Frameworks

Introduction

In this chapter, we will discuss a current problem for DSP system design and suggest a possible solution provided by TI.

Learning Objectives

Objectives

◆ System Block Diagram
◆ Standard I/O (SIO) - Using Streams
◆ Device Drivers (IOM)
◆ Reference Frameworks (RF)
◆ Lab 12/12a – Using SIO and Modifying an IOM Driver
Chapter Topics

Frameworks ..............................................................................................................................................12-1

System Software and I/O Interfacing ........................................................................................................12-3
  Growing Your Own Algorithm? ..................................................................................................................12-3
  System Software ..................................................................................................................................12-3
  BIOS I/O Interface Models ....................................................................................................................12-4
  Lab12 – Example SIO/Driver Architecture .........................................................................................12-5
  Standardized I/O (SIO) – Concepts ........................................................................................................12-6
  SIO – Creating the Streams ..................................................................................................................12-7
  SIO – Allocating Buffers and Priming the Streams .............................................................................12-7
  SIO – TSK Code Using Streams ...........................................................................................................12-8

Understanding Device Drivers (IOM) .....................................................................................................12-9
  IOM – LAB 12 Example SIO/Driver Architecture ............................................................................12-9
  IOM – (I/O Mini) Driver Files ................................................................................................................12-10
  IOM – Mini-Driver Interface ..................................................................................................................12-10
  IOM – Driver Development Kit (DDK) ..............................................................................................12-11

Reference Frameworks (RFx) ..................................................................................................................12-12
  What is a Reference Framework? .........................................................................................................12-12

Lab 12 – Using SIO (Streams) and Drivers ..............................................................................................12-15
  Lab 12 Procedure ................................................................................................................................12-16

Lab12a: Modifying the Driver ..................................................................................................................12-22
  Build the New Library: myDriver.lib .................................................................................................12-27
  Remove Old Driver/Source Files and add myDriver.lib ...............................................................12-29
  Make the last few Code Adjustments ...............................................................................................12-29
  Build – Load – Run - Save ...................................................................................................................12-30
System Software and I/O Interfacing

Growing Your Own Algorithm?

Grow Your Own ...

- too costly to develop
- too costly to enhance
- too costly to maintain

alg
app + alg
app + algA + algB + ...
app + sched + algAn + algBn + ...
app + sched + I/O + algAn + algBn + ...
app + sched + I/O + comm + algAn + algBn + ...

System Software

Program = Code + Data
Embedded System = Program + Mem. Management + Init + H/W + I/O ...

- XDAIS provides a common interface to Algorithms
- But, what common interface exists for hardware?
- Let’s break it into two pieces: interface + driver

Interface first...
BIOS I/O Interface Models

DSP/BIOS Thread Types

- TSK or SWI
- SWI
- TSK or SWI

I/O Mini-Driver (IOM)

- SIO
- PIP
- GIO

Any mini-driver can be used with any DSP/BIOS I/O model

- All models pass pointers to buffers instead of copying data.
- SIO and GIO provide blocking functions; PIP does not.
- SIO is the most flexible and easiest to use with IOM drivers.

Let's look at our lab's system architecture...
System Software and I/O Interfacing

Lab12 – Example SIO/Driver Architecture

SIO (Standardized I/O) is a communication protocol – or interface – that can be used to communicate between a thread (in our case, a TSK) and a driver. The key point here is that if both the application (TSK) software and the driver use the same interfacing method (I/O), they can be written independently and neither one needs to know the specifics of what the other is doing with the data.

There are actually three types of interfaces as we discussed before (PIP, SIO, GIO). SIO happens to be the easiest to use when talking to a driver – so that’s what we’re going to use in the lab.

The analogy on right hand side fits nicely. The hardware (McBSP, EDMA, codec) are the “power plant”. They produce the data (electricity). The driver contains the transmission lines and the adapter to adapt the high voltage lines down to a plug in your house or someone else’s. SIO is the plug of the fan. You can take your fan (TSK) anywhere you like and plug it into a socket and make it work. You don’t have to know where the power plant is and you need not be concerned with how the high voltage is converted to the socket you use in your home. Also, the power plant and transmission lines need not care WHAT you’re plugging into the wall – but the electricity flows and everything works nicely. This is the beauty of using streams.

Lab12 – Example SIO/Driver Architecture

Let’s take a closer look at SIO...
Standardized I/O (SIO) – Concepts

This is a simplified picture of how SIO works. But actually, it’s this simple. SIO consists of two types of streams – an INPUT stream and an OUTPUT stream. SIO uses fancy names like “issue” (which means GIVE a buffer) and “reclaim” (which means TAKE a buffer). Many systems give and take full and empty buffers all over the system.

The IOM driver on the left fills up buffers and issues them to the IN stream and reclams (takes) empty buffers back from the TSK to fill them up again. On the TSK side, the code will issue empty buffers to the driver to fill up and reclaim (take) full buffers to process. The OUT stream works the same way.

