Application Report
SPRAAB8 – September 2005

Building IOM-Compliant Drivers from the DCP Tool

R. Stephen Preissig *Technical Training Organization*

ABSTRACT

The input/output mini-driver (IOM) is the standard driver model for Texas Instruments' DSP/BIOS™ operating system. A driver built on this model can be easily and efficiently tied to the software interrupt (SWI) and task (TSK) threads that are used in the DSP/BIOS scheduler via the BIOS transport mechanisms of stream input/output (SIO) and pipe (PIP) modules. This application report shows how to adapt the large number of data converter drivers available in TI's data converter plug-in (DCP) tool to conform to the IOM standard. An intermediate layer of software is developed to interface the drivers produced by the DCP tool to the class drivers defined by DSP/BIOS. Though developed around the AIC23 DCP driver for use on the DSK6416, this interface layer is written generically so that it may be adapted with minimal effort to any DCP-generated driver.

This application report contains project code that can be downloaded from this link. http://www-s.ti.com/sc/psheets/spraab8/spraab8.zip

Contents

List of Figures

Trademarks

DSP/BIOS, Code Composer Studio are trademarks of Texas Instruments.

1 The Input/Output Mini-driver Model

The input/output mini-driver (IOM) model of DSP/BIOS was created to provide a standard I/O interface for use with DSP/BIOS systems. A key component of this driver model is the standard set of interfaces available for connecting IOM-compliant drivers to the system. Any IOM-compliant driver can be connected through a driver input/output (DIO) interface layer to an SIO stream object or through a pipe input/output (PIO) to a PIP. SIO streams and PIP pipes provide powerful and easy to use constructs for data movement in DSP/BIOS systems. They were developed specifically for use with the DSP/BIOS thread types of hardware interrupts, software interrupts, and tasks.

For further information on hardware interrupt (HWI), SWI, and TSK objects, please refer to the *TMS320 DSP/BIOS User's Guide* (SPRU423).

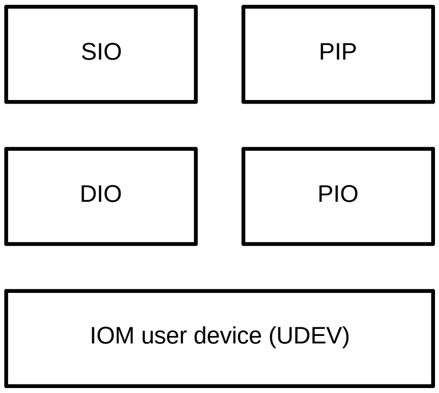

Figure 1. Layers of a DSP/BIOS Driver

The DSP/BIOS driver model includes three layers. The lowest layer is the IOM, which is registered into BIOS via an user device object. The IOM layer must be written by the driver author. For each physical port (such as a serial port on the DSP), there will be a single user-defined device (UDEV) accessed using a BIOS driver. The IOM does not directly interface to the application. Each channel (such as an input or output channel on a serial port), interfaces to the application through an SIO or PIP data transport object. The interface between the IOM and the SIO or PIP object is a driver input/output (DIO) or pipe input/output object (PIO), respectively.

For background on SIO, DIO, PIP, and PIO objects in the BIOS driver model, please refer to the *TMS320 DSP/BIOS User's Guide* (SPRU423) and the *DSP/BIOS Device Driver Developer's Guide* (SPRU616).

There are three key elements that must be incorporated to achieve the goal of converting the data converter tool's output into an IOM-compliant driver. The first is a standard set of functions which the DIO or PIO layer uses in order to interface to it, encapsulated into an IOM interface table. In addition to the IOM interface table, an IOM driver must define the port and channel objects which are used by the DIO layer in order to reference it. When the DIO layer accesses one of the API methods of the IOM interface table, these objects are passed as arguments to ensure that the proper channel or port is accessed. These objects contain all of the state variables used to manage the corresponding port or channel so that the API method called has access to them. The IOM must contain ISRs to respond to the interrupts generated by the underlying hardware to maintain synchronization.

The channel, port objects, and ISRs used in an IOM are application dependent and their structures are defined by the IOM author. The IOM interface table is required to conform to a specific format used by the DIO or PIO layer to interface to the IOM.

```
typedef Int  (*IOM_TmdBindDev)(Ptr *devp, Int devid, Ptr devParams);
typedef Int  (*IOM_TmdUnBindDev)(Ptr devp);
typedef Int  (*IOM_TmdControlChan)(Ptr chanp, Uns cmd, Ptr args);
typedef Int  (*IOM_TmdCreateChan)(Ptr *chanp, Ptr devp, String name, Int mode,
                      Ptr chanParams, IOM_TiomCallback cbFxn, Ptr cbArg);
typedef Int  (*IOM_TmdDeleteChan)(Ptr chanp);
typedef Int  (*IOM_TmdSubmitChan)(Ptr chanp, IOM_Packet *packet);
typedef struct IOM_Fxns
{
    IOM_TmdBindDev       mdBindDev;
    IOM_TmdUnBindDev     mdUnBindDev;
    IOM_TmdControlChan   mdControlChan;
    IOM_TmdCreateChan    mdCreateChan;
    IOM_TmdDeleteChan    mdDeleteChan;
    IOM_TmdSubmitChan    mdSubmitChan;
} IOM_Fxns;
```

A short description of the required functions of the IOM vector table (vTab) is given below. For further detail, please refer to the *DSP/BIOS Driver Developer's Guide* (SPRU616).

1.1 mdBindDev

mdBindDev is called automatically by DSP/BIOS during its initialization phase for every user device (UDEV) object which has been declared in the system. This initialization phase is executed before the C program's entry into main and after execution of the `c_int00` standard C initialization routine.

IOM channels may be dynamically created and deleted using the DIO_create, DIO_delete, SIO_create, and SIO_delete functions, but all user devices must be statically created in the BIOS textual configuration file.

It is the responsibility of the mdBindDev function to create and initialize the driver's port object, a handle to which is returned by reference through the device pointer, devp. Typically, this function would also be used to initialize all physical device ports used by the driver. For instance, the driver for a data converter connected to the DSP via a serial port would often use `mdBindDev` to configure the serial port registers and then program the data converter for operation if necessary.

1.2 mdUnBindDev

The mdUnBindDev function is never called as UDEV objects must be statically created in the system and cannot be deleted. It is included in the vTab structure for future upgrades in order to support dynamic deletion of UDEV objects and would be used to free port resources back to the system.

1.3 mdControlChan

mdControlChan is provided to support the SIO_ctrl function. The mdControlChan function is typically used to issue control operations to the device associated with the driver. For instance, SIO_ctrl might be used to change the gain in a codec which supports a programmable gain.

1.4 mdCreateChan

mdCreateChan is called by BIOS whenever a DIO object is created in the system. For statically created objects, mdCreateChan is called during the BIOS initialization phase following `c_int00` before entry into main.

It is the responsibility of the mdCreateChan function to create and initialize the driver's channel object for each channel that is added to the driver. A handle to the created channel object is returned to the calling function via the chanp pointer, which is passed into the function by reference so that its value can be overwritten. Also, a handle to the port object which is created and returned by mdBindDev is given to the function as a parameter for use if needed.

Common duties to incorporate into mdCreateChan are configuration of the DMA channel if used and initialization of any objects used by the channel, such as a queue for storing data packets until they can be sent or received. Furthermore, mdCreateChan needs to initialize the channel object with the callback function and argument that are passed as its parameters. This callback function is provided by the DIO and must be called at the completion of each transfer in order to inform the DIO that the transfer has completed.

1.5 mdDeleteChan

myDeleteChan is called by BIOS whenever a DIO object is deleted from the system using DIO_delete. It is the responsibility of mdDeleteChan to free any resources which the channel had allocated for itself, such as a DMA channel if used and any memory resources which were allocated in mdCreateChan.

1.6 mdSubmitChan

mdSubmitChan is the mechanism used to pass packets to the driver when SIO_issue (for an output channel) or SIO_reclaim (for an input channel) is called. The channel object of the channel to be used for the read or write is passed to the function by the DIO or PIO, as well as the size and location of the buffer to read from (for an output channel) or write into (for an input channel), which are passed in the IOM_packet.

It is the duty of mdSubmitChan to set up data transfers for each of the data packets that it is called with. The actual transfer will likely be handled either by a DMA channel or by an ISR, so the mdSubmitChan call needs to either configure the DMA channel if used, or set state variables in the channel object as needed to support the transfer. Furthermore, if a transfer is currently in progress, it is the job of mdSubmitChan to store the packet for later transfer. Typically, IOM drivers use a queue for this purpose.

2 The Data Converter Plug-In Tool

The driver files which are generated by the data converter plug-in tool (DCP) of Code Composer Studio are similar in both form and function to the IOM model, as might be expected. Similarly to the IOM interface table (i.e. the vTab), the driver produced by the DCP has a TI data converter function table defined by the TTIDC structure in “TIDC_api.h”.

```
typedef struct
{
      TTIDCSTATUS (*configure) (void *pDc);
      void (*power) (void *pDc, int bDown);
      long (*read) (void *pDc);
      void (*write) (void *pDc, long lData);
      void (*rblock) (void *pDC, void *pData,
           unsigned long ulCount, void (*callback) (void *));
      void (*wblock) (void *pDC, void *pData,
           unsigned long ulCount, void (*callback) (void *));
      void* reserved[4];
} TTIDC;
```

