, Modify Source File

1. If not still open from Lab13_1, re-open project Lab13.pjt in C:\DSP2833x_V4\Labs.

2. Open the file "Lab13_1.c" and save it as "Lab13_2.c"

3. Exclude file "Lab13_1.c" from build. Use a right mouse click at file "Lab13_1.c", and enable "Exclude File(s) from Build".

4. In "Lab13_2.c", replace the variable "fsin" by two new global variables:

    unsigned int fsine_left= 110;   // left audio frequency in Hertz
    unsigned int fsine_right = 220; //right audio frequency in Hertz

Next, we have to change the interrupt service routine "McBSP_A_TX_isr()". The calculation for the next amplitude must now be performed independently for the left and right signals.

5. In the function "McBSP_A_TX_isr()" double the local variables, one set for left and one set for the right channel:

    static unsigned int left_linear_idx=0, right_linear_idx;
    unsigned int left_lut_idx,right_lut_idx;
    long left_amplitude,right_amplitude;

6. Using the new set for the "left"- variables, change the code sequence to calculate "amplitude" into a sequence to calculate "left_amplitude".

7. Add a similar calculation for value "right_amplitude".

8. Load registers DX2 and DX1 with the upper and lower halves of the variable "left_amplitude".

9. Add a waiting loop to wait until the first 32 - bit value has been copied into the McBSP - shift Register XSR2/XSR1. The status flag "XRDY" is set in such a case:

while(McbspaRegs.SPCR2.bit.XRDY == 0);

> Note: The line above is a wait construction, which should never be used in an interrupt service routine of a real project. The two basic rules of coding ISRs are (1) keep ISRs as short as possible and (2) never include wait loops, because they can stall the whole project. However, since we are learning students, we are allowed to do everything (unless your teacher intervenes…). In "Lab3_3" we will improve the code in a way that we can avoid the wait - construction from above!

10. After the wait - construction, add two more load instructions for registers DX2 and DX1, now for value "right_amplitude".

11. Change the code to increment and limit variable "linear_idx" into "left_linear_idx".

12. Add similar instructions as in step 11 for the variable "right_linear_idx".

Build, Load and Run

13.   Click the "Rebuild Active Project " button or perform:

Project → Rebuild All (Alt +B)

14.   Load the output file in the debugger session:

Target → Debug Active Project

and switch into the "Debug" perspective.

15.   Verify that in the debug perspective the window of the source code "Lab13_2.c" is high-lighted and that the blue arrow for the current Program Counter position is placed under the line "void main(void)".

16.   Perform a real time run.

Target → Run

17.   Connect headphones to J25 (Headphone - Out) of the Peripheral Explorer Board. A signal of 220 Hz should be audible on the right channel and a signal of 110 Hz on the left channel. Or, use an oscilloscope to measure the signal amplitudes on J25 left and right channels:
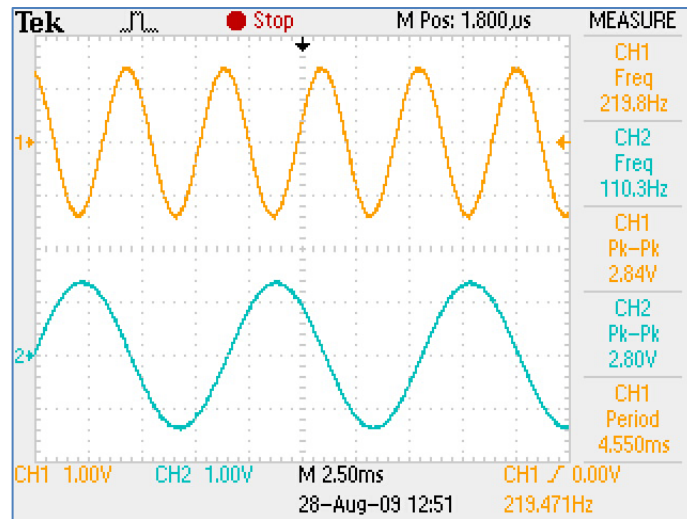
Right Channel (yellow) and Left Audio Channel (blue)

18.   Using a Watch Window change the signals "fsine_left" and "fsine_right" of the head-phone:

Note: If you use "real-time debug mode", which we used in previous chapters, you can change the variables whilst the code is running!

Lab 13_3: Dual audio tone and XRDY - Interrupt

Objective

In Lab13_2 we used an interrupt service routine with a wait construction in it. For a real project such a technique is unacceptable. Therefore we have to improve our AIC23B Lab now!

The solution is based on using a different interrupt source. Labs13_2 was based on frame sync - interrupts, which were caused by the AIC23B at 44.1 kHz. In response, we had to load two pairs of values into registers DX2/DX1. But before we could load the 2nd pair, we had to wait, because there is only one register pair DX2/DX1. This was the reason for the unfortunate wait-loop in Lab13_2.

For Lab13_3 we will use the XRDY - signal to request an interrupt service. This interrupt is triggered each time a 32 - bit word is loaded from DX2/DX1 into the McBSP-internal shift registers XSR2/XSR1. The idea for the new lab is this:
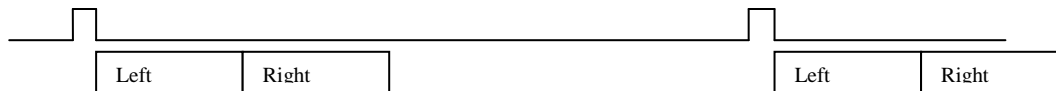
In the middle of the left transmission we get the 1st XRDY - interrupt. We use this ISR to load the right channel data into DX2/DX1. Similar, in the middle of the right transmission, we get the 2nd XRDY - interrupt. We use this ISR to load the next left channel data into DX2/DX1 - in preparation of the next transmission, which will be started by the next external frame - sync - signal (MFSXA).

Summary: We will use an alternating technique in the interrupt service for XRDY. Every odd interrupt will load the next left value, every odd interrupt the next right value.

Procedure

Open Project, Modify Source File

1.   If not still open from Lab13_2, re-open project Lab13.pjt in C:\DSP2833x_V4\Labs.

2.   Open the file "Lab13_2.c" and save it as "Lab13_3.c"

3.   Exclude file "Lab13_2.c" from build. Use a right mouse click at file "Lab13_2.c", and enable "Exclude File(s) from Build".

4.   In the function "McBSPA_Init()", change the initialization for the register "SPRC2", to now request a transmit interrupt service on event "XRDY" (hint: change bit "XINTM").

5.   Change interrupt service function "McBSP_A_TX_isr()"

- Add a new local variable "even". We will use this variable as a switch to alternately execute the reload code for phase 1 (left channel) or phase 2 (right channel):

static unsigned int even = 0;//switch reload phase1 / phase2

- Next, add an if-statement and include all instructions that are related to the "left"-variables into the "TRUE" - block and all "right"-variables into the "FALSE" - block:

```
if (even == 0)
{
        // Calculation for amplitude left channel
        // load left_amplitude to DX2/DX1
        // increment and limit left_linear_idx
        // even =  1
}
else
{
        // Calculation for amplitude right channel
        // load right_amplitude to DX2/DX1
        // increment and limit right_linear_idx
        // even = 0
}
```

6.   The McBSP - Transmit - Interrupt is now based on XRDY. This signal is generated after a word has been loaded from DX2/DX1 into XSR2/XSR1. To get the first interrupt, we have to force a load into DX2/DX1. In main, just before we enter the endless while(1) loop, add the two following lines:

McbspaRegs.DXR2.all = 0;

McbspaRegs.DXR1.all = 0;

This code will force a first XRDY - interrupt.

Build, Load and Run

7.   Click the "Rebuild Active Project " button or perform:

Project → Rebuild All (Alt +B)

8.   Load the output file in the debugger session:

Target → Debug Active Project

and switch into the "Debug" perspective.

9.   Verify that in the debug perspective the window of the source code "Lab13_3.c" is high-lighted and that the blue arrow for the current Program Counter position is placed under the line "void main(void)".

10. Perform a real time run.

    Target ➔ Run

11. Connect headphones to J25 (Headphone - Out) of the Peripheral Explorer Board. A signal of 220 Hz should be audible on the right channel and a signal of 110 Hz on the left channel. Or, use an oscilloscope to measure the signal amplitudes on J25 left and right channel:
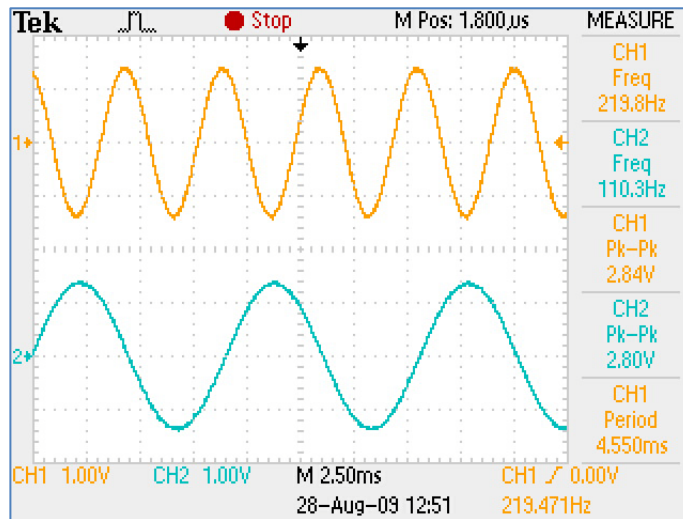
Right Channel (yellow) and Left Audio Channel (blue)

12. Using a Watch Window change the signals "fsine_left" and "fsine_right" of the head-phones:

Note: If you use "real-time debug mode", which we used in previous chapters, you can change the variables whilst the code is running!

Lab Exercise 13_4: EEPROM via McBSP

Objective

The Peripheral Explorer Board is equipped with a serial SPI - EEPROM (CSI25C256K or AT25256). This lab exercise will write and read into this memory.

Write:
We will use the Peripheral Explores Boards push-button PB1 (GPIO17) to start a write to the EEPROM. The data to be written into the EEPROM is the current position of the 4 - bit hexadecimal digital input encoder (GPIO12...15). Only bits 3...0 of the EEPROM memory address will be used.

Read:
We will use push-button PB2 (GPIO48) to read the EEPROM. From the 8 bit data only bits 2...0 will be displayed on LEDs LD4 (GPIO49), LD3 (GPIO34), and LD1 (GPIO9). Note: LED LD2 (GPIO11) cannot be used for this exercise, because the Peripheral Explorer Board uses this output line to control the "chip-select" signal (/CS) of the EEPROM.

EEPROM AT25256 Exercise:

Lab13_4:
- Initialize McBSP-B in SPI-Mode for AT25256
- Write data to EEPROM, if button PB1 (GPIO17) is pushed. Read the current value from Hex-Encoder (GPIO12…15) and store it into EEPROM-address 0x0040, bits 3…0.
- Read data from EEPROM-address 0x0040, when button PB2 (GPIO48) is pushed and display bits 2…0 at LEDs LD4 (GPOI49), LD3 (GPIO34) and LD1 (GPIO9).