Instead of copying buffers, streams (SIO) passes pointers to the buffers increasing the efficiency of the system. Another nice feature of streams is that a “reclaim” blocks (pends) until the buffer is issued by the other side. So, the TSK might say “give me a buffer” using a “reclaim” and the TSK will pend until that buffer is ready. No additional coding steps are necessary.

- Communications protocol: issue = give a buffer, reclaim = take a buffer
- TSK and IOM use a common interface (SIO) and are independent
- Reclaim blocks (pends) until buffer is ready (has been issued)
- Pointers to the buffers are passed, not the buffers themselves (efficiency)
- Multiple buffers can be issued – SIO maintains the queue

So, what does our code look like to create/use streams?
SIO – Creating the Streams

1. SIO – Creating the Streams

```c
/* inStream and outStream are SIO handles created in main */
SIO_Handle inStream, outStream;

void createStreams()
{
    SIO_Attrs attrs;
    attrs = SIO_ATTRS;
    attrs.align = BUFALIGN;
    attrs.model = SIO_ISSUERECLAIM;
    attrs.segid = ISRAM;

    /* open the I/O streams */
    inStream = SIO_create("dioCodec", SIO_INPUT, BUFFSIZE*4, &attrs);
    outStream = SIO_create("dioCodec", SIO_OUTPUT, BUFFSIZE*4, &attrs);
}
```

SIO – Allocating Buffers and Priming the Streams

2. SIO – Allocate Buffers and Prime Streams

```c
void primeStreams()
{
    Ptr rcvPing, rcvPong, xmtPing, xmtPong;

    /* Allocate buffers for the SIO buffer exchanges */
    rcvPing = (Ptr)MEM_calloc(0, BUFFSIZE*4, BUFALIGN);
    rcvPong = (Ptr)MEM_calloc(0, BUFFSIZE*4, BUFALIGN);
    xmtPing = (Ptr)MEM_calloc(0, BUFFSIZE*4, BUFALIGN);
    xmtPong = (Ptr)MEM_calloc(0, BUFFSIZE*4, BUFALIGN);

    /* Issue the first & second empty buffers to the input stream */
    SIO_issue(inStream, rcvPing, BUFFSIZE*4, NULL);
    SIO_issue(inStream, rcvPong, BUFFSIZE*4, NULL);

    /* Issue the first & second empty buffers to the output stream */
    SIO_issue(outStream, xmtPing, BUFFSIZE*4, NULL);
    SIO_issue(outStream, xmtPong, BUFFSIZE*4, NULL);
}
```
void processBuffer(void)
{
    short *source;
    short *dest;
    createStreams();
    primeStreams();
    while(1) {
        SIO_reclaim(inStream,(Ptr *)&source, NULL);
        SIO_reclaim(outStream,(Ptr *)&dest, NULL);
        // *** PROCESS ***
        SIO_issue(outStream, dest, BUFFSIZE*4,NULL);
        SIO_issue(inStream, source, BUFFSIZE*4,NULL);
    }
}
Understanding Device Drivers (IOM)

IOM – LAB 12 Example SIO/Driver Architecture

Now that we understand streams (SIO) and how to make them work, let’s focus our attention on the other side of the system software – the driver. The driver is built using the Chip Support Library (CSL) that makes specific API calls to talk directly to the hardware (EDMA, McBSP, codec).

So far in this class, you’ve done all of that work – writing configuration structures and _open() and _config() code to talk to the hardware. What the driver (IOM) does is encapsulates all of the necessary code to talk to the hardware and places a stream (SIO) interface of top of that to talk to application software (like our TSK).

In the lab, we’re going to do two things: (1) drop in an off-the-shelf driver for the DSK and change our TSK to use streams to communicate with it; (2) modify the driver to perform channel sorting. Both of these activities will be beneficial to any system designer.

Lab12 – Example SIO/Driver Architecture
IOM – (I/O Mini) Driver Files

IOM Driver Files

DSK 6416 IOM Driver Files (from DDK)

- dsk6416_aic23.c  ➢ AIC23 codec driver implementation specific to the DSK6416 board.
- dsk6416_codec_devParams.c  ➢ Defines the default parameters used for DSK6416_EDMA_AIC23 IOM driver
- c6x1x_edma_mcbsp.c  ➢ Generic McBSP driver for the TMS320C6x1x series. Uses the EDMA.
- dsk6416_edma_aic23.c  ➢ Driver for the aic23 codec on the 6416 DSK. Requires the generic TMS320C6x1x McBSP driver.

Note: 6713 DSK files are the same other than the generic driver

➢ To add channel sorting to the EDMA, we need to modify the last two files which contain the EDMA structures/initialization.
➢ We will modify these files, then create our own library (output a .lib file instead of .out) – myDriver.lib – to use in our project.

IOM files contain functions and data structures...

IOM – Mini-Driver Interface

Mini-Driver Interface (IOM)

IOM Interface Consists Of:

Functