TMS320C6000 McBSP Initialization

Shaku Anjanaiah
Brad Cobb

Digital Signal Processing Solutions

ABSTRACT

The TMS320C6000™ multichannel buffered serial port (McBSP) can operate in a variety of modes, as per application requirements. For proper operation, the serial port must be initialized in a specific order.

This document describes the initialization steps necessary when either the EDMA or the CPU is used to service the McBSP data. Typically, the EDMA is used to perform read/write transfers from/to the McBSP. The EDMA transfers are read/write synchronized, and the McBSP provides these synchronization events. Alternatively, in cases where the CPU reads from DRR and writes to DXR, either the polled or interrupt method can be used. The sample code described in this application report can be downloaded from http://www.ti.com/lit/zip/SPRA488.

Contents

List of Figures

1 Design Problem

How do I initialize the McBSP for correct frame synchronization and data exchange? What are the important initialization steps and their order of execution?

2 McBSP Introduction

The main functional blocks of the McBSP are shown in Figure 1. They are:

- **Transmitter:** The transmitter section is responsible for the serial transmission of data that is written in DXR. The contents of DXR are copied to the transmit shift register XSR. The transfer starts as soon as the transmit frame sync (FSX) is detected. One bit of data is transmitted or shifted out of XSR on every transmit clock CLKX. New data can be written to DXR using either the CPU or the EDMA.
- **Receiver:** The data received on the DR pin is shifted into the receive shift register (RSR) on every receive clock (CLKR). Again, the actual shifting in of data begins after detection of a receive frame sync (FSR). The data in RSR is copied to a receive buffer register (RBR), and then to the data receive register (DRR). The DRR can be read by either the CPU or the EDMA.
- **Sample Rate Generator:** As the name implies, this module generates control signals such as the transmit/receive clocks and frame sync signals necessary for data transfer to and from the McBSP. Clock generation circuitry allows user to choose either the CPU clock or an external source via CLKS to generate CLK(R/X). Frame sync signal properties, such as frame period and frame width, are also programmable. FS(R/X), CLK(R/X) are bidirectional pins, and therefore, can be inputs or outputs.
- **Events/Interrupt Generation:** The McBSP generates sync events to the EDMA to indicate that data is ready in DRR, or that DXR is ready for new data. They are read sync event, REVT, and write sync event, XEVT. Similarly the CPU can read/write to the McBSP based on interrupts (RINT and XINT) generated by the McBSP.

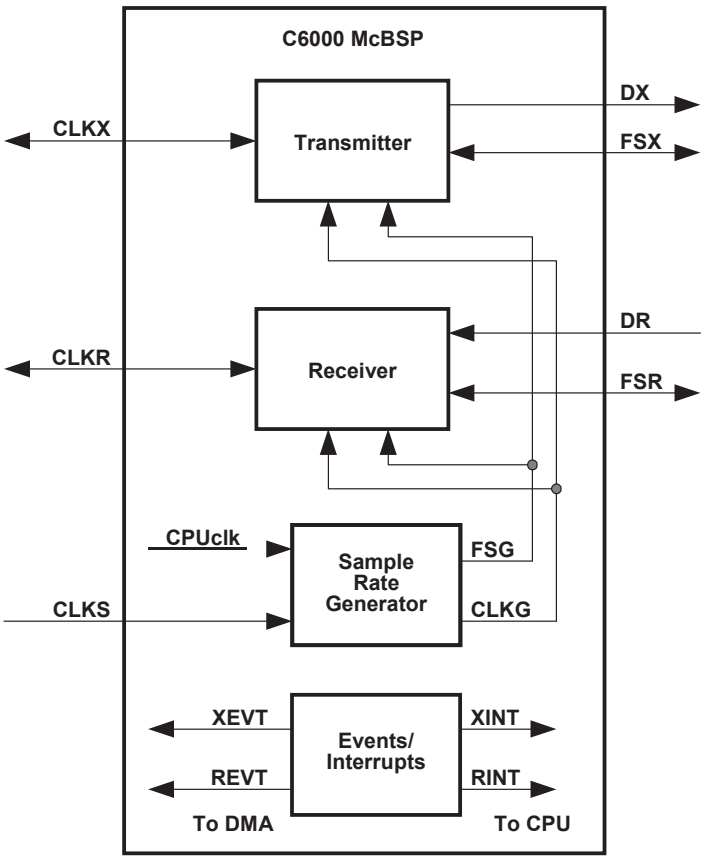

Figure 1. McBSP Functional Block Diagram

3 Servicing the McBSP

The TMS320C6000 can service the McBSP via the CPU or the EDMA. All control registers have to be programmed via the CPU, but the data registers DXR and DRR can be accessed either by the CPU or the EDMA. Typically, the EDMA channel(s) is used to read/write to the data registers, thus relieving the CPU from servicing a slow peripheral. For more details of EDMA servicing a peripheral, please refer to the application reports *TMS320C6000 DMA Applications* (SPRA529) and *TMS320C6000 Enhanced DMA: Example Applications* (SPRA636).

- Via EDMA: EDMA write accesses to the DXR use the write synchronization event, XEVT, provided by the McBSP. Similarly, an EDMA read of the DRR is synchronized by the internal REVT signal from the McBSP. Therefore, the EDMA reads from or writes to the McBSP for every serial element transfer. Once the EDMA completes the required number of element transfers, it can be programmed to generate a channel-complete interrupt to the CPU, if required.
- Via CPU: When the CPU is used to service the McBSP, it can be done in either interrupt-driven method or the polled method. Polling method ties up the CPU while waiting for data to be transmitted or received. The (R/X)RDY bits in the SPCR are polled for receive/transmit ready condition of the McBSP.

In the interrupt-driven case, the CPU performs other processing while the interrupts from the McBSP signal the CPU when it needs to be served. The default value of (R/X)INTM = 00b causes the McBSP to interrupt the CPU via (R/X)INT on every element transfer if the CPU interrupts are enabled. XINT is generated when the DXR is ready to accept new data, and RINT is generated when a new serial element has been received in the DRR. Other interrupt mode settings of (R/X)INTM are not meant for servicing the McBSP for data reads/writes, but for diagnostic and tracking purposes.

4 Initialization Requirements

Generation of control signals such as the clock, frame sync, and clock source in the McBSP is programmable. The order in which the respective modules are activated is important for correct operation of the McBSP.

Consider, for example, the case when the transmitter is the clock and frame master meaning it is responsible for generation of clocks and frames for itself and to the device (receiver) it is communicating to. The first step is to ensure that the slave (in this case the receiver) is awake (taken out of reset), and is ready to receive frame and data from the transmitter. This is followed by taking the transmitter out of reset, and then activating the frame sync generator in the transmitter. This ensures that the receiver does not lose the first frame and its data.

5 McBSP Initialization Cases

This section describes the step-by-step procedure for McBSP initialization based on CPU or EDMA data transfers. The following three methods and the initialization procedure are summarized in the flowchart shown in Figure 2.

- EDMA transfers: The following steps describe the setup of interrupts, EDMA, and the McBSP in the required order.

1. Set $\overline{\text{GRST}}$=$\overline{\text{XRST}}$=$\overline{\text{RRST}}$=$\overline{\text{FRST}}$=0. If coming out of device reset, this is not required.
2. Program the sample rate generator register (SRGR), serial port control register (SPCR), pin control register (PCR), and receive control register (RCR) to the required values.

   CAUTION:
   Do not set the bits described in Step 1 while programming these registers.
3. Take the sample rate generator out of reset by setting $\overline{\text{GRST}}$=1 in the SPCR. Internal clock CLKG is now driven by the chosen clock source and as per programmed clock divide-down.

   NOTE: If frame syncs and clocks are inputs to both transmit and receive sections of the McBSP, this step is not required.
4. Wait 2 bit clocks (CLKR/X). A simple formula to arrive at this number in terms of CPU clock cycles is:
   a. For CPU clock as source: Let P=(1/CPUclock).

      The number of CPU clocks equal to 2 data bit-clocks is:

      N = (1+CLKGDV) * 2, where min. value of CLKGDV = 1

   b. For CLKS as clock source: Let P_S=(1/CLKS frequency) and P=(1/CPUclock).

      The number of CPU clocks equal to 2 data bit-clocks is:

      N = (1+CLKGDV) * 2 * (P_S /P), where min. value of CLKGDV = 0 and CLKS is not greater than (CPUclock/2)

      In general, the following formula can be used depending on the clock source:

      N = ((1+CLKGDV) * 2 * CLKSM) + ((1+CLKGDV) * 2 * (P_S /P) * (!CLKSM))
5. Enabling Interrupts: To use interrupts, you have to set the Global Interrupt Enable (GIE) and Non-Maskable Interrupt Enable (NMIE) bits in the IER.

   For DMA, select the DMA channel you want to use. Enable CPU interrupts that correspond to the DMA channel that will be used to service the McBSP. These interrupts can be used to notify end of frame. The default mapping of DMA channel-complete interrupts to CPU is as follows:

   DMA channel 0 → CPU interrupt 8

   DMA channel 1 → CPU interrupt 9

   DMA channel 2 → CPU interrupt 11

   DMA channel 3 → CPU interrupt 12

For EDMA, enable only CPU interrupt 8, EDMA_INT. When using EDMA you must also select the appropriate EDMA channels to service the McBSP. Select from channels 12−15 for TMS320C621x™/TMS320C671x™ devices and from channels 12−15 and 17−18 for TMS320C64x™ devices.

6. For DMA transfers, put the DMA in a stop condition. Clear any previous R/WSTAT bits so that unwanted transfers do not occur.
7. EDMA initialization: Program the EDMA channels for required operation. The following would be a typical set up:

   Source address = DRR for reads OR memory location for writes.

   Destination address = memory location for reads OR DXR for writes.

   Transfer counter = number of elements to be transferred.

   Receive synchronization event, *R/WSYNC* = REVT from McBSP for reads.

   Transmit synchronization event, *R/WSYNC* = XEVT from McBSP for writes.

   DMA channel complete interrupt bit, *TCINT* = enabled

   Priority bit, *PRI* = 1; optional, but recommended.
8. Instruct the EDMA to run. For DMA, set START=01b in the DMA channel's primary control register to start the DMA without auto-initialization. For EDMA, set the bit in the EDMA event enable register that corresponds to the desired channel.
9. Take the section (transmitter/receiver) that is not a frame master (frame sync is an input) out of reset by setting $\overline{\text{XRST}}$ or $\overline{\text{RRST}}$ = 1. Now the slave is ready to accept a frame sync and start data transfer. Alternatively, a new frame sync interrupt ((R/X)INTM = 10b) can be used to wake up the transmitter/receiver.
10. Pull the frame master (transmitter or receiver) out of reset ($\overline{\text{XRST}}$ or $\overline{\text{RRST}}$ = 1).
11. If FSGM = 1 (frame sync generated by sample rate generator), enable frame sync generator by setting $\overline{\text{FRST}}$ = 1. If FSGM = 0, frames are generated on every DXR to XSR copy, and therefore, $\overline{\text{FRST}}$ is not used. In any case, the master now starts data transfer.