Similarly to the IOM driver, the driver produced by the DCP defines a port object. The DCP driver does not define channel objects. The information which would typically be stored in a channel object in the IOM model (such as EDMA channel information) is stored in the DCP port object. The entire object is initialized by a single configure function, instead of separating the configuration of the driver into port (mdBindDev) and channel (mdCreateChannel) configuration functions.

configure

This function must be called by the DCP driver user to initialize the data converter, serial port and EDMA channel(s) if used. Only after configure is called may power, read, write, rblock, and wblock be called. The configure function must be passed a handle to the data converter object which is created by the DCP and placed at the top of the `_obj.c` file.

power

This is an optional function used to power down data converters that support power down.

read and write

These functions read or write a single sample.

rblock and wblock

These functions read or write a block of data from or into an array. In addition to being passed a pointer to the array and the size of the array (count), these functions may optionally be passed a pointer to a callback function (or NULL value of 0 if no callback is desired). If a callback function is specified, then it will be called upon the completion of the block transfer.

For more information on the DCP format, access the Data Converter Plug-in via Tools → Data Converter Support and press the help button under the System tab.

There is also an application note available titled *Getting Started with the Data Converter Plug-in* (slaa210).

3 Interface Layer – Building IOM Compliant Drivers from DCP Tool Outputs

This application note develops an adaptation layer to interface the DCP driver to the DIO or PIO so that DCP driver files may be used with stream and pipe interfaces analogously to the drivers provided in TI's driver development kit (DDK). As a test case, the layer was developed using the aic23 codec and tested on both the DSK6416 and the DSK6713, both of which use this codec. The adaptation layer has been written generically, however, so that only minor or perhaps no changes will be needed in order to use it with other data converter devices.

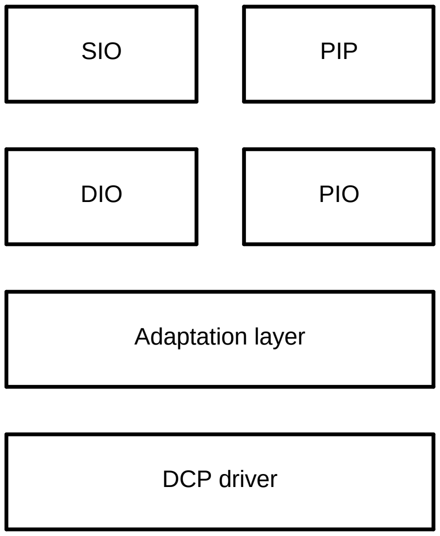

Figure 2. DCP Driver interfaced to DIO and PIO through adaptation layer

In order for the adaptation layer to interface to DIO and PIO objects, it must incorporate an IOM function table, i.e. the vTab. Each of the functions in this adaptation layer IOM function table will then call the DCP driver functions as appropriate. Furthermore, this adaptation layer will manage a queue of pending transfers (functionality not built into the taic23 driver) to support streams and pipes with more than two buffers.

An additional goal is to develop the adaptation layer in a fashion that is general enough that it may be easily used with other DCP drivers in addition to the taic23. In order to meet this goal, only functions from the DCP driver's function table are used. None of the elements in the Aic23_1 object are directly accessed by the adaptation layer, with the exception of these six functions, which are guaranteed to be present in all DCP drivers.

The remainder of this section will therefore step through the IOM functions developed for the adaptation layer to explain how they were developed.

3.1 mdBindDev

As previously stated, the mdBindDev function is called automatically by BIOS during the BIOS initialization phase for every user device defined in the BIOS configuration file. The device identification number (devid) and the device configuration parameters (devParams) which are passed to the function are specified in the BIOS configuration file as entries in the user device. This implementation does not use the devid value which is passed to the function, so this value can be set to zero in the user device object. The devParams pointer needs to be a pointer to the DCP object, i.e. Aic23_1 in this case, which contains the configuration information for the serial port and data converter as well as state variables. The function modifies the passed-by-reference pointer devp to point to the port object created by the function.

It is worth noting that neither mdBindDev nor any of the other functions in the adaptation layer directly reference either this object or any of the API functions generated by the DCP (which are accessed instead through their pointers in the DCP object). This is done in order to reduce the effort necessary to use this adaptation layer with other DCP generated device drivers.

```
static Int mdBindDev(Ptr *devp, Int devid, Ptr devParams)
{
       int oldInUse;
       TTIDCSTATUS status;
       TTIDC *hDCFxns = devParams;
       // Driver requires valid DC object passed as its parameters
   if (devParams == NULL) {
      return (IOM_EBADARGS);
    }
```

The first section of mdBindDev declares the necessary variables. One of these variables, hDCFxns, is worth commenting on. The reader will note that it is of type TTIDC, which is the structure which holds the six functions of the DCP driver’s functions table (configure, power, read, write, rblock and wblock). This pointer is initialized to devParams, which will always point to the DC object (i.e. Aic23_1). This works in the general case because pointers to the six functions are always the first six elements of every DC object. This pointer to the table of DC functions is used to call the functions when needed instead of calling them directly. This is done because the names for these functions differ depending on which data converter is used. By accessing the functions through the DC object, a more generic method is used that will not have to be changed if a data converter other than the AIC23 is used.

```
oldInUse = ATM_setu(&(port.inUse), TRUE);
   if(oldInUse)
        return(IOM_EINUSE);
```

In these lines of code, the driver sets a flag in the port object to show that it is in use and tests to see if it was previously in use so that users do not accidentally create two user devices which have accessed the same port object.

```
port.hDCObj = devParams;
```

A handle to the data converter object (which is required to be passed as the device parameters structure, devParams) is one of the elements in the port object.

```
status = hDCFxns->configure(devParams);
if(status == TIDC_NO_ERR){
             *devp = &port;
    return (IOM_COMPLETED);
    }
    else{
        if(status == TIDC_ERR_BADARGS )
             return ( IOM_EBADARGS );
        if(status == TIDC_ERR_NOCHIPRES )
                     return ( IOM_EFREE );
        return ( IOM_EBADIO );
    }
}
```

The final section of the mdCreateChan function calls the DC configure method via the functions table handle hDCFxns. If this DC configure function returns a value other than TIDC_NO_ERR (i.e. no error), then the mdCreateChan function will return an error as well. If no error is encountered, the devp pointer is set to point to the port object and the function returns IOM_COMPLETED.

3.2 mdCreateChan

mdCreateChan is called upon the creation of an SIO stream object using either static creation via the BIOS configuration file (in which case it is called during BIOS init) or at run time using SIO_create. Upon creation of a new SIO object, the SIO_create function calls Dxx_open to open the DIO object specified. Dxx_open in turn calls mdCreateChan, using the parameters specified in the DIO object. The pointer to a channel parameters structure is set as a property of the DIO, and the callback function and argument are non-user-specified values passed by the DIO, used by the IOM driver to inform the DIO layer when a transfer has completed. Additionally, the DIO layer passes by reference the pointer chanp (channel pointer), which mdCreateChan sets to point to the channel object that it creates. It is this channel object which the DIO layer references in subsequent calls to mdSubmitChan.

Though the DCP driver does not use a channel structure, the mdCreateChan function of the IOM-DCP adaption layer creates a channel object, which is used to manage incoming packets. The AIC23 DCP driver can accept only one packet at a time for transfer in the general case (see Section 5, AIC23 DCP Driver Issue), after which the rblock and wblock functions simply return with no effect. This trait of the DCP driver is specified in d2iuser.h as the MAXLINKCNT parameter, which is set to 1 for this driver. In order to support SIOs with greater than one buffer, a queue is set up and managed by each channel. The queue, as well as the various other parameters specific to each channel, are encased into the channel object.

```
static Int mdCreateChan(Ptr *chanp, Ptr devp, String name, Int mode,
                     Ptr chanParams, IOM_TiomCallback cbFxn, Ptr cbArg)
{
       IOMChanHandle hChan;
       IOMPortHandle hPort;
       int oldInUse;
       hPort = devp;
```

The mdCreateChan function begins with the standard declarations phase and initializes the port handle.

```
    if (mode == IOM_INPUT) {
#if INPUT_SUPPORTED
       hChan = &inputChan;
       oldInUse = ATM_setu(&(hChan->inUse), TRUE);
       hChan->mode = INPUT;
#else
       return(IOM_EBADARGS);
#endif
    }
    else {
#if OUTPUT_SUPPORTED
       hChan = &ouputChan;
          oldInUse = ATM_setu(&(hChan->inUse), TRUE);
       hChan->mode = OUTPUT;
#else
       return(IOM_EBADARGS);
#endif
    }
    If(oldInUse)
        Return(IOM_EINUSE);
```

The mdCreateChan function tests to see if the channel it is requested to open is an input channel or an output channel. Assuming that the requested mode is supported by the driver (as set by the user in "d2iuser.h"), mdCreateChannel initializes the channel structure appropriately. If the requested mode is not supported, it returns IOM_EBADARGS. The function also tests to see if the requested channel was already in use and if so returns the error IOM_EINUSE.

```
/* Initialize the channel structure */
hChan->hDCObj = hPort->hDCObj;
hChan->IOMPort = hPort;
hChan->cbFxn = cbFxn;
hChan->cbArg = cbArg;
hChan->submitCount = 0;
swiIsrAttrs.priority = IOM_SWI_PRI;
hChan->swiIsr = SWI_create(&swiIsrAttrs);
/* Initialize the packet queue */
QUE_new(&hChan->packetQueue);
QUE_new(&hChan->allPacketQueue);
*chanp = (Ptr) hChan;
return (IOM_COMPLETED);
}
```

The primary duty of mdCreateChan is to initialize the channel structure as seen in the lines of code above. Additionally, the function initializes the queue objects which are used to manage incoming packets so that the DCP driver is not overloaded. Finally, the function sets the value of the passed-by-reference pointer chanp to point to the newly initialized channel object as a return value to the calling DIO and returns IOM_COMPLETED.

mdCreateChan also creates a BIOS software interrupt object, a handle to which is kept in the channel object. This software interrupt is used to encase the IOM adaptation layer's callback function (which is called by the DCP driver whenever a transfer completes.) This callback function is placed into a software interrupt because it does not require immediate completion, and by placing it into a SWI, it is guaranteed not to interfere with other hardware interrupts in the system, which may have immediate real-time completion requirements.