13 - 39

Hardware Description:

The SPI - Interface of this device is connected to the F28335 - McBSP - channel B. The following schematic gives the hardware - details:
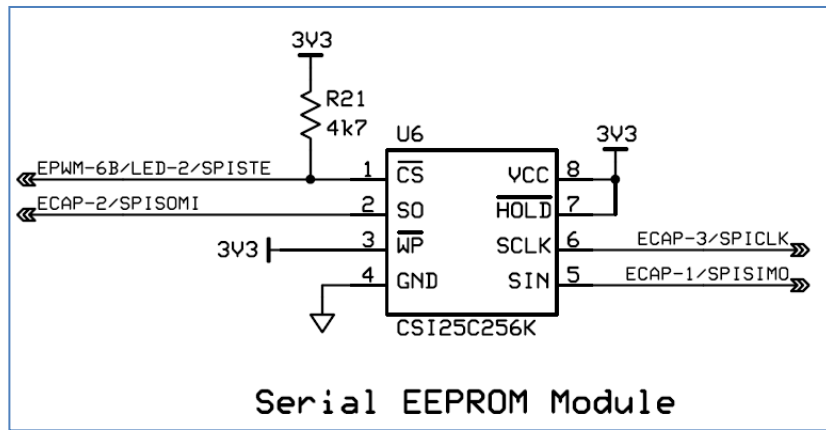

Peripheral Explorer Board EEPROM Circuitry

- Input pin "/CS" connected to GPIO11, which could be initialized as Signal "SPI-Slave Transmit Enable (SPISTE)". However, since we will use McBSP-B in a SPI operating - mode, we are not able to generate "SPISTE" from this interface. Therefore we will use GPIO11 as digital output line controlled by software.

- Output pin "Slave Out (SO)" is connected to GPIO25, which can be initialized as McBSP-signal "MDRB". In SPI - mode, this pin will operate as "Slave Out Master In" signal.

- Input pin "Slave Clock (SCLK)" is connected to GPIO26, which can be initialized as McBSP-signal "MCLKB". In SPI - mode this pin will operate as "SPI - clock" signal.

- Input pin "Slave In (SIN)" is connected to GPIO24, which will be initialized as McBSP-signal "MDXB". In SPI-mode, this pin will feature the "Master Out Slave In" data signal.

Timing Diagram

The AT25256 has the following timing requirements:
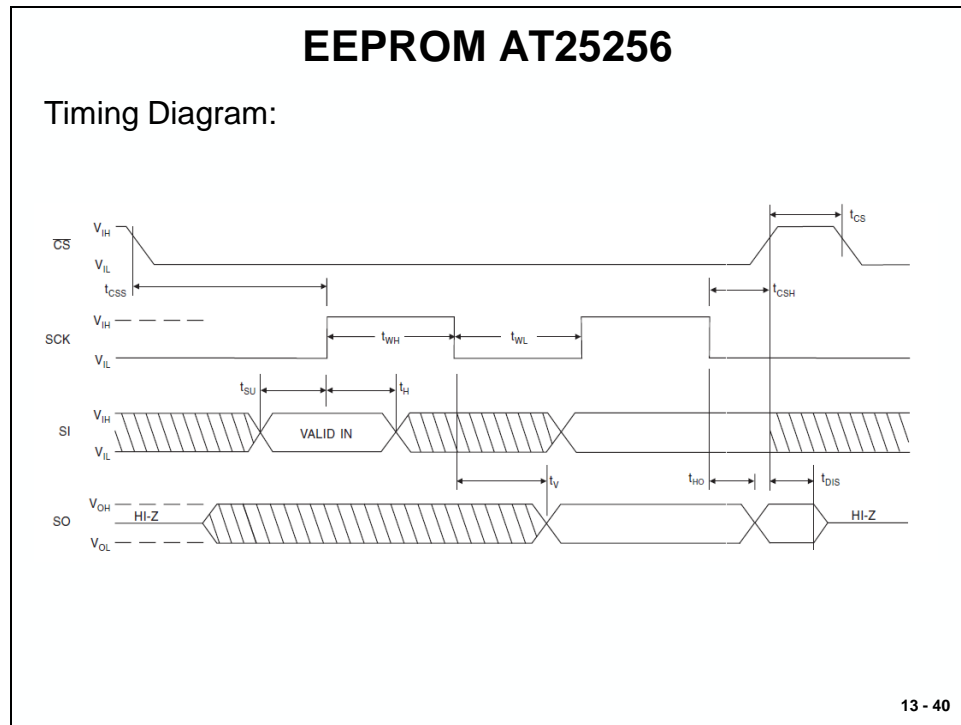
Source: http://www.atmel.com document "doc0872.pdf"
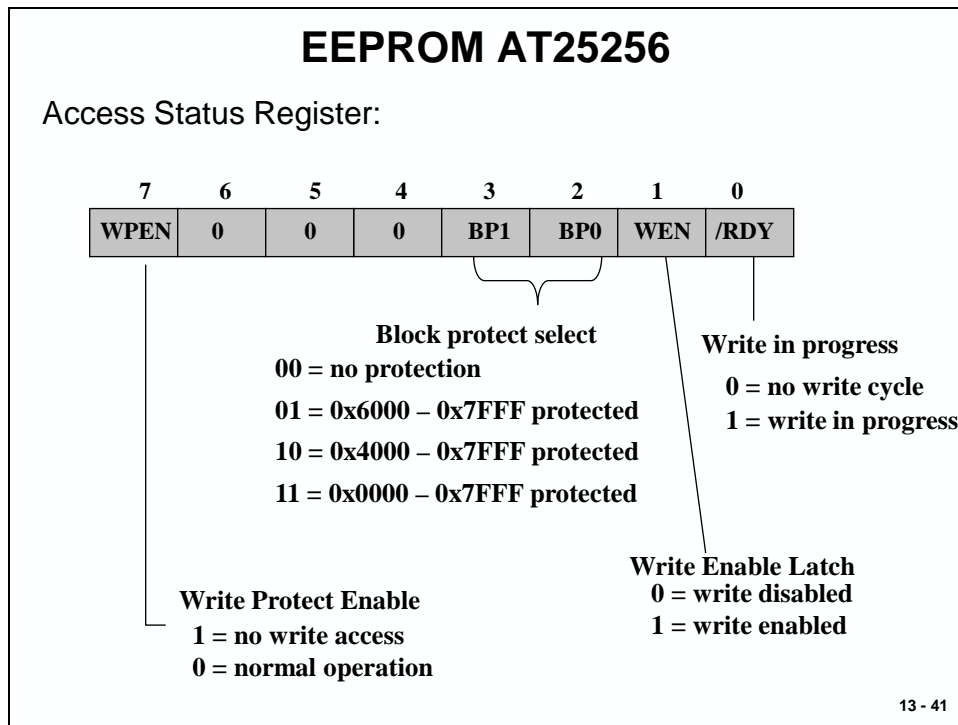
An access cycle is enclosed within an active /CS - signal. At the beginning, we will need to set /CS low and when we have transmitted the frame, we need to de-activate /CS by making it high again.