- **Interrupt-driven CPU transfers:** Setting (R/X)INTM=00b in the SPCR allows the McBSP to interrupt the CPU whenever data is ready in DRR, or when data can be written to DXR. The initialization steps are similar to EDMA driven transfers, except that EDMA is not used. Therefore, replace steps 5 through 8 above with the following:

1. Map the required XINT(0/1/2) and/or RINT(0/1/2) interrupts to the CPU via the interrupt multiplexer registers.
2. Enable the mapped interrupts.

   Once the McBSP is initialized (after step 11 above), each element in the transfer will cause the execution of an ISR that writes to DXR or reads from DRR.

- **Polled CPU transfers:** The transmit and receive ready (R/X)RDY bits in SPCR are polled to determine the readiness of the transmitter and receiver. Here, the CPU has to check for this condition, which might prevent it from performing any needed processing. The McBSP initialization process has some differences compared to the previous two methods of McBSP service. Since neither EDMA nor interrupt-driven transfer is applicable for this case, steps 5 through 8 are not required. The steps are, therefore, 1 to 4, 9 to 11 followed by the polling loop.

TMS320C64x is a trademark of Texas Instruments.

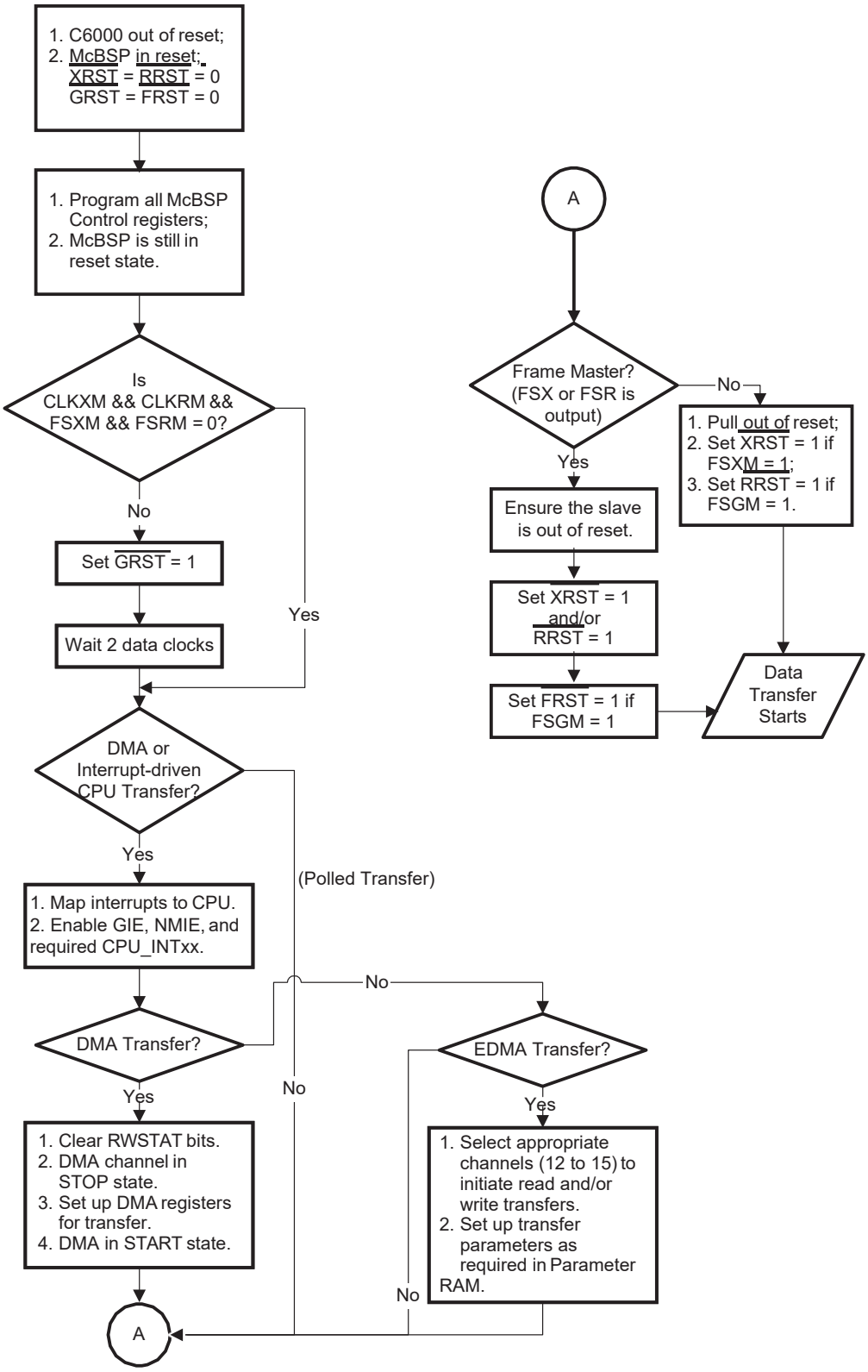

Figure 2. McBSP Initialization Flowchart

6 Using the Chip Support Library (CSL)

The chip support library makes it easier to program the McBSP's operation by the use of macros and functions that streamline the code. Below is a list of the basic CSL components that are needed to program the McBSP and EDMA for operation. To see the CSL in operation, please see the sample code in Appendix A. For a complete listing of CSL functions and their syntax, refer to the *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401).

Define the chip you wish to use

```
#define CHIP_6711
```

Include the relevant header files

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

Declare CSL objects

```
MCBSP_Handle hMcbsp
DMA_Handle hDma
EMDA_Handle hEdma
EDMA_Handle hEdma_NULL
```

Setup the IRQ vector table

```
IRQ_setVecs(vectors)
```

Initialize the CSL library

```
CSL_init()
```

Setup the McBSP configuration structure

```
MCBSP_Config mcbspCfg
```

Open the McBSP channel

```
hMcbsp = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
```

Apply the configuration structure to the channel

```
MCBSP_config(hMcbsp, &mcbspCfg)
```

Enable the sample rate generator

```
MCBSP_enableSrgr(hMcbsp)
```

Reset EDMA channels

```
DMA_reset(INV)
EDMA_clearPram(0x00000000)
```

Enable interrupts

```
IRQ_nmiEnable()
IRQ_globalEnable()
IRQ_disable(IRQ_EVT_DMAINT0)
IRQ_clear(IRQ_EVT_DMAINT0)
IRQ_enable(IRQ_EVT_DMAINT0)
```

Open the DMA or EDMA channels depending on the C6000 device

```
hDma = DMA_open(DMA_CHA2, DMA_OPEN_RESET);
hEdma = EDMA_open(EDMA_CHA_REVT0, EDMA_OPEN_RESET);
```

Configure the EDMA channels for operation

```
DMA_configArgs(...)
EDMA_configArgs(...)
```

For EDMA, configure and link to a terminating NULL parameter set

```
hEdma_NULL = EDMA_allocTable(-1);
EDMA_reset(hEdma_NULL);
EDMA_link(hEdma, hEdma_NULL); /* Link to terminating event */
```

Start the DMA or enable the EDMA

```
DMA_start(hDma)
EDMA_enableChannel(hEdma)
```

Enable the McBSP for receive or transmit

```
MCBSP_enableRcv(hMcbsp)
MCBSP_enableXmt(hMcbsp)
```

Enable the frame sync generator

```
MCBSP_enableFsync(hMcbsp)
```

Handle EDMA interrupts

```
EDMA_intTest(...)
EDMA_intClear(...)
```

Close the McBSP and EDMA channels

```
MCBSP_close(hMcbsp)
DMA_close(hDma)
EDMA_close(hEdma)
```

7 References

1. TMS320C6000 DMA Example Applications (SPRA529).
2. TMS320C6000 Enhanced DMA: Example Applications (SPRA636).
3. TMS320C6000 Chip Support Library API Reference Guide (SPRU401).

