A DSP/BIOS EDMA McASP Device Driver for TMS320C6x1x DSPs

Software Development Systems

ABSTRACT

This document describes the usage and design of the generic TMS320C6x1x EDMA multichannel audio serial port (McASP) device driver. This device driver is written in conformance to the DSP/BIOS IOM device driver model and handles communication to and from the McASP, and uses the EDMA to transfer the data. It can be used as a general-purpose stand-alone mini-driver to access a serial port, or alongside a codec-specific mini-driver.

Features:

- Multi-instance (can handle multiple serial McASP peripherals simultaneously).
- Supports use of cache.
- Keeps external frame sync.
- Supports any EDMA sample size (8, 16 or 32 bits).
- Designed for (but not limited to) use with codec drivers.

Contents

List of Figures

List of Tables

1 Usage

The device driver described here is part of an IOM mini-driver. That is, it is implemented as the lower layer of a 2-layer device driver model. The upper layer is called the class driver and can be either the DSP/BIOS GIO, SIO/DIO, or PIP/PIO modules. The class driver provides an independent and generic set of APIs and services for a wide variety of mini-drivers and allows the application to use a common interface for I/O requests. Figure 1 shows the overall DSP/BIOS device driver architecture. For more information about the IOM device driver model as well as the GIO, SIO/DIO, and PIP/PIO modules, see the *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

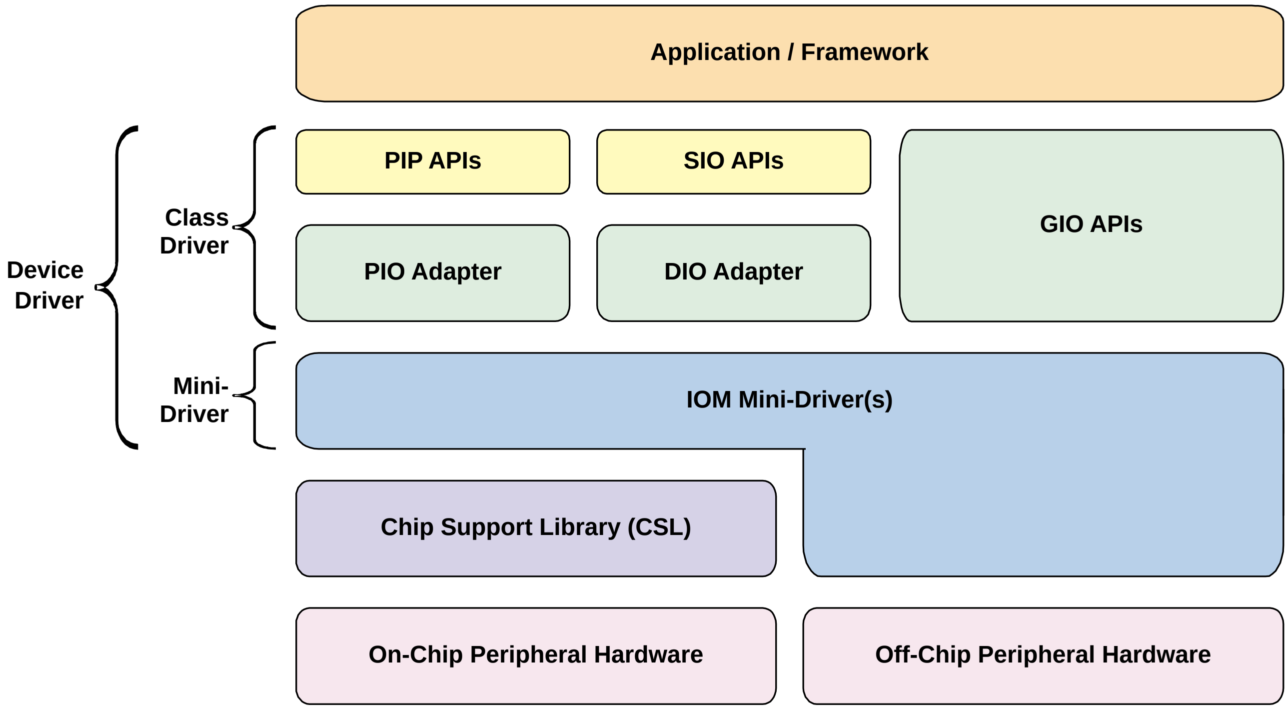

Figure 1. DSP/BIOS IOM Device Driver Model

This device driver can be used as a general-purpose stand-alone mini-driver to interface with the McASP peripheral on TMS320C6x1x chips using the EDMA. However, this device driver can also be used in conjunction with a codec-specific portion of the mini-driver to handle its data processing. In that case, the codec-specific part only has to set up the codec and pass the required parameters to this generic part of the mini-driver. Figure 2 shows the data flow between the components in a system in which the mini-driver is split into a generic part and a codec-specific part.

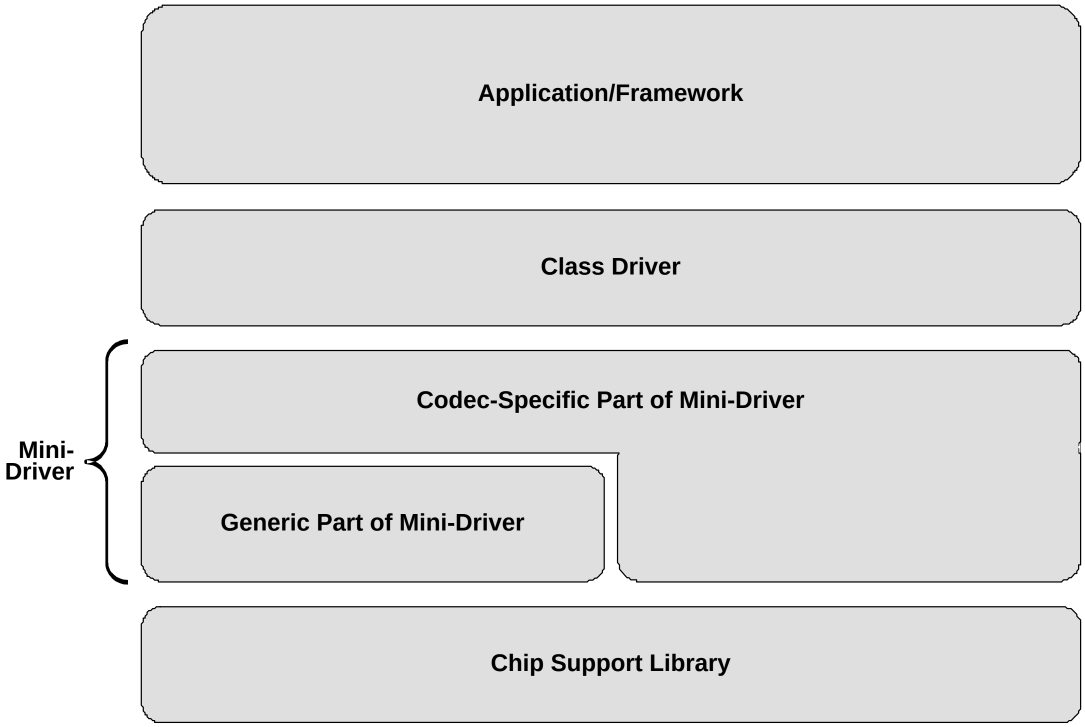

Figure 2. Codec Device Driver Partitioning

1.1 Configuration

To use this driver with a codec-specific portion to the mini-driver, the properties should be set up for the codec device driver. In that case, the codec-specific part will set up this generic device driver. Refer to the individual codec-specific device driver documentation on how to set up the properties in this case.

To use this device driver as a stand-alone mini-driver, a device entry for every McASP port instance must be added to the DSP/BIOS configuration tool. Below is a description of the properties of this device entry when using this device driver as a stand-alone mini-driver.

- **Init function table:** Type _C6X1X_EDMA_MCASP_init.
- **Function table ptr:** Type _C6X1X_EDMA_MCASP_FXNS.
- **Function table type:** Select IOM_Fxns.
- **Device ID:** Type 0 for McASP0 or 1 for McASP1. This device driver supports McASP0 and McASP1 without modifications to the source.
- **Device params ptr:** A pointer to your instance of the device parameter structure (described below). This device driver has no default parameters, which means this property cannot be left to 0x0.
- **Device global data ptr:** This property must be set to 0x0.

The channel parameters (described below) are passed to the device driver during run-time when creating the device communications channel. Refer to the *DSP/BIOS Device Driver Developer's Guide* (SPRU616) for more information on how to pass channel parameters when using this device driver with SIO.

This device driver uses the supplied McASP and EDMA CSL configuration structures (see the parameter sections below) to set up the McASP and EDMA respectively.

1.2 Device Parameters

```
typedef struct C6X1X_EDMA_MCASP_DevParams {
      Int versionId;
      Bool cacheCalls;
      Uns enableClkg;
      Uns enableHclkg;
      Uns enableFsyncg;
      Int irqId;
      MCASP_Config *mcaspCfgPtr;
      Int inEvtIrqId;
      Int outEvtIrqId;
      C6X1X_EDMA_MCASP_EvtCallback *evtCallback;
      Uns inEvtIntrMask;
      Uns outEvtIntrMask;
      Uns edmaIntrMask;
} C6X1X_EDMA_MCASP_DevParams;
```

- **versionId:** Version number of the driver.
- **cacheCalls:** If this parameter is set to TRUE the device driver will treat buffers issued to any IOM channel associated with this device (port) as if they are in cacheable memory and the L2 data cache is enabled.
- **enableClkg:** If this parameter is set to C6X1X_EDMA_MCASP_XMT, C6X1X_EDMA_MCASP_RCV, C6X1X_EDMA_MCASP_RCVXMT or C6X1X_EDMA_MCASP_XMTRCV, the device driver will enable the internal clock generator when the McASP is started, which can be set independently for the transmit section and receive section.
  - C6X1X_EDMA_MCASP_XMT: transmitter only
  - C6X1X_EDMA_MCASP_RCV: receiver only
  - C6X1X_EDMA_MCASP_RCVXMT or C6X1X_EDMA_MCASP_XMTRCV: Both transmitter and receiver
- **enableHClkg:** If this parameter is set to C6X1X_EDMA_MCASP_XMT, C6X1X_EDMA_MCASP_RCV, C6X1X_EDMA_MCASP_RCVXMT or C6X1X_EDMA_MCASP_XMTRCV, the device driver will enable the internal high-frequency clock generator when the McASP is started, which can be set independently for the transmit section and receive section.
- **enableFsyncg:** If this parameter is set to C6X1X_EDMA_MCASP_XMT, C6X1X_EDMA_MCASP_RCV, C6X1X_EDMA_MCASP_RCVXMT or

C6X1X_EDMA_MCASP_XMTRCV, the device driver will enable the internal frame sync generator when the McASP is started, which can be set independently for the transmit section and receive section.

- **irqId:** This parameter selects which IRQ number to use for the EDMA interrupt. The system default is 8.
- **mcaspCfgPtr:** This parameter is a pointer to a CSL configuration structure which will be passed to MCASP_config() for the McASP port.
- **inEvtIrqId:** This parameter selects which IRQ number to use for the McASP Receive Event interrupt. If inEvtIrqId < 0, then the driver will use the driver's default value of 5.
- **outEvtIrqId:** This parameter selects which IRQ number to use for the McASP Transmit Event interrupt. If outEvtIrqId < 0, then the driver will use the driver's default value of 6.
- **evtCallback:** This parameter is a pointer to a event callback structure. If NULL, no event is registered.
- **InIntrMask:** Interrupt mask, set in the input ISR.
- **outIntrMask:** Interrupt mask, set in the output ISR.
- **edmaIntrMask:** Interrupt mask, set in the edma ISR.