3.3 mdSumbitChan

mdSubmitChan is the main engine of the IOM driver. Each time that SIO_issue (for an output stream) or SIO_reclaim (for an input stream) is called, the command is transferred through the DIO layer to eventually call mdSubmitChan. The DIO passes mdSubmitChan the channel object handle chanp which was initialized in mdCreateChan as well as an IOM_Packet (by pointer) which is generated using the parameters passed to SIO_issue or SIO_reclaim. The IOM packet contains the location and size of the buffer to be read from or written to as well as a few other pieces of information.

```
static Int mdSubmitChan(Ptr chanp, IOM_Packet *packet)
{
    IOMChanHandle chan = chanp;
    Uns imask;
       TTIDC *hDCFxns = chan->hDCObj;
       void *pData;
    unsigned long ulCount;
```

The function begins with a declarations phase of local variables. A functions table pointer is cast from chan->hDCObj (the handle to the DCP driver object which is one of the elements in the channel structure). This assumes that the DCP driver function table (create, power, read, write, rblock and wblock) is the first element in the DCP object, which should be true for all DCP-generated drivers. As with mdBindDev, the DCP driver functions are called only via this functions table provided in the DCP object in order to improve portability.

```
if (packet->cmd == IOM_FLUSH  || packet->cmd == IOM_ABORT)
                 return(IOM_ENOTIMPL);
if (packet->cmd != IOM_READ && packet->cmd != IOM_WRITE) {
    return(IOM_ENOTIMPL);
    }
```

This version of the interface layer does not support flush or abort commands (generated by SIO_flush and SIO_idle calls as well as timeouts of pending transfers). While it would have been possible to flush the queue of pending IOM packets managed by the channel object, the DCP driver provides no method for aborting the current transfer via the six standard functions provided in its function table. Thus these SIO functions could not be supported while maintaining portability of the interface layer.

At this time, single read and writes are also not supported by the driver. While these are a natural fit to the read and write functions provided in the DCP function table, they are less likely to be used than the block versions, especially for PIP and SIO-based transfers. Many of the drivers in the IOM Driver Development Kit do not support single transfers, as is also the case with the DCP-generated drivers, many of which have only dummy place holders for the single-word read and write function calls.

```
// Disable interrupts to protect submitCount
imask = HWI_disable();
//  If there is no space available for the new packet, put it on a
//  queue to be linked in when space is available. Otherwise link it in.
if (chan->submitCount >= MAXLINKCNT) {
    QUE_enqueue(&chan->packetQueue, packet);
    }
    else {
       pData = packet->addr;
       ulCount = (packet->size) >> 2;
       if(chan->mode == INPUT)
                hDCFxns->rblock(chan->hDCObj, pData, ulCount, &rIsrIOM);
       else
                hDCFxns->wblock(chan->hDCObj, pData, ulCount, &wIsrIOM);
    }
chan->submitCount++;
```

mdSubmitChan next tests to see if there is room in the DCP driver to support a newly added packet via rblock or wblock. With the AIC23 DCP driver, only one packet may be guaranteed linkable at a time. After the first packet submission, a second packet is not guaranteed linkable until the transfer completes (see Section 5, AIC23 DCP Driver Issue). A second packet submitted to the DCP before the first transfer has completed may be lost. In such a setup, the MAXLINKCNT should be set to one, and any packets submitted to the interface layer which cannot be support by the DCP will be added onto a queue for later submission. If there is room for the DCP driver to supported a new packet (i.e. chan -> submitCount < MAXLINKCNT), it is submitted via rblock or wblock as appropriate. Regardless of whether it is queued or submitted directly, the submission counter (chan --> submitCount) is incremented.

It should be noted that when the rblock or wblock function is called to pass the packet to the DCP driver, a callback function is specified. This callback function is executed by the DCP driver upon the completion of a packet transfer. The callback functions which the interface layer specifies to the driver are rIsrIOM and wIsrIOM, which are discussed below.

```
               QUE_enqueue(&chan->allPacketQueue, packet);
HWI_restore(imask);
return (IOM_PENDING);
}
```

The function completes by placing the packet onto a submission queue named allPacketQueue. This is a separately managed queue from the packetQueue, which is used to manage those packets which are waiting for submission to the DCP driver. allPacketQueue is used to store all packets so that they may be returned to the DIO layer upon completion of the transfer, as will be seen in rIsrIOM.

3.4 *rIsrIOM / rIsrSWI*

The interface layer defines interrupt service routines (ISRs) for both the receive and transmit channels. Since these routines are nearly identical, only the receive channel is discussed in detail below.

As noted above, rIsrIOM and wIsrIOM are passed as callback functions to the DCP function calls of rblock and wblock to be called upon completion of the packet transfers. These functions are used to inform the DIO layer that the transfer has completed (through a second callback function, stored in the channel object) as well as to manage the pending transfers queue and begin a new transfer if one is pending.

```
void rIsrIOM(void *unused)
{
       SWI_post(inputChan.swiIsr);
}
```

The rIsrIOM simply posts the software interrupt in which the bulk of the processing of the ISR is completed. The function which is run within the SWI context is rIsrSWI, below.

```
void rIsrSWI(void *hDCChan)
{
     IOM_Packet *packet;
     TTIDC *hDCFxns = inputChan.hDCObj;
     void *pData;
     unsigned long ulCount;
```

The ISR routine begins with the declaration phase. As in mdSubmitChan and mdBindDev, a pointer to the DCP driver’s function table called hDCFxns is created and initialized to the DCP object, which assumes that this function table is the first element in the DCP object.

```
packet = QUE_dequeue(&inputChan.allPacketQueue);
packet->status = IOM_COMPLETED;
/* Call the callback function */
(*inputChan.cbFxn)(inputChan.cbArg, packet);
inputChan.submitCount--;
```

The ISR’s first responsibility is to inform the DIO layer that the transfer has completed. It does this via the callback function and argument, which are specified by the DIO when it calls mdCreateChan and are stored in the channel object. One of the parameters required by the DIO in this callback function is a pointer to the packet which has completed. This packet is pulled off of the allPacketQueue where it was placed in mdSubmitChannel when it was submitted and used as an argument in the callback function. Finally, the submitCount counter is decremented to indicate that a transfer has completed and there are therefore one fewer submitted transfers awaiting completion.

```
 /*
* See if there are any unlinked packets in the packetQueue
* and if so link them.
*/
if (inputChan.submitCount >= MAXLINKCNT) {
    packet = QUE_dequeue(&inputChan.packetQueue);
                 pData = packet->addr;
                 ulCount = (packet->size) >> 2;
                 hDCFxns->rblock(inputChan.hDCObj, pData, ulCount, &rIsrIOM);
    }
}
```

The ISR completes by checking to see if there are pending transfers on the packetQueue, and if so, submits the transfer to the DCP via a similar mechanism to what has already been described in mdSubmitChan.

3.5 mdDeleteChan

The mdDeleteChan function is called whenever an SIO stream is deleted using SIO_delete. The function frees resources by emptying the two channel queues and also marks the channel no longer in use.

```
static Int mdDeleteChan(Ptr chanp)
{
       IOMChanHandle hChan;
       hChan = chanp;
       hChan->inUse = FALSE;
      // Empty both queues
      while(QUE_get(&hChan->packetQueue) != &hChan->packetQueue);
      while(QUE_get(&hChan->allPacketQueue) != &hChan->allPacketQueue);
      return(IOM_COMPLETED);
}
```

mdDeleteChan returns IOM_COMPLETED upon freeing the requested channel, which is passed via the channel pointer chanp.

4 Using the IOM-compliant DCP Driver

In addition to the source code for the DCP-IOM interface layer described in the previous section, the downloadable files with this application note include loop-through examples for the DSK5510, DSK6416, and DSK6713. These examples are based upon the generated output files of the Data Converter Plug-in for the AIC23 codec, which is used on all three boards. The following setup was used to generate and test the files.

- Data Converter Plug-in – DCP v3.51 was used to generate the DCP driver files
- CCS – Code Composer Studio™ v3.1 was used, configured with the IDE v.5.90 and BIOS v.4.90 codegen tools v.5.10 and chip support library v.2.31, the standard modules which ship with the version 3.1 release.
- DSP/BIOS – Additionally, projects were built to support the above configuration with BIOS v.5.20, which does not ship standard with CCS version 3.1 but is available via update advisor.

Note:

A minor modification was necessary to the taic23_bo.c generated file of the 5510 AIC23 DCP driver in order to achieve correct operation.

In the DSK5510 driver example, the following lines of code had to be removed from the main driver file taic23_ob.c:

```
#ifdef CHIP_5502
#error This driver does not support the C5502!
#endif
```

These lines of code were causing an error even when the user did not specify the c5502 as the DSP in use. The issue is caused by one of the HAL layers, which redefines CHIP_5502 as 1 if it exists and 0 if not, which means that it becomes defined either way and causes this error to trip.