To write data into the EEPROM, the F2833x has to generate the data bit first; with a clock delay of ½ cycles, the rising edge is the strobe pulse for the EEPROM to store the data.

When reading the EEPROM, the falling clock edge causes the EEPROM to send out data and the rising clock edge the F2833x can read the valid data bit.

AT25256 Status Register

The AT25256 internal Status Register controls write accesses to the internal memory.

```
                    EEPROM AT25256

        Access Status Register:


          7     6     5     4     3     2     1     0
        ┌──────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
        │ WPEN │  0  │  0  │  0  │ BP1 │ BP0 │ WEN │ /RDY│
        └──────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
                                   └──┬──┘

                        Block protect select         Write in progress
                      00 = no protection              0 = no write cycle
                      01 = 0x6000 – 0x7FFF protected  1 = write in progress
                      10 = 0x4000 – 0x7FFF protected
                      11 = 0x0000 – 0x7FFF protected
                                                   Write Enable Latch
                                                   0 = write disabled
                                                   1 = write enabled
                       Write Protect Enable
                       1 = no write access
                       0 = normal operation

                                                          13 - 41
```

It also flags the current status of the EEPROM:

Bit 0 ("/RDY') flags whether an internal write cycle is in progress or not. Internal write cycles are started at the end of a command sequence and last quite long (maximum 10ms). To avoid the interruption of a write cycle in progress any other write access should be delayed as long as /RDY=1.

Bit 1("Write Enable Latch") is a control bit that must be set to 1 for every write access to the EEPROM. After a successful write cycle this bit is cleared by the EEPROM.

Bits 3 and 2("Block Protect Select") are used to define the area of memory that should be protected against any write access. We will not use any protection in our lab, so just set the two bits to '00'.

Bit 7 ("Write Protect Enable") allows us to disable any write access into the Status Register. For our Lab we will leave this bit cleared all the time (normal operation).
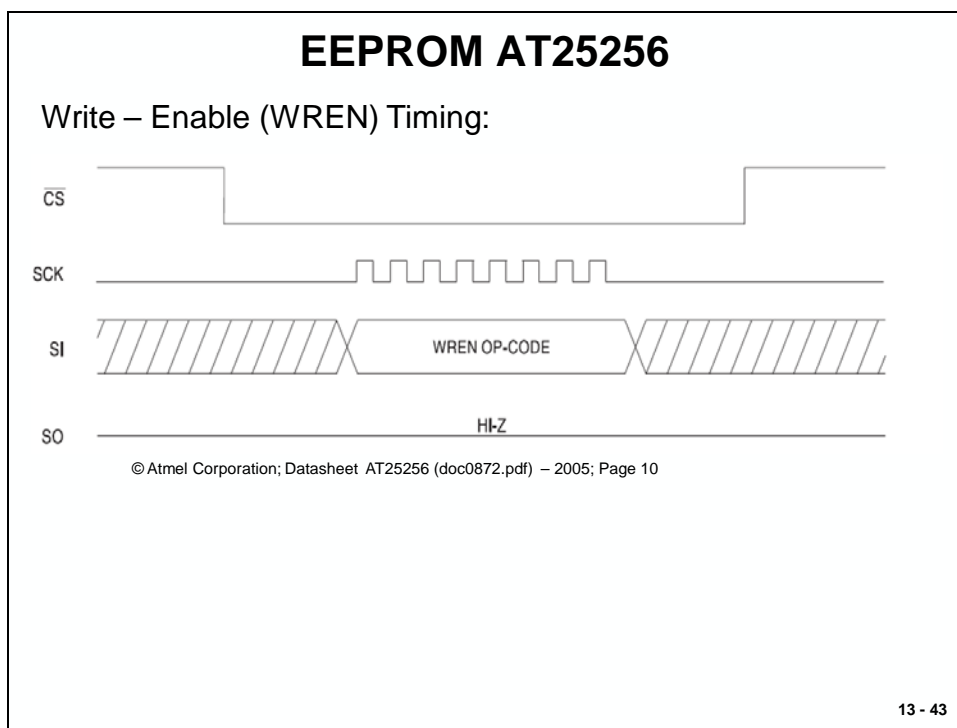
Instruction Register

The AT25128/256 utilizes an 8-bit instruction register. The list of instructions and their operation codes are contained in the next slide. All instructions, addresses and data are transferred with the MSB first and start with a high-to-low CS transition.