1.3 Event Interrupt Processing

This driver implements processing of special McASP event interrupts, such as those raised by Receive Buffer Overrun/ Transmit Buffer Underrun conditions. Users can register those events through configuration. If an event interrupt is asserted, this code will examine the source of the interrupt and then get information about the event and the device id. The event callback is then called so that this information can be exported back to the application and acted on in an implementation specific manner. A data structure specific to event interrupt processing is defined in the driver header file c6x1x_edma_mcasp.h and is implemented as follows:

```
typedef struct C6X1X_EDMA_MCASP_EvtCallback {
      C6X1X_EDMA_MCASP_TevtCallback evtFxn;
      Uns evtMask;
} C6X1X_EDMA_MCASP_EvtCallback;
```

- **evtFxn:** This parameter is a pointer to the user defined callback function.
- **evtMask:** This parameter is ored events to be registered

  Possible event constant:

  ```
  C6X1X_EDMA_MCASP_EVT_ROVRN
  C6X1X_EDMA_MCASP_EVT_RSYNCERR
  C6X1X_EDMA_MCASP_EVT_RCKFAIL
  C6X1X_EDMA_MCASP_EVT_RDMAERR

  C6X1X_EDMA_MCASP_EVT_XUNDRN
  C6X1X_EDMA_MCASP_EVT_XSYNCERR
  ```

```
C6X1X_EDMA_MCASP_EVT_XCKFAIL
C6X1X_EDMA_MCASP_EVT_XDMAERR
```

1.4 Channel Parameters

```
typedef struct C6X1X_EDMA_MCASP_ChanParams {
      Uns tdmChans;
      Uns edmaChan;
      EDMA_Config *edmaCfgPtr;
} C6X1X_EDMA_MCASP_ChanParams;
```

- **tdmChans:** This parameter should be set to the number of TDM channels the McASP is using for this IOM channel (e.g., 1 for mono, 2 for stereo etc.). This value will be used by the driver to maintain the frame sync. Refer to the section “External frame sync” below for details.
- **edmaChan:** This parameter selects which EDMA channel number to map for the McASP transmit for output or the McASP receive for input.
- **edmaCfgPtr:** A pointer to a CSL configuration structure to be passed to EDMA_config() for the EDMA channel used by this IOM channel.

1.5 Control Commands

This device driver has no run-time control commands.

2 Architecture

This section describes the design and implementation of the device driver. The driver uses various DSP/BIOS and CSL modules (see Appendix A). Refer to the *TMS320C6000 DSP/BIOS Application Programming Interface* (SPRU403) and *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401). The technical details of the EDMA are available from *TMS320C6000 Peripherals Reference Guide* (SPRU190). The technical details of the McASP are available from *TMS320C6000 Multichannel Audio Serial Port (McASP) Reference Guide* (SPRU041).

2.1 Data Structures

This driver uses two internal data structures, a port object and a channel object, to maintain its state during execution. This device driver is multi-instance, which means it can handle several McASP ports running simultaneously. Each McASP used needs a port object instance associated with it to maintain its state. In turn, every port has two associated channel object instances (one for input and one for output) that hold the IOM channel states during execution. The contents of these structures are described below.

2.1.1 The Port Object

```
/* Number of IOM channels per port (must be 2, one for input and one for
output) */
#define NUMCHANS 2
/* Structure containing port specific variables */
typedef struct PortObj {
    Uns inUse;
    Int devid;
    Bool cacheCalls;
    Uint32 enableHclkg;
    Uint32 enableClkg;
    Uint32 enableFsyncg;
    MCASP_Handle hMcasp;
    ChanObj chans[NUMCHANS];
    Uns chanCreated;
    C6X1X_EDMA_MCASP_TevtCallback evtCallback;
    Uns evtMask;
} PortObj, *PortHandle;
```

- **inUse:** This variable is set when this port is configured so that it can fail if another attempt to configure the port is made.
- **devid:** The device driver stores the devid sent to it during configuration in order to know which port it is during execution (e.g., McASP0 or McASP1).
- **cacheCalls:** The device parameter cacheCalls passed to the device driver during configuration is stored here during execution. The parameter is explained under “Device parameters” above.
- **enableHclkg:** The device parameter enableHclkg passed to the device driver during configuration is stored here during execution used when McASP startup. The parameter is explained under “Device parameters” above.
- **enableClkg:** The device parameter enableClkg passed to the device driver during configuration is stored here during used when McASP startup. The parameter is explained under “Device parameters” above.
- **enableFsyncg:** The device parameter enableFsyncg passed to the device driver during configuration is stored here during used when McASP startup. The parameter is explained under “Device parameters” above.
- **hMcasp:** This variable holds the CSL handle returned by the CSL function MCASP_open() for this port and is used to access the McASP during execution.
- **chans:** An array holding the channel objects associated with this port.
- **chanCreated:** This variable is set when a channel in this port has been created so that the driver can sync the start of internal clock, high frequency clock, frame sync for transmit channel and receive channel.

- **evtCallback:** The callback evtFxn which the device parameter evtCallback passed to the device driver during configuration is stored here.
- **evtMask:** The callback evtMask which the device parameter evtCallback passed to the device driver during configuration is stored here.

2.1.2 The Channel Object

```
/* Maximum number of EDMA jobs linked at a time (must be 2) */
#define MAXLINKCNT 2
/* Structure containing IOM channel specific variables */
typedef struct ChanObj {
    Uns inUse;
    Int mode;
    struct PortObj *port;
    EDMA_Handle xferPram;
    EDMA_Handle pramTbl[MAXLINKCNT];
    EDMA_Handle prevPramPtr;
    EDMA_Handle loophEdma;
    IOM_Packet *flushPacket;
    IOM_Packet *abortPacket;
    IOM_Packet *packetList[MAXLINKCNT];
    QUE_Obj packetQueue;
    Int submitCount;
    Int writeIndex;
    Int readIndex;
    Int tcc;
    IOM_TiomCallback cbFxn;
    Ptr cbArg;
} ChanObj, *ChanHandle;
```

- **inUse:** This variable is set when this IOM channel is configured so that it can fail if another attempt to configure the IOM channel is made.
- **mode:** When a channel is created, its mode is specified (i.e., input or output). This variable holds this mode, but uses internally specified values instead of the specified IOM mode (IOM_INPUT or IOM_OUTPUT). The reason why internal variables are used is that it needs to use the mode as an index, but cannot use the IOM modes since they are bit masks.
- **port:** This variable holds a pointer to the port object which owns this channel.
- **xferPram:** This CSL EDMA handle is returned when opening the EDMA channel with the CSL function EDMA_open(). It holds the currently executing EDMA job during execution.
- **pramTbl:** This array holds EDMA parameter RAM used for linking and double buffering.

- **prevPramPtr:** This parameter RAM is used to hold the previous EDMA job. Its use is described below under “Data flow”.
- **loophEdma:** This EDMA parameter RAM holds the Loop EDMA job. The use of the Loop job is described in the section “External frame sync” below.
- **flushPacket:** Since this device driver uses an asynchronous flush command, this is where the flush packet sent to the IOM channel is stored when such a command has been issued.
- **abortPacket**: Since this device driver uses an asynchronous abort command, this is where the abort packet sent to the IOM channel is stored when such a command has been issued.
- **packetList**: An array holding the IOM packets which are linked in the EDMA channel.
- **packetQueue:** A software queue holding issued IOM packets which are issued but not linked in the EDMA channel.
- **submitCount:** This variable holds the number of packets submitted (issued) to the channel.
- **writeIndex:** Used by the mdSubmitChan() function to keep track of which space in the packetList and which EDMA parameter RAM from pramTbl it should use for an issued packet.
- **readIndex:** Used by the ISRs to keep track of which IOM packet has been completed.
- **tcc:** This variable holds the EDMA Transfer Complete Code used by this channel.
- **cbFxn;** The callback function specified when creating the IOM channel is stored here. It is used to send an IOM packet to the upper layers.
- **cbArg:** The callback argument specified when creating the channel is stored here. It is used in conjunction with the callback function to send an IOM packet to the upper layers.

2.2 Data Flow

This section describes how a buffer is processed and passes through this device driver. When an IOM packet is issued to an IOM channel, it is first checked to see which command has been issued. This driver supports the commands IOM_READ, IOM_WRITE, IOM_ABORT and IOM_FLUSH. When the term “link” is used below, it refers to the EDMA’s ability to link EDMA jobs to each other, see *TMS320C6000 Peripherals Reference Guide* (SPRU190).

2.2.1 The IOM Read and Write Commands

The device driver handles read and write similarly. The mode of the IOM channel to which the packet was issued decides whether it is a read or a write command, not the IOM packet command field.

First there is a check to see if there is space available to link a new EDMA job in the corresponding EDMA channel (A maximum of 2 at a time, see “Constraints” below). If not, the IOM packet is put on a queue (packetQueue) until there is space available for the job. If there is space available, the packet is put on the packetList and available parameter RAM for the job is allocated. Depending on whether this is an input or an output channel, the destination or the source field is set with the packet’s address field respectively. If cacheCalls is set to TRUE, the cache is flushed or cleaned for the buffer. This is described in depth in the section “Cache coherency” below.

This driver supports any element (sample) size the EDMA supports (i.e., 8, 16 or 32 bits). The IOM packet uses nmadus (number of minimal addressable data units) for its size field, which is 8 bits on a TMS320C6xxx. When setting the count (CNT) field in the EDMA parameter RAM, the device driver reads the ESIZE parameter in the OPT field of the EDMA configuration used by this IOM channel to calculate the number of samples a packet is (which is what the CNT field needs).

The driver then links this EDMA parameter space to the Loop EDMA job. Normally an EDMA job would link to a NULL parameter set, but we use the Loop EDMA job to maintain the frame sync. This is described in detail in the section "External frame sync" below.

The device driver then links this new job to the currently executing job. Looking at Figure 3 below, this means that the currently executing EDMA job, which was B before, now becomes A, and the new EDMA job becomes the new B job. While linking these EDMA jobs, we disable the EDMA channel to make sure the current job doesn't complete before the linking is done.

When this setup is done, the EDMA will start the new EDMA job corresponding to the issued IOM packet (input or output depending on the IOM channel) when the currently executing job terminates and invokes the job it links to, which is the new job.

When an EDMA job completes, an EDMA interrupt will occur. This driver uses the EDMA dispatcher from the CSL to execute the ISR corresponding to the TCC that was asserted. The use of the EDMA dispatcher allows separate ISRs for input and output (isrInput and isrOutput), but since the driver is multi-instance, the device driver still has to check which job was completed. Even though the device driver has separate ISRs for input and output, some common code is put in the function isrCommon to save code space.

The ISR fetches the completed input or output EDMA job's corresponding packet from the packetList in the channel object, marks it as completed and then calls the callback function on this packet to send it to the upper layers. When the callback is done, the ISR checks to see if there are any issued packets that haven't been linked in. If there is, the device driver gets them from the packetQueue in the channel object and links them in the same way as if they were an issued new job.

2.2.2 The IOM Abort Command

When an abort command is issued, the device driver checks if there are any buffers currently issued to the IOM channel. If there is not, the device driver returns IOM_COMPLETED synchronously. If there are buffers issued to the IOM channel, the device driver saves the issued abort packet in abortPacket. It then links the currently executing job to the Loop job to make sure only one more "real" EDMA job completes. After this, the device driver returns IOM_PENDING.

This means that the driver does an asynchronous abort when there are buffers issued to the IOM channel. The reason why the device driver cannot abort the currently executing EDMA job and return all packages as aborted is that the device driver might lose the frame sync if this is done (see "External frame sync" below).

When an ISR is called and an abort packet has been issued to the corresponding IOM channel (abortPacket is non-NULL), the device driver calls the callback on all issued packets (both from packetList and packetQueue) with the packets' status field set to IOM_ABORTED to notify the above layers that their data is not to be trusted. It then resets the IOM channel state (readIndex, writeIndex and submitCount) before calling the callback on the abort packet itself (with status IOM_COMPLETED) and finally sets abortPacket to NULL.

2.2.3 The IOM Flush Command

A flush command is handled similarly to an abort command. This device driver treats a flush of an IOM input channel exactly as if abort was called on the channel. Flushing an IOM output channel is also very similar to an abort command in that it's asynchronous if packets have been issued to the IOM channel, and synchronous if not. However, in the flush command case, the device driver does not change the linking of the EDMA when flush is called on an IOM output channel. Nor does it discard the previously issued IOM Packets queued by the device driver. It merely sets the flushPacket to be the submitted flush packet and let's the currently issued packets complete. When the ISRs execute they will check to see if the last job has been completed (submitCount is 0) and if there is a flush packet (flushPacket is non-NULL). If so, the device driver calls the callback on the flush packet with status set to IOM_COMPLETED.

2.3 External Frame Sync

This section describes the problem with an external frame sync, and how this driver deals with this problem.

2.3.1 The External Frame Sync Problem

This problem occurs when the McASP is externally clocked and receives its clock from, for example, a codec. Figure 3 shows the samples the DSP is receiving from a stereo codec that generates the frame sync. Every left-right pair is a frame. A breakpoint occurs after a left sample, which means the DSP is halted. When the DSP resumes execution, it expects a right sample, but since the codec has continued to send samples while the DSP was halted, there is only a 50% chance (in the stereo case) that this will be the case. If, as in Figure 3, another left sample is received instead of a right sample, the channels will be switched. This means that if you were listening to music from a stereo codec when the halt occurred, there is a 50% chance that you will be hearing the music meant for your left ear in your right ear and vice versa. Note that the risk of failure increases with the number of McASP TDM channels used.

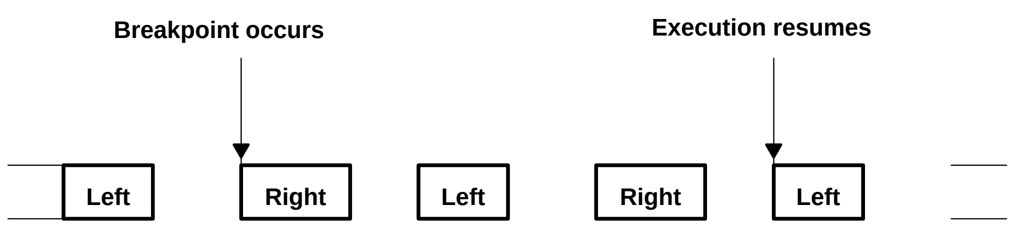

Figure 3. The Frame Sync Problem

The problem occurs in the transmit case as well, since the stereo codec will continue receiving while the DSP is halted, and it's only 50% chance that the DSP will give the codec a sample from the channel the codec expects to receive it from when execution is resumed.

2.3.2 This Driver’s Solution to the External Frame Sync Problem

Figure 4 shows this drivers solution to the frame sync problem. It shows two normal data EDMA jobs (A and B), where A is linked to B. Normally B would be linked to a null job, which would terminate the EDMA channel, but instead we link B to a Loop job. For transmit, this Loop job sends zeros to the McASP, and for receive the Loop job receives data into a buffer, a buffer which is never processed. The number of elements the Loop job transmits is given as a channel parameter (tdmChans), and depends on how many TDM slots the McASP is using. For the stereo codec mentioned above, this value would be two. The Loop job does not generate a TCC, which means it will not generate an EDMA interrupt upon completion.

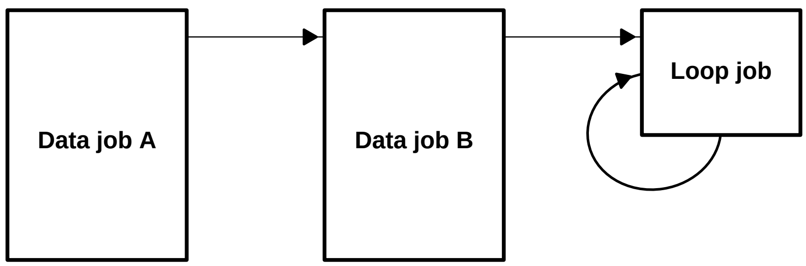

Figure 4. The Loop Job Solution to the External Frame Sync Problem

When a breakpoint occurs while A is running, the McASP will continue to transfer data. When A is done, B will continue since A is linked to B. When B is done, the Loop job starts. Since the Loop job is linked to itself, it will run continuously until another EDMA job is linked in. When execution is resumed, one of the ISR’s will be executed and will notice that the Loop job is running and that more than one buffer has been issued (submitCount > 1). This can only happen if an emulation halt occurred while A was executing. It will link the currently running EDMA job (which is the Loop job) to A. This means that two buffers (A and B) that were issued before the breakpoint occurred are reused. This is done because the application doesn’t know that a breakpoint has occurred, and this way we help preserve double buffering for the application (if used). Note that in order for this reuse of buffers to work, the device driver not only links the currently executing job to the new job when a new IOM packet is issued to an IOM channel, it also links the corresponding parameter space (saved in prevPramPtr) to the new job in order for the device driver to “remember” its links.

However, if a breakpoint occurs when B is being executed, we have no way of telling whether a breakpoint has occurred, or if the channel is being starved from the above layers. The situation where the ISR finds that the Loop job is running and one (not two or more buffers as above) buffer is currently issued to the channel could happen for two reasons. It could either mean that B was terminated successfully and there is nothing more to send, since the EDMA linking feature will have the Loop job running when the ISR is called for B. It could also mean that an emulation halt has occurred while B was transmitting, and the execution is now resumed. The driver treats this situation as if the channel has been starved, and does not reuse B (but calls the callback on the IOM packet as usual).

When a communication channel is created (input or output) for a device, an EDMA channel is set up and started with the Loop job. This means that after an IOM channel is created, there is data going to or from the McASP even before any buffers are issued to the channel. When the first IOM packet is issued it’s buffer will be linked to the currently executing job (the Loop job), and start as soon as the Loop job is finished.

By always forcing the EDMA transfer a multiple of tdmChans elements of data, the frame synchronization with the codec will be preserved through execution.

2.4 Cache Coherency

If the buffers that are submitted to the driver are in cacheable memory (typically SDRAM), and the L2 data cache is enabled, the driver takes care of cleaning and flushing the cache accordingly. This has to be done, since the EDMA accesses external memory directly through the EMIF, while the CPU goes through the cache when accessing the data. To enable the driver's cache coherency code, set the cacheCalls device parameter to TRUE. The device driver will then flush output buffers and clean the input buffers from cache when the buffers are submitted.

Both flushing the cache for the input buffers and cleaning the cache for the output buffers are done when the IOM packet (with the buffer) is issued. This is natural for output buffers, but one might think that cleaning the cache for input buffers should be done in the ISR upon completion. This is done when the IOM input packet is issued to reduce the overhead of the ISR and because the clean of a TMS320C6x cache also implies a write-back of dirty data. Since the application gives control of the packet to the device driver when issuing an IOM packet, this should not be a problem.

If buffers are placed in external memory for use with this device driver they should be aligned to a 128 bytes boundary. In addition the buffers should be of a size multiple of 128 bytes as well for the cache to work optimally.

3 Constraints

This device driver doesn't link more than two EDMA jobs at a time (double buffering). For applications that burst small buffers at a high bit rate this can be a problem, since the latency of linking in a new EDMA job after an EDMA job has completed can be too high. This should be very rare though. Note that the application can issue more than two IOM packets to any IOM channel at a time. The device driver will store them in a queue until space is available.

4 References

All these documents are available from http://www.ti.com.

1. *DSP/BIOS Device Driver Developer's Guide* (SPRU616)
2. *TMS320C6000 DSP Multichannel Audio Serial Port (McASP) Reference Guide* (SPRU041)
3. *TMS320C6000 DSP/BIOS Application Programming Interface* (SPRU403)
4. *TMS320C6000 Chip Support Library API Reference Guide* (SPRU401)
5. *TMS320C6000 Peripherals Reference Guide* (SPRU190)

Appendix A Device Driver Data Sheet

A.1 Device Driver Library Name

c6x1x_edma_mcasp.l67 for TMS320C671x DSPs.

A.2 DSP/BIOS Modules Used

- HWI – Hardware Interrupt Manager
- QUE – Queue Manager
- IOM – I/O Manager
- ATM – Atomic Manager

A.3 DSP/BIOS Objects Used

- QUE_Obj

A.4 CSL Modules Used

- McASP module
- EDMA module
- IRQ module
- CACHE module

A.5 CPU Interrupts Used

- EDMA interrupt

A.6 Peripherals Used

- McASP
- EDMA
- EMIF

A.7 Interrupt Disable Time

Maximum time that hardware interrupts can be disabled by the driver: 353 cycles. This measurement is taken using the compiler option –O3.

A.8 Memory Usage

Table A–1. Device Driver Memory Usage

	Uninitialized memory	Initialized memory
CODE	—	10848 (8-bit bytes)
DATA	448 (8-bit bytes)	88 (8-bit bytes)

NOTE: This data was gathered using the sectti command utility.
Uninitialized data: .bss
Initialized data: .cinit + .const
Initialized code: .text + .text:init

IMPORTANT NOTICE

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265