Appendix A

The following sample code illustrates the sequence of programming the McBSP control registers for initialization and data formats. Some functions in the code also show the EDMA register setup for reading/writing from/to the McBSP. Note that this is a sample code for digital loopback mode, where the transmit and receive pins in one McBSP in the C6000 are connected internally via software (DLB = 1).

```
/*-------------------------------------------------------------------------*/
/* TI Proprietary Information                                              */
/* 08/03/01                                                                */
/* mcbsp-init.c:                                                           */
/*                                                                         */
/* McBSP0 is used in DLB mode with the (E)DMA to service the McBSP.        */
/* CLKX and FSX generated using CPU clock and SRG.                         */
/*-------------------------------------------------------------------------*/

/* Chip definition, change this accordingly */
#define CHIP_6711 /* Enter chip */

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

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

#define XFER_SIZE 128 /* Number of elements to transfer */

#define BUFFER_SIZE 256
/* Global variables used in interrupt ISRs */
volatile int recv0_done = FALSE;
volatile int xmit0_done = FALSE;
```

```
/*---------------------------------------------------------------------------*/
/* Declare CSL objects */

MCBSP_Handle hMcbsp0;       /* Handles for McBSP */

#if (DMA_SUPPORT)
DMA_Handle hDma1;           /* Handles for DMA */
DMA_Handle hDma2;
#endif

#if (EDMA_SUPPORT)          /* Handles for EDMA */
EDMA_Handle hEdma1;
EDMA_Handle hEdma2;
EDMA_Handle hEdmadummy;
#endif

/*---------------------------------------------------------------------------*/
/* External functions and function prototypes */

void init_mcbsp0(void);
void set_interrupts_dma(void);
void set_interrupts_edma(void);

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

/*---------------------------------------------------------------------------*/
/* main()                                                                    */
/*---------------------------------------------------------------------------*/

void
main(void)
{
/* Declaration of local variables */
Uint32 xfer_size;
Uint32 wait = 0;
Uint32 y;  /* Counter to initialize buffers */
```

```
/* Create buffers. */
#if (DMA_SUPPORT)
static Uint32 dmaInbuff[BUFFER_SIZE];   /* buffer for DMA supporting devices */
static Uint32 dmaOutbuff[BUFFER_SIZE];
#endif

#if (EDMA_SUPPORT)
static Uint32 edmaInbuff[BUFFER_SIZE];  /* buffer for EDMA supporting devices */
static Uint32 edmaOutbuff[BUFFER_SIZE];
#endif

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

xfer_size = XFER_SIZE;  /* set the size of the transfer */

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

/* initialize the the McBSP */
init_mcbsp0();

/* Enable sample rate generator - GRST=1 */

MCBSP_enableSrgr(hMcbsp0);
for (wait=0; wait<0x10; wait++);  /* Wait states after SRG starts */

#if (DMA_SUPPORT)       /* for DMA supporting devices */
DMA_reset(INV);         /* reset all DMA channels */
#endif

#if (EDMA_SUPPORT)           /* for EDMA supporting devices */
EDMA_clearPram(0x00000000); /* Clear PaRAM RAM of the EDMA */
set_interrupts_edma();
#endif
```

```
/*-------------------------------------------------------------------------*/
/* DMA channels 1 and 2 config structures                                  */
/*-------------------------------------------------------------------------*/
#if (DMA_SUPPORT)       /* for DMA supporting devices */

for (y=0;y<xfer_size;y++) {
     dmaOutbuff[y]=0x0000000+y;  /* Initialize the transmit buffer */
     dmaInbuff[y]=0;      /* Initialize the receive buffer */
     }

  /* Channel 2 transmits the data */
  hDma2 = DMA_open(DMA_CHA2, DMA_OPEN_RESET);  /* Handle to DMA channel 2 */
DMA_configArgs(hDma2,
 DMA_PRICTL_RMK(
  DMA_PRICTL_DSTRLD_DEFAULT,
  DMA_PRICTL_SRCRLD_DEFAULT,
  DMA_PRICTL_EMOD_DEFAULT,
  DMA_PRICTL_FS_DEFAULT,
  DMA_PRICTL_TCINT_ENABLE, /* TCINT =1 */
  DMA_PRICTL_PRI_DMA,      /* DMA high priority */
  DMA_PRICTL_WSYNC_XEVT0,  /* Set synchronization event XEVT0=01100 */
  DMA_PRICTL_RSYNC_DEFAULT,
  DMA_PRICTL_INDEX_DEFAULT,
  DMA_PRICTL_CNTRLD_DEFAULT,
  DMA_PRICTL_SPLIT_DEFAULT,
  DMA_PRICTL_ESIZE_32BIT,  /* Element size 32 bits */
  DMA_PRICTL_DSTDIR_DEFAULT,
  DMA_PRICTL_SRCDIR_INC,   /* Increment source by element size */
  DMA_PRICTL_START_DEFAULT
  ),
 DMA_SECCTL_RMK(
 #if (!CHIP_6201)
        DMA_SECCTL_WSPOL_NA, /* only on C6202(B)/C6203(B)/C6204/C6205 */
        DMA_SECCTL_RSPOL_NA, /* only on C6202(B)/C6203(B)/C6204/C6205 */
        DMA_SECCTL_FSIG_NA, /* only on C6202(B)/C6203(B)/C6204/C6205 */
        #endif
        DMA_SECCTL_DMACEN_DEFAULT,
        DMA_SECCTL_WSYNCCLR_DEFAULT,
        DMA_SECCTL_WSYNCSTAT_DEFAULT,
        DMA_SECCTL_RSYNCCLR_DEFAULT,
```

```
        DMA_SECCTL_RSYNCSTAT_DEFAULT,
        DMA_SECCTL_WDROPIE_DEFAULT,
        DMA_SECCTL_WDROPCOND_DEFAULT,
        DMA_SECCTL_RDROPIE_DEFAULT,
        DMA_SECCTL_RDROPCOND_DEFAULT,
        DMA_SECCTL_BLOCKIE_ENABLE, /* Enables DMA channel interrupt */
        DMA_SECCTL_BLOCKCOND_DEFAULT,
        DMA_SECCTL_LASTIE_DEFAULT,
        DMA_SECCTL_LASTCOND_DEFAULT,
        DMA_SECCTL_FRAMEIE_DEFAULT,
        DMA_SECCTL_FRAMECOND_DEFAULT,
        DMA_SECCTL_SXIE_DEFAULT,
        DMA_SECCTL_SXCOND_DEFAULT
        ),
 DMA_SRC_RMK((Uint32)dmaOutbuff), /* Set source to dmaOutbuff */
 DMA_DST_RMK(MCBSP_ADDRH(hMcbsp0, DXR)), /* Set destination to DXR   */
 DMA_XFRCNT_RMK(
  DMA_XFRCNT_FRMCNT_DEFAULT,
  DMA_XFRCNT_ELECNT_OF(xfer_size)
  )
);
  /* Channel 1 receives the data */
  hDma1 = DMA_open(DMA_CHA1, DMA_OPEN_RESET);  /* Handle to DMA channel 1    */
DMA_configArgs(hDma1,
 DMA_PRICTL_RMK(
  DMA_PRICTL_DSTRLD_DEFAULT,
  DMA_PRICTL_SRCRLD_DEFAULT,
  DMA_PRICTL_EMOD_DEFAULT,
  DMA_PRICTL_FS_DEFAULT,
  DMA_PRICTL_TCINT_ENABLE, /* TCINT =1 */
  DMA_PRICTL_PRI_DMA,      /* DMA high priority */
  DMA_PRICTL_WSYNC_DEFAULT,
  DMA_PRICTL_RSYNC_REVT0,  /* Set synchronization event REVT0=01101 */
  DMA_PRICTL_INDEX_DEFAULT,
  DMA_PRICTL_CNTRLD_DEFAULT,
  DMA_PRICTL_SPLIT_DEFAULT,
  DMA_PRICTL_ESIZE_32BIT,  /* Element size 32 bits */
  DMA_PRICTL_DSTDIR_INC,   /* Increment destination by element size */
  DMA_PRICTL_SRCDIR_DEFAULT,
  DMA_PRICTL_START_DEFAULT
```

```
  ),
 DMA_SECCTL_RMK(
 #if (!CHIP_6201)
        DMA_SECCTL_WSPOL_NA, /* only on C6202(B)/C6203(B)/C6204/C6205 */
        DMA_SECCTL_RSPOL_NA, /* only on C6202(B)/C6203(B)/C6204/C6205 */
        DMA_SECCTL_FSIG_NA, /* only on C6202(B)/C6203(B)/C6204/C6205 */
        #endif
        DMA_SECCTL_DMACEN_DEFAULT,
        DMA_SECCTL_WSYNCCLR_DEFAULT,
        DMA_SECCTL_WSYNCSTAT_DEFAULT,
        DMA_SECCTL_RSYNCCLR_DEFAULT,
        DMA_SECCTL_RSYNCSTAT_DEFAULT,
        DMA_SECCTL_WDROPIE_DEFAULT,
        DMA_SECCTL_WDROPCOND_DEFAULT,
        DMA_SECCTL_RDROPIE_DEFAULT,
        DMA_SECCTL_RDROPCOND_DEFAULT,
        DMA_SECCTL_BLOCKIE_ENABLE, /* Enables DMA channel interrupt */
        DMA_SECCTL_BLOCKCOND_DEFAULT,
        DMA_SECCTL_LASTIE_DEFAULT,
        DMA_SECCTL_LASTCOND_DEFAULT,
        DMA_SECCTL_FRAMEIE_DEFAULT,
        DMA_SECCTL_FRAMECOND_DEFAULT,
        DMA_SECCTL_SXIE_DEFAULT,
        DMA_SECCTL_SXCOND_DEFAULT
        ),
 DMA_SRC_RMK(MCBSP_ADDRH(hMcbsp0, DRR)), /* Set source to DRR */
 DMA_DST_RMK((Uint32)dmaInbuff),  /* Set destination to dmaInbuff */
 DMA_XFRCNT_RMK(
 DMA_XFRCNT_FRMCNT_DEFAULT,
 DMA_XFRCNT_ELECNT_OF(xfer_size)
 )
);

set_interrupts_dma(); /* initialize the interrupts */

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

#endif  /* end for dma supporting devices */
```

```
/*-------------------------------------------------------------------------*/
/* EDMA channels 12 and 13 config structures                               */
/*-------------------------------------------------------------------------*/

#if (EDMA_SUPPORT)      /* for EDMA supporting devices  */

for (y=0;y<xfer_size;y++) {   /* Initialize the Outbuff */
     edmaOutbuff[y]=0x00000000+y;
     edmaInbuff[y]=0;
     }

hEdma1 = EDMA_open(EDMA_CHA_REVT0, EDMA_OPEN_RESET);

EDMA_configArgs(hEdma1,
 EDMA_OPT_RMK(
  EDMA_OPT_PRI_HIGH,       /* High priority EDMA */
  EDMA_OPT_ESIZE_32BIT,    /* Element size 32 bits */
  EDMA_OPT_2DS_NO,
  EDMA_OPT_SUM_NONE,
  EDMA_OPT_2DD_NO,
  EDMA_OPT_DUM_INC,        /* Destination increment by element size */
  EDMA_OPT_TCINT_YES,      /* Enable Transfer Complete Interrupt */
  EDMA_OPT_TCC_OF(13),     /* TCCINT = 0xD, REVT0 */
  #if (C64_SUPPORT)
  EDMA_OPT_TCCM_DEFAULT,
  EDMA_OPT_ATCINT_DEFAULT,
  EDMA_OPT_ATCC_DEFAULT,
  EDMA_OPT_PDTS_DEFAULT,
  EDMA_OPT_PDTD_DEFAULT,
  #endif