<div style="border:1px solid black; padding:1em;">

EEPROM AT25256

Instruction Register:

Instruction	Description	Code
WREN	Write Enable	0000 0110
WRDI	Write Disable	0000 0100
RDSR	Read Status Register	0000 0101
WDSR	Write Status Register	0000 0001
READ	Read Data	0000 0011
WRITE	Write Data	0000 0010

13 - 42

</div>

Before we can start our Lab procedure, we need to discuss these instructions in little more detail.

The **"Write Enable (WREN)"** command must be applied to the EEPROM to open the Write Enable Latch (WEN) **prior to each** WRITE and WDSR instruction. The command is an 8-clock SPI- sequence, as shown on the next slide (Slide 13-43):

EEPROM AT25256

Write – Enable (WREN) Timing:

© Atmel Corporation; Datasheet AT25256 (doc0872.pdf) – 2005; Page 10

13 - 43

The **"RDSR"** instruction allows the Status Register to be read. The Status Register may be read any time. It is recommended to use this instruction to check the "Write In Progress" (/RDY) bit **before** sending a new instruction to the EEPROM. This is also possible to read the Status Register continuously.

EEPROM AT25256

Read Status Register (RDSR) Timing:

© Atmel Corporation; Datasheet AT25256 (doc0872.pdf) – 2005; Page 10

13 - 44
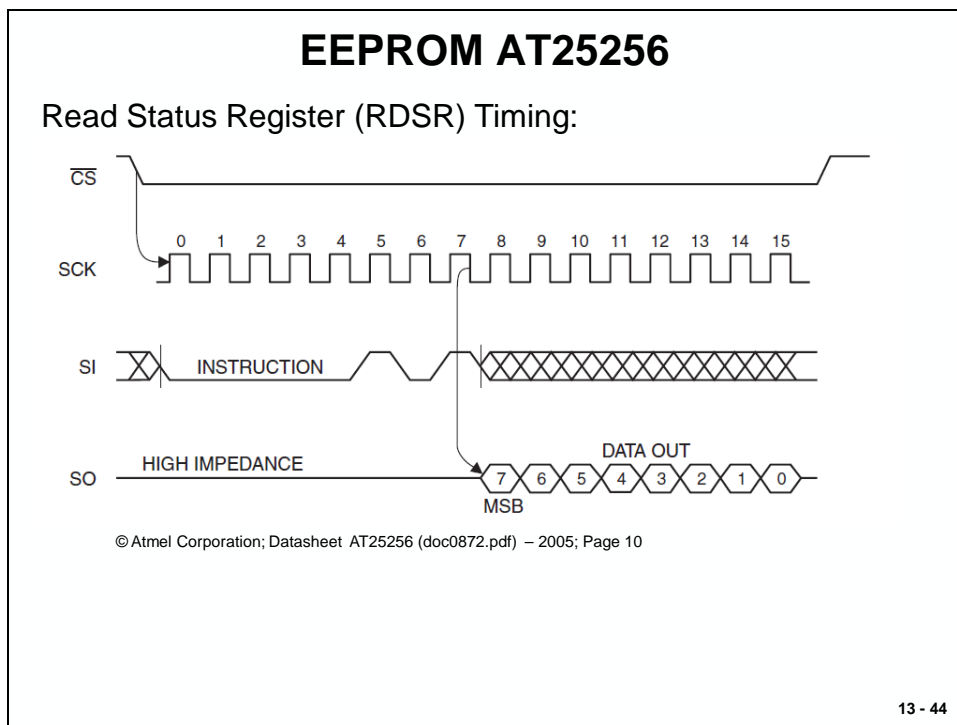
The **"READ"** instruction is used to read data out of the EEPROM. The address range of the AT25256 is from 0 to 0x7FFF. After the first 8 - bit (instruction code), the address is transmitted as a 16 - bit address. As shown in Slide 13-45, the cycle is finished with the de-activation of the /CS signal (high). However, if /CS stays active (low) and the master applies another 8 clock pulses, an internal address counter is incremented with each READ instruction.
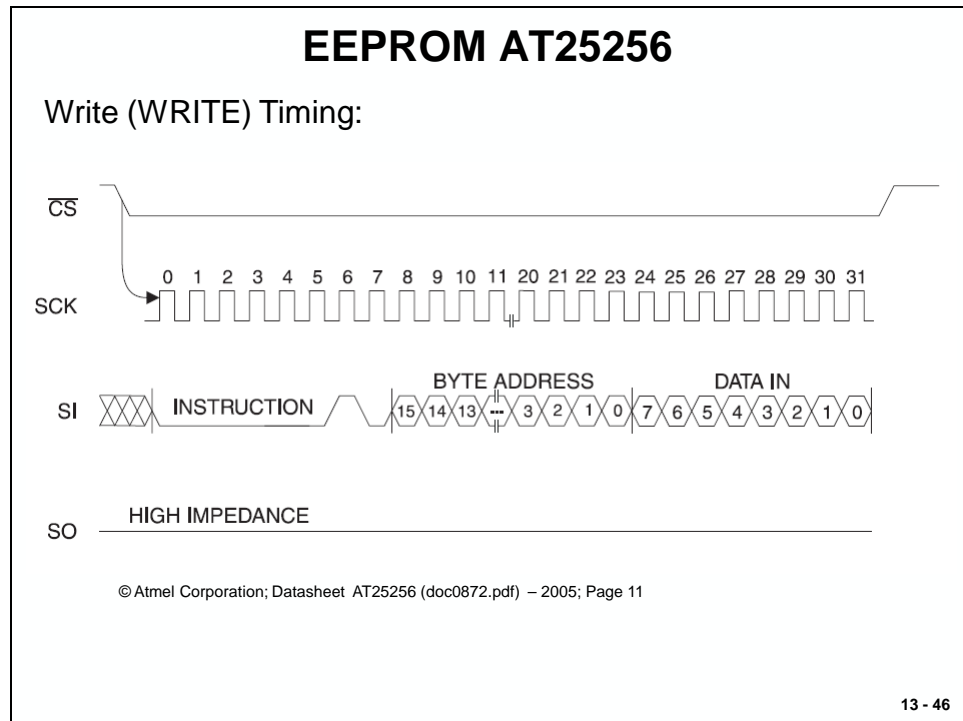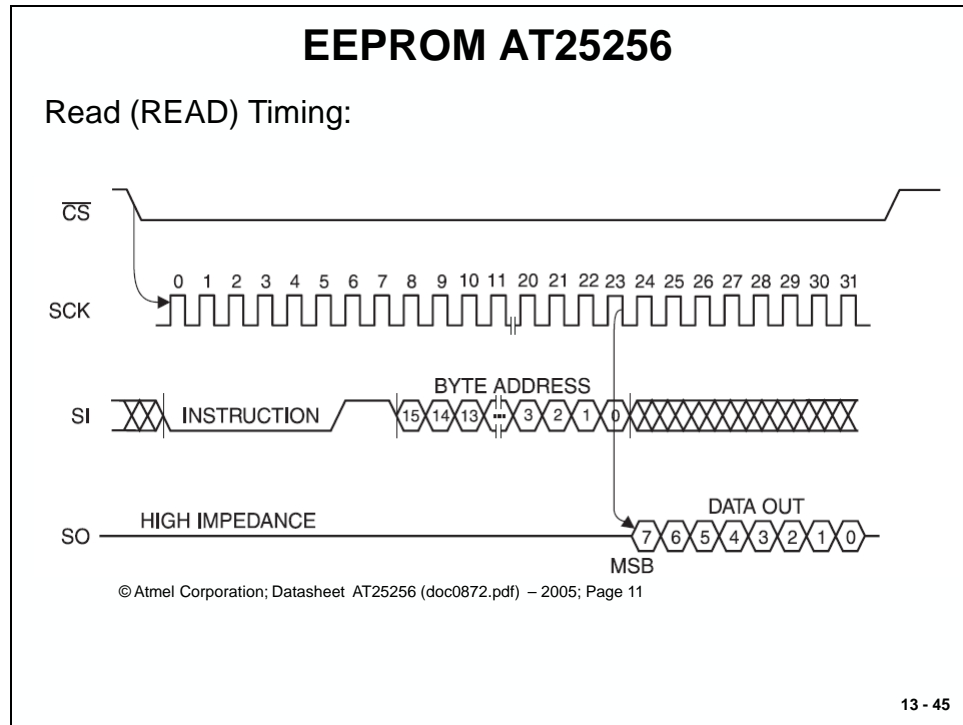
The **"WRITE"** instruction is used to write data into the EEPROM. The instruction is terminated by a rising edge at signal chip select (/CS) high. At this point the internal self - timed write cycle actually starts, at the end of which the "Write In Progress"(/RDY) bit of the Status Register is reset to 0.

Procedure

Open Project, Modify Source File

1. If not still open from Lab13_3, re-open the project Lab13.pjt in C:\DSP2833x\Labs.

2. Open the file "Lab13_1.c" and save it as "Lab13_4.c"

3. Remove the file "Lab13_3.c" from the project and add "Lab13_4.c" to it. Note: optionally you can also keep "Lab13_3.c" in the project, but exclude it from build. Use a right mouse click on file "Lab13_3.c", select "File Specific Options"; in category "General" enable "Exclude from Build".

4. At the beginning of file "Lab13_4.c" we can simplify our coding by adding some useful macros:

   ```
   #define WRITE_BUTTON      GpioDataRegs.GPADAT.bit.GPIO17    // PB1
   #define READ_BUTTON       GpioDataRegs.GPBDAT.bit.GPIO48    // PB2
   #define CS_EEPROM  GpioDataRegs.GPADAT.bit.GPIO11    //CS - EEPROM
   // Instruction Register Definitions for EEPROM
   #define WREN 0x06   // Write Enable
   #define WRDI 0x04   // Write Disable
   #define RDSR 0x05   // Read Status Register
   #define WRSR 0x01   // Write Status Register
   #define READ 0x03   // Read Command
   #define WRITE 0x02   // Write Command
   ```

5. Remove all lines (Prototypes, Function calls and Function definitions) for functions "AIC23_init()" , "McBSP_A_TX_isr()" and "SPIA_Init()". We do not need these functions in this lab. Also remove the global variables "volume", fsin" and "sine_table", including the DATA_SECTION statement for "sine_table".

6. Next, change the interrupt enable lines. For the IER register, enable the INT1 line only. Remove the PIE - interrupt enable line for McBSP-A (register PIEIER6). The only active interrupt in lab 13_4 is the CPU - Timer 0 interrupt service.

7. Inspect and change function "Gpio_select()". Delete the setup for GPIO16, 18, 19, 20, 21, 22, 23, 58 and 59 as SPI-A or McBSP-A signals. Instead, initialize GPIO24, 25 and 26 to their McBSP-B - function. Set the direction of GPIO9 (LD1), GPIO11 (/CS-EEPROM), GPIO34 (LD3) and GPIO49 (LD4) to output. Make sure, that lines GPIO17 and GPIO48 are initialized as inputs, because they are used as push-button lines. Finally in register GPADAT, set the data level for GPIO11 to 1, because this is the passive level for the chip select line (/CS) of the EEPROM.