4.1 Limitations of the DCP-IOM Interface Layer

The main limitation of the DCP-IOM interface layer is that it does not support the full functionality of the SIO stream module. SIO_abort and SIO_idle are not supported, and neither are SIO timeouts because the abort command is called internally to the DIO when a timeout occurs. In order to ensure no timeouts, the value of SYS_FOREVER must be specified as the timeout in the SIO attributes (which is the default.) The reason that these functions are not supported is explained in more detail see Section 3.3.

A secondary limitation is that the configuration of the BIOS hardware interrupt objects (HWI) to call the DCP ISRs as necessary through the interrupt dispatcher is not included in the DCP-IOM interface layer. The reason is that the procedure for accomplishing this varies between DSP platforms, and one goal of creating this interface layer was portability across platforms. The code to plug the correct ISR routines into the HWI dispatcher is included in each of the examples provided in the downloadable code associated with this document. Examples for the C5510 DSK, C6713 DSK, and C6416 DSK are all provided. The main.c file for each of these examples contains an initIrq function to handle plugging of the ISR routines, the code from which may be applied to the user's driver in the mdCreateChan function of the interface layer if desired.

4.2 Procedure for Building a Custom Driver

The following is the procedure for creating a custom IOM-compliant driver from the DCP tool and the DCP-IOM interface layer.

1. **Begin by creating a new project or opening the project that you would like to add the custom driver to.**

   If you would like to build a library file for your custom driver, select project →new and specify library (.lib) as the project type.

2. **Next, generate DCP driver files via the Data Converter Support Plug-in.**

   The Data Converter Plug-in may be accessed via (Tools | Data Converter Support…) in code composer studio. If Data Converter Support does not appear under the tools menu, then the Data Converter Plug-in should be installed via update advisor (Help | Update Advisor | Check for Updates…)

   The Data Converter Support window will open on the System tab. Locate the desired data converter from the drill-down lists. When you have located the correct data converter, right click on it and select (Add).

   Select the DSP tab and ensure that the “Dispatcher in DSP/BIOS Used” check box has been checked. (If this box is not checked, the tool will insert the interrupt keyword in front of the ISRs that it generates, which will interfere with DSP/BIOS operation.) Configure the DSP settings as well as the data converter settings as desired under their appropriate tabs.

3. **Insert the DCP-IOM interface layer into your project.**

   The source code files required are dcp2iom.c, dcp2iom.h and d2iuser.h. The header files will automatically be added by dcp2iom.c, assuming that they are included in the same directory.

4. **(Optional) Configure d2iuser.h as needed.**

   If using an ADC or DAC d2iuser.h should be configured so that driver does not allow input or output channel which does not exist. Furthermore, the priority of internally-used SWIs for the DCP-IOM interface layer can be set, as well as setting the MAXLINKCNT parameter, which specifies the maximum number of transfer requests that the DCP driver is guaranteed to accept. For more information, see Section 3.3, mdSubmitChan.

5. **Create the user-defined object (UDEV) for the DCP-IOM interface layer in DSP/BIOS.**

   The UDEV object should be configured as follows:

```
init function = _IOMInit;
function table pointer = _IOMFunctionTable
function table type = IOM_fxns
device id = 0 (unused)
device parameters pointer = (pointer to the driver object created by the DC Support tool,

preceded by an underscore, i.e. _Aic23_1)
device global data pointer = 0 (unused)
```

6. **Create the DIO and SIO or PIO and PIP objects for each channel in DSP/BIOS.**

   No channel parameter structure needs to be specified (field should be left as 0). The UDEV specified by the DIO or PIO must be the name of the UDEV created above. Note that if desired, the SIO or PIP may be dynamically created at run-time, as shown in the included examples.

1. **Finally, configure the Hardware Interrupts to call the proper DCP ISR.**

   This may be accomplished by an ISR configuration function in your main program listing or integrate this configuration into the mdCreateChan function of the DCP-IOM interface layer. Example for configuration of the ISRs is included in the loop-through examples for each platform by the function initIsr.

Note:

After completing the above steps, the user should be able to build a working driver; however, there may be a real-time inconsistency present in the system which is most noticeable in continuous data flow systems (such as audio systems). This is actually an issue with the DCP driver itself and will be encountered whether the driver is adapted to the IOM format or not. For the fix to this issue, please read the next section, Section 5, AIC23 DCP Driver Issue.

5 AIC23 DCP Driver Issue

During the development of the example drivers included with this application note (see Appendix A), an inconsistency in the DCP driver files was found which may lead to diminishing of input and output quality due to real-time misses. The point is a subtle one and is due to an issue in the application of the EDMA interrupt in the DCP driver.

Before specifying this issue, let us examine the DMA linking process used by the DCP generated driver.