  EDMA_OPT_LINK_YES,       /* Enable linking to NULL table*/
  EDMA_OPT_FS_NO
  ),
  EDMA_SRC_RMK(MCBSP_ADDRH(hMcbsp0, DRR)), /* Set source to DRR */
  EDMA_CNT_RMK(0,xfer_size),
  EDMA_DST_RMK((Uint32)edmaInbuff), /* Set destination to edmaInbuff */
  EDMA_IDX_RMK(0,0),
  EDMA_RLD_RMK(0,0)
  );
```

```
hEdma2 = EDMA_open(EDMA_CHA_XEVT0, EDMA_OPEN_RESET);

EDMA_configArgs(hEdma2,

 EDMA_OPT_RMK(
  EDMA_OPT_PRI_HIGH,       /* High priority EDMA */
  EDMA_OPT_ESIZE_32BIT,    /* Element size 32 bits */
  EDMA_OPT_2DS_NO,
  EDMA_OPT_SUM_INC,        /* Source increment by element size */
  EDMA_OPT_2DD_NO,
  EDMA_OPT_DUM_NONE,
  EDMA_OPT_TCINT_YES,      /* Enable Transfer Complete Interrupt */
  EDMA_OPT_TCC_OF(12),     /* TCCINT = 0xC, XEVT0 */
  #if (C64_SUPPORT)
  EDMA_OPT_TCCM_DEFAULT,
  EDMA_OPT_ATCINT_DEFAULT,
  EDMA_OPT_ATCC_DEFAULT,
  EDMA_OPT_PDTS_DEFAULT,
  EDMA_OPT_PDTD_DEFAULT,
  #endif
  EDMA_OPT_LINK_YES,       /* Enable linking to NULL table*/
  EDMA_OPT_FS_NO
  ),
 EDMA_SRC_RMK((Uint32)edmaOutbuff), /* Set source to edmaOutbuff */
 EDMA_CNT_RMK(0,xfer_size),
 EDMA_DST_RMK(MCBSP_ADDRH(hMcbsp0, DXR)), /* Set destination to DXR0 */
 EDMA_IDX_RMK(0,0),
 EDMA_RLD_RMK(0,0)
       );

  hEdmadummy = EDMA_allocTable(-1); /* Dynamically allocates PaRAM RAM table */
  EDMA_configArgs(hEdmadummy, /* Dummy or Terminating Table in PaRAM    */
         0x00000000,  /*  Terminate  EDMA  transfers  by  linking  to  */
         0x00000000,          /* this NULL table                        */
         0x00000000,
         0x00000000,
         0x00000000,
         0x00000000
         );
```

```
  EDMA_link(hEdma1, hEdmadummy); /* Link terminating event to the EDMA event */
  EDMA_link(hEdma2, hEdmadummy);

  EDMA_enableChannel(hEdma1);    /* Enable EDMA channels */
  EDMA_enableChannel(hEdma2);

#endif  /* End for EDMA supporting devices */

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

MCBSP_enableFsync(hMcbsp0);

/* Wait for interrupt to the CPU when DMA transfer/receive is done */
  while (!xmit0_done || !recv0_done);

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

#if (DMA_SUPPORT)        /* close DMA channels */
DMA_close(hDma1);
DMA_close(hDma2);
#endif

#if (EDMA_SUPPPORT)
EDMA_close(hEdma1);      /* close EDMA channels */
EDMA_close(hEdma2);
EDMA_freeTable(hEdmadummy);
#endif

} /* end main */

/*-------------------------------------------------------------------------*/
/* init_mcbsp0()                                                           */
/*-------------------------------------------------------------------------*/
/* MCBSP Config structure */
/* Setup the MCBSP_0 as a master */
void
init_mcbsp0(void)
{
MCBSP_Config mcbspCfg0 = {
```

```
#if (EDMA_SUPPORT)
MCBSP_SPCR_RMK(
 MCBSP_SPCR_FREE_DEFAULT,
 MCBSP_SPCR_SOFT_DEFAULT,
 MCBSP_SPCR_FRST_YES,  /* Frame sync not reset */
 MCBSP_SPCR_GRST_DEFAULT,
 MCBSP_SPCR_XINTM_XRDY, /* XINT driven by XRDY */
 MCBSP_SPCR_XSYNCERR_DEFAULT,
 MCBSP_SPCR_XRST_DEFAULT,
 MCBSP_SPCR_DLB_ON,    /* Digital Loopback Mode enabled */
 MCBSP_SPCR_RJUST_RZF,  /* Right-justify and zero-fill MSBs in DRR */
 MCBSP_SPCR_CLKSTP_DEFAULT,
 MCBSP_SPCR_DXENA_OFF,
 MCBSP_SPCR_RINTM_RRDY,  /* RINT driven by RRDY */
 MCBSP_SPCR_RSYNCERR_DEFAULT,
 MCBSP_SPCR_RRST_DEFAULT
 ),
#endif
#if (DMA_SUPPORT)
   MCBSP_SPCR_RMK(
 MCBSP_SPCR_FRST_NO,
 MCBSP_SPCR_GRST_DEFAULT,
 MCBSP_SPCR_XINTM_XRDY,  /* XINT driven by XRDY */
 MCBSP_SPCR_XSYNCERR_DEFAULT,
 MCBSP_SPCR_XRST_DEFAULT,
 MCBSP_SPCR_DLB_ON,    /* Digital Loopback Mode enabled */
 MCBSP_SPCR_RJUST_RZF,  /* Right-justify and zero-fill MSBs in DRR */
 MCBSP_SPCR_CLKSTP_DEFAULT,
 MCBSP_SPCR_RINTM_RRDY,  /* RINT driven by RRDY */
 MCBSP_SPCR_RSYNCERR_DEFAULT,
 MCBSP_SPCR_RRST_DEFAULT
          ),
#endif
#if (EDMA_SUPPORT)
    MCBSP_RCR_RMK(
 MCBSP_RCR_RPHASE_SINGLE, /* Single phase frame */
 MCBSP_RCR_RFRLEN2_DEFAULT,
 MCBSP_RCR_RWDLEN2_DEFAULT,
 MCBSP_RCR_RCOMPAND_MSB,  /* No companding */
 MCBSP_RCR_RFIG_YES,   /* Ignore unexpected sync pulses */
```

```
 MCBSP_RCR_RDATDLY_1BIT,  /* 1-bit delay */
 MCBSP_RCR_RFRLEN1_OF(0), /* 1 word per phase */
 MCBSP_RCR_RWDLEN1_32BIT, /* 32-bit receive element length */
 MCBSP_RCR_RWDREVRS_DISABLE
 ),
#endif
#if (DMA_SUPPORT)
    MCBSP_RCR_RMK(
 MCBSP_RCR_RPHASE_SINGLE, /* Single phase frame */
 MCBSP_RCR_RFRLEN2_DEFAULT,
 MCBSP_RCR_RWDLEN2_DEFAULT,
 MCBSP_RCR_RCOMPAND_MSB,  /* No companding */
 MCBSP_RCR_RFIG_YES,   /* Ignore unexpected sync pulses */
 MCBSP_RCR_RDATDLY_1BIT,  /* 1-bit delay */
 MCBSP_RCR_RFRLEN1_OF(0), /* 1 word per phase */
 MCBSP_RCR_RWDLEN1_32BIT  /* 32-bit receive element length */
 ),
#endif
#if (EDMA_SUPPORT)
    MCBSP_XCR_RMK(
 MCBSP_XCR_XPHASE_SINGLE, /* Single phase frame */
 MCBSP_XCR_XFRLEN2_DEFAULT,
 MCBSP_XCR_XWDLEN2_DEFAULT,
 MCBSP_XCR_XCOMPAND_MSB,  /* No companding */
 MCBSP_XCR_XFIG_YES,   /* Ignore unexpected sync pulses */
 MCBSP_XCR_XDATDLY_1BIT,  /* 1-bit delay */
 MCBSP_XCR_XFRLEN1_OF(0), /* 1 word per phase */
 MCBSP_XCR_XWDLEN1_32BIT, /* 32-bit receive element length */
 MCBSP_XCR_XWDREVRS_DISABLE
 ),
#endif
#if (DMA_SUPPORT)
    MCBSP_XCR_RMK(
 MCBSP_XCR_XPHASE_SINGLE, /* Single phase frame */
 MCBSP_XCR_XFRLEN2_DEFAULT,
 MCBSP_XCR_XWDLEN2_DEFAULT,
 MCBSP_XCR_XCOMPAND_MSB,  /* No companding */
 MCBSP_XCR_XFIG_YES,  /* Ignore unexpected sync pulses */
 MCBSP_XCR_XDATDLY_1BIT,  /* 1-bit delay */
 MCBSP_XCR_XFRLEN1_OF(0), /* 1 word per phase */
 MCBSP_XCR_XWDLEN1_32BIT  /* 32-bit receive element length */
```

```
 ),
#endif
     MCBSP_SRGR_RMK(
 MCBSP_SRGR_GSYNC_DEFAULT,
 MCBSP_SRGR_CLKSP_DEFAULT,
 MCBSP_SRGR_CLKSM_INTERNAL, /* srg clock derived internally */
  MCBSP_SRGR_FSGM_DXR2XSR,
 MCBSP_SRGR_FPER_DEFAULT,
 MCBSP_SRGR_FWID_DEFAULT,
 MCBSP_SRGR_CLKGDV_OF(7)
 ),
MCBSP_MCR_DEFAULT,
MCBSP_RCER_DEFAULT,
MCBSP_XCER_DEFAULT,
     MCBSP_PCR_RMK(
 MCBSP_PCR_XIOEN_DEFAULT,
 MCBSP_PCR_RIOEN_DEFAULT,
 MCBSP_PCR_FSXM_INTERNAL, /* Internal frame sync signals used */
 MCBSP_PCR_FSRM_INTERNAL, /* Internal frame sync signals used */
 MCBSP_PCR_CLKXM_OUTPUT,
 MCBSP_PCR_CLKRM_OUTPUT,
 MCBSP_PCR_CLKSSTAT_DEFAULT,
 MCBSP_PCR_DXSTAT_DEFAULT,
 MCBSP_PCR_FSXP_ACTIVEHIGH,
 MCBSP_PCR_FSRP_ACTIVEHIGH,
 MCBSP_PCR_CLKXP_RISING,
 MCBSP_PCR_CLKRP_FALLING
 )
};

hMcbsp0 = MCBSP_open(MCBSP_DEV0, MCBSP_OPEN_RESET);
MCBSP_config(hMcbsp0, &mcbspCfg0);
}

/*-------------------------------------------------------------------------*/
/* set_interrupts_dma()                                                    */
/*-------------------------------------------------------------------------*/
#if (DMA_SUPPORT)
void                            /* Set the interrupts */
set_interrupts_dma(void)        /* if the device supports DMA */
```

```
{
 IRQ_nmiEnable();
 IRQ_globalEnable();
 IRQ_disable(IRQ_EVT_DMAINT2);
 IRQ_disable(IRQ_EVT_DMAINT1);   /* INT11 and INT9 */
 IRQ_clear(IRQ_EVT_DMAINT2);
 IRQ_clear(IRQ_EVT_DMAINT1);
 IRQ_enable(IRQ_EVT_DMAINT2);
 IRQ_enable(IRQ_EVT_DMAINT1);
 return;
}
#endif

/*-------------------------------------------------------------------------------*/
/* set_interrupts_edma()                                                         */
/*-------------------------------------------------------------------------------*/
#if (EDMA_SUPPORT)
void                          /* Set the interrupts */
set_interrupts_edma(void)     /* if the device supports EDMA */
{
 IRQ_nmiEnable();
 IRQ_globalEnable();
 IRQ_reset(IRQ_EVT_EDMAINT);
 IRQ_disable(IRQ_EVT_EDMAINT);
 EDMA_intDisable(12);         /* ch 12 for McBSP transmit event XEVT0 */
 EDMA_intDisable(13);         /* ch 13 for McBSP receive event REVT0 */
 IRQ_clear(IRQ_EVT_EDMAINT);
 EDMA_intClear(12);
 EDMA_intClear(13);
 IRQ_enable(IRQ_EVT_EDMAINT);
 EDMA_intEnable(12);
 EDMA_intEnable(13);

 return;
}
#endif
```

```
/*-------------------------------------------------------------------------------*/
/*    DMA DATA TRANSFER COMPLETION ISRs                                          */
/*-------------------------------------------------------------------------------*/
interrupt void     /* vecs.asm hooks this up to IRQ 11 */
c_int11(void)      /* DMA ch2 */
{
#if (DMA_SUPPORT)
xmit0_done = TRUE;
return;
#endif
}

interrupt void       /* vecs.asm hooks this up to IRQ 09 */
c_int09(void)        /* DMA ch1 */
{
#if (DMA_SUPPORT)
recv0_done = TRUE;
return;
#endif
}

interrupt void     /* vecs.asm hooks this up to IRQ 08 */
c_int08(void)      /* for the EDMA */
{
 #if (EDMA_SUPPORT)
  if (EDMA_intTest(12))
  {
  xmit0_done = TRUE;
  EDMA_intClear(12); /* clear CIPR bit so future interrupts can be recognized */
  }

  else if (EDMA_intTest(13))
  {
  recv0_done = TRUE;
  EDMA_intClear(13); /* clear CIPR bit so future interrupts can be recognized */
  }
#endif
return;
}
/*----------------------End of mcbsp-init.c-----------------------------------*/
```

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265