8. Rename the function "McBSPA_Init()" to "McBSPB_Init()" and change the code of this function. The best way is to remove all old lines and code new lines. Please keep the "EALLOW" - statement at the beginning and the "EDIS" - statement at the end of the function code:

- Register SPCR1:
  - Enable clock-stop mode (CLKSTP) with ½ cycle clock delay
  - Enable receiver (RRST)

- Register SPRC2:
  - Set Free-Run (FREE) in a break event
  - Enable transmitter (XRST)
  - Enable sample rate generator (GRST)
  - Release frame logic from reset (FRST)

- Register PCR:
  - Valid transmit data on rising edge of clock (CLKXP)
  - Receive data sampled on rising edge of clock (CLKRP)
  - McBSP is master in SPI (CLKXM)
  - McBSP-clock derived from LSPCLK (SCLKME)
  - Transmit frame sync generated internally (FSXM)
  - Transmit frame sync pulses are active low(FSXP)

- Register SRGR1:
  - Set CLKG frequency to 1 MBit/s. (CLKGDV)

- Register SRGR2:
  - Select LSPCLK as input clock source (CLKSM)
  - Generate frame sync when DXR is copied into XSR (FSGM)

- Register XCR1:
  - Select 1 word per frame (XFRLEN1)
  - Select 8 bit per word (XWDLEN1)

- Register XCR2:
  - Select Single Phase Transmit (XPHASE)
  - No Data delay between sync and first data bit (XDATDLY)

- Register RCR1:
  - Select 1 word per frame (RFRLEN1)
  - Select 8 bit per word (RWDLEN1)

- Register RCR2:
  - Select Single Phase Receive (RPHASE)
  - No Data delay between sync and first data bit (RDATDLY)

Next we have to prepare some access functions to the EEPROM. Recall that the access to this device is controlled by a sequence of serial commands (see Slide 13-42).

10. Write a function **"McBSP_B_EEPROM_Read_Status()"**. As the name indicates, we will use this function to read the current value from the EEPROM status register (see Slide 13-41). The bit "/RDY" is important. If a previous command has not completed (/RDY =1), we cannot apply another one to the EEPROM. Here is the required function code:

```
int McBSP_B_EEPROM_Read_Status(void)
{
        unsigned int k;
        CS_EEPROM  = 0;                         // activate /CS of EEPROM
        McbspbRegs.DXR1.all = RDSR;             // read status register command
        while (McbspbRegs.SPCR1.bit.RRDY == 0); // wait for end of SPI - cycle
        k=McbspbRegs.DRR1.all;          // dummy read to release receiver

        McbspbRegs.DXR1.all = 0;        // dummy data to drive 8 more SPICLK
        while (McbspbRegs.SPCR1.bit.RRDY == 0); // wait for end of SPI - cycle
        k=McbspbRegs.DRR1.all;          // read status , LSB is WIP
        CS_EEPROM = 1;                  // deactivate /CS of EEPROM
        return (k);
}
```

Add a function prototype at the beginning of "Lab13_4.c".

Note: in this function and in all other functions of this lab we apply "while"-loops to wait until a certain condition becomes true. This is a first and simple approach to communicate with the device. However, there is a big problem with such a technique, since the program will stall, if the condition never becomes true. So please do not use such a technique in a real application! Instead, use timeout wait - loops or interrupt service routines in such events. If you have spare laboratory time, you can try to improve your project in such a way, after you have a first running example.

11. Write a new function **"McBSP_B_EEPROM_Write_Enable()"**. Later we will use this function to enable a write instruction into the EEPROM. Such a preceding step is necessary for all write accesses. Add:

```
void McBSP_B_EEPROM_Write_Enable(void)
{
        volatile int dummy;
        CS_EEPROM = 0;                          // activate /CS of EEPROM
        McbspbRegs.DXR1.all = WREN;             // write enable command
        while (McbspbRegs.SPCR1.bit.RRDY == 0); // wait for end of cycle
        dummy = McbspbRegs.DRR1.all; // dummy read to clear RX
        CS_EEPROM = 1;                          // deactivate /CS of EEPROM
}
```

Also add a function prototype for this function at the beginning of "Lab13_4.c".

12.   Write a new function **"McBSP_B_EEPROM_Write()".** This will be the function to write a new 8-bit pattern into an EEPROM-address. Add:

```
void McBSP_B_EEPROM_Write(int address,int data)
{
        volatile int dummy;
        CS_EEPROM = 0;                          // activate /CS of EEPROM
        McbspbRegs.DXR1.all = WRITE;        // send write code
        while (McbspbRegs.SPCR2.bit.XRDY == 0); // wait for end of cycle
        McbspbRegs.DXR1.all = (address>>8);  // write upper address byte
        while (McbspbRegs.SPCR2.bit.XRDY == 0); // wait for end of cycle
        McbspbRegs.DXR1.all = address;          // write lower address byte
        while (McbspbRegs.SPCR2.bit.XRDY == 0); // wait for end of cycle
        McbspbRegs.DXR1.all = data;             // send data
        while (McbspbRegs.SPCR2.bit.XRDY == 0); // wait for end of cycle
        dummy = McbspbRegs.DRR1.all;            // dummy read to clear receiver
        while (McbspbRegs.SPCR1.bit.RRDY == 0);
        dummy = McbspbRegs.DRR1.all;
        CS_EEPROM = 1;                          // deactivate /CS of EEPROM
}
```

Add a function prototype at the beginning of "Lab13_4.c".

13.   Write a new function **"McBSP_B_EEPROM_Read()".** This will be the function to read an 8 - bit data from an EEPROM - address. Add:

```
int McBSP_B_EEPROM_Read(int address)
{
        int data;
        CS_EEPROM = 0;                          // activate /CS of EEPROM
        McbspbRegs.DXR1.all = READ;         // Read op-code
        while (McbspbRegs.SPCR2.bit.XRDY == 0);
        McbspbRegs.DXR1.all = address>>8;   // upper byte of read address
        while (McbspbRegs.SPCR2.bit.XRDY == 0);
        McbspbRegs.DXR1.all = address;      // lower byte of read address
        while (McbspbRegs.SPCR2.bit.XRDY == 0);
        McbspbRegs.DXR1.all = 0x00;         // send dummy data
        while (McbspbRegs.SPCR2.bit.XRDY == 0);
        data = McbspbRegs.DRR1.all;         // Clear receive flag
        while (McbspbRegs.SPCR1.bit.RRDY == 0);
        data = McbspbRegs.DRR1.all;         // Read data from memory
        CS_EEPROM = 1;                          // deactivate /CS of EEPROM
        return(data);
}
```

Add a function prototype at the beginning of "Lab13_4.c".

Now we are ready to add function calls to write into or to read from the EEPROM into our main-loop.

14.   In the function "main()" just before we enter the endless while(1)-loop, add code to check if bit "/RDY" of the EEPROM status - register is zero. Use the function "McBSP_B_EEPROM_Read_Status()" to read the current status byte. Continue into the while(1)-loop only if "/RDY" is zero. If it is not zero, repeat the status check.

15. In the endless while(1)-loop of "main()", just after the 100 milliseconds wait construction based on CPU-Timer 0 and the service - instructions for the watchdog, add new code to write or to read the EEPROM.

Write into EEPROM - Address 0x0040:

Recall, that we would like to execute a write access at address 0x0040, if button PB1 (GPIO17) is pushed (this line is zero, if the button is pressed down). If this is true:

- Call function "McBSP_B_EEPROM_Write_Enable()".

- Wait until EEPROM status bit "WEN" = 1. Use function "McBSP_B_EEPROM_Read_Status" to get the latest status.

- Call function "McBSP_B_EEPROM_Write()" to write data to the EEPROM. The first parameter is the internal EEPROM - Address (for example 0x0040), the second parameter is the data. For data we will use the current value of the Hex-Encoder device (GPIO15…GPIO12), which inputs a value between 0 and 15.

- Wait until EEPROM status - bit "/RDY" is zero. Use the function "McBSP_B_EEPROM_Read_Status()" to get the latest status.

- Note: At the end of this sequence you should also add some code lines to make sure that the second execution of a write sequence is only possible, after button PB1 has been released. Since the watchdog is active, you cannot simply put a wait-loop here!

Read from EEPROM - Address 0x0040:

Finally we have to add some code to read EEPROM - address 0x0040, if button PB2 (GPIO48) has been pushed (this line is zero, if the button is pressed down). If this is true:

- Call the function "McBSP_B_EEPROM_Read()" and store the return value in a new local unsigned integer variable "data_read".

- Copy bit 2 of "data_read" to LED LD4 (GPIO49), bit 1 to LED LD3(GPIO34) and bit 0 of "data_read" to LED LD1 (GPIO9).

Build, Load and Run

16. Click the "Rebuild Active Project " button or perform:

Project → Rebuild All (Alt +B)

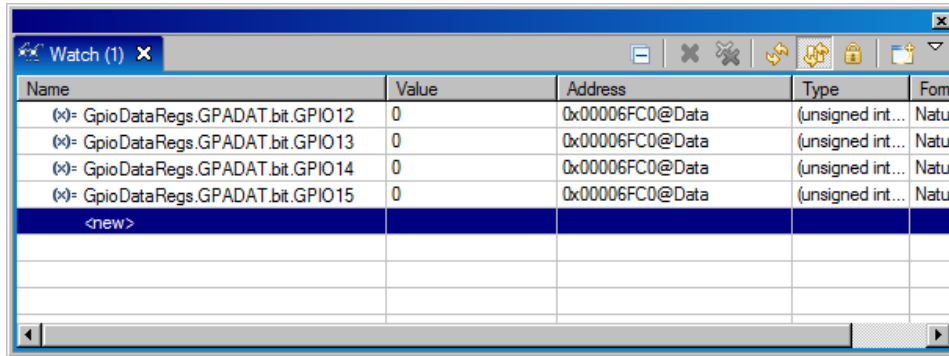17. Load the output file in the debugger session:

Target → Debug Active Project

and switch into the "Debug" perspective.

18. Verify that in the debug perspective the window of the source code "Lab13_3.c" is high-lighted and that the blue arrow for the current Program Counter position is placed under the line "void main(void)".

19. Perform a real time run.

   Scripts ➔ Realtime Emulation Control ➔ Run_Realtime_with_Restart

20. Turn the 4-Bit Hex-Encoder to a known state. Hint: Use a Watch window and monitor GPIO15…GPIO12.

21. Now press button PB1. The Encoder value is written into EEPROM - Address 0x0040.

22. Now press button PB2. The bits 2…0 of the returned value from EEPROM address 0x0040 should be displayed at LEDs LD4, LD3 and LD1. It should correspond to the status of GPIO12, GPIO13 and GPIO14 which was written into the EEPROM by the last write command.

23. Stop the Realtime Run:

   Scripts ➔ Realtime Emulation Control ➔ Full Halt

24. Close Code Composer Studio and switch off the Peripheral Explorer Board. After a few seconds re-power the board and start Code Composer Studio. Download the project into the DSP, run it and push the read button PB2 first. Now the LEDs LD4, LD3 and LD1 should display the last value that has been stored in EEPROM-address 0x0040 before the power has been switched off. (An EEPROM is non - volatile memory that retains the information when power supply has been switched off).

   END of Lab-Exercise 13_4

Optional Exercise (EEPROM and SCI):

If your Laboratory time permits, you can try to combine Lab13_4 (EEPROM) with one of the exercises from chapter 9 (SCI - module). The task is to store a whole message, which is transmitted by a host to the F2833x via RS232. If PB2 is pushed, the message, which is stored inside the F2833x, must be sent back to the host.