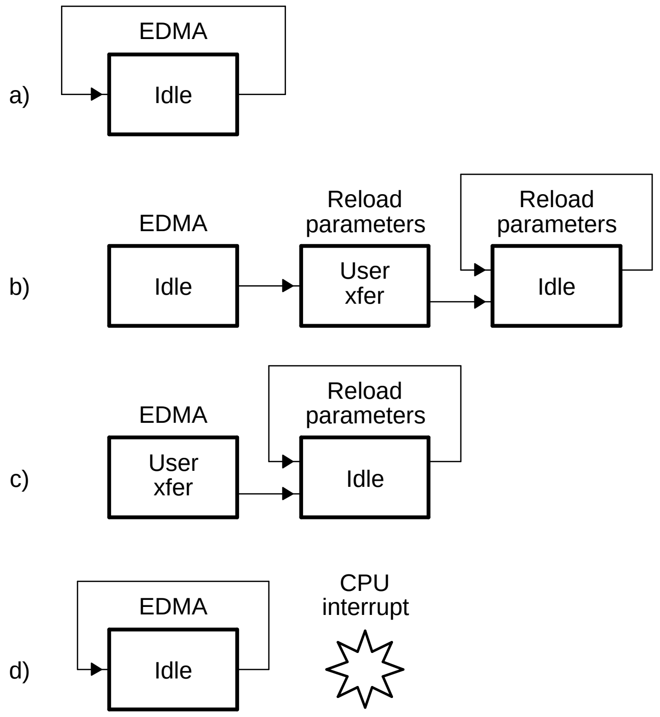

Figure 3. Stages of Reload Parameter Links in the DCP Driver

As shown in Figure 3, the DMA utilization by the DCP driver goes through four stages. When the driver is first initialized (a), the DMA is initialized with an idle transfer, the reload field of which is set to reload another idle transfer in the reload parameters. The first user call of AIC23_submitTransfer (b) loads a user transfer into a reload parameters space and the reload field of the currently running idle transfer is tied to these reload parameters. The reload field of the user transfer is tied to an idle transfer because no other user transfer has yet been specified. When the current idle transfer completes ©), the user transfer is loaded into the DMA to begin running. When the user transfer completes (d), a CPU interrupt is generated, and the reload parameters of the user transfer are loaded into the DMA, which in this case are the idle loop.

The issue with this implementation is that the CPU interrupt does not occur until the completion of the user transfer. By the time the corresponding interrupt service routine has started running, the idle loop has already been loaded into the DMA and has begun running. The result, in an audio application, is a glitch of noise that becomes increasingly perceptible as buffer size is reduced. It should be noted that this is not an issue only when the DCP driver is being converted into an IOM-compliant driver via the DCP-IOM interface layer, but whenever it is being used.

What would be needed in order to overcome this issue would be to modify the DCP driver such that it is guaranteed to always be able to accept two submitTransfer requests in a row such that the completion of the first transfer could reload with the second transfer. This would probably be most easily accomplished in this case by extending the AIC23 driver to have two EDMA parameter reload spaces per channel (plus the third for the idle loop parameter reload) and building the driver to ping-pong between those two reloads. Unfortunately, this discussion is beyond this application note. However, the DCP-IOM interface layer source code has been written with the constant MAXLINKCNT (specified in d2iuser.h) such that it can easily be interfaced to such a modified driver (or any DCP-generated driver which does support multiple data packet submissions at a time). By setting MAXLINKCNT to 2, the user can interface the DCP-IOM interface layer into the newly created DCP driver without having to make any other changes to the DC-IOM interface layer.

6 References

1. *TMS320 DSP/BIOS User's Guide* (SPRU423)
2. *DSP/BIOS Device Driver Developer's Guide* (SPRU616)
3. *Getting Started with the Data Converter Plug-in* (SLAA210)

Appendix A Listing of Included Source Files

The companion source files to this application note are downloadable for free from the application note's abstract page by selecting "Download Associated Code Files" at the bottom.

The following seven directories are present at the top level of the .zip file:

- **Bios4.9 dsk5510 dcp2iom**
  This directory contains a Code Composer Studio v3.1 / BIOS v4.9 project for a test working loop through example on the c5510 DSP Starter Kit developed from the AIC23 DCP driver and the interface layer outlined in this application note.
- **Bios4.9 dsk6416 dcp2iom**
  Loop through example on the c6416 DSP Starter Kit, BIOS v4.9
- **Bios4.9 dsk6713 dcp2iom**
  Loop through example on the c6713 DSP Starter Kit, BIOS v4.9
- **Bios5.2 dsk5510 dcp2iom**
  Loop through example on the c5510 DSP Starter Kit, BIOS v5.2
- **Bios5.2 dsk6416 dcp2iom**
  Loop through example on the c6416 DSP Starter Kit, BIOS v5.2
- **Bios5.2 dsk6713 dcp2iom**
  Loop through example on the c6713 DSP Starter Kit, BIOS v5.2
- **Dcp2iom interface layer**
  This directory contains the source for the three files developed in this application note which constitute the DCP-IOM interface layer.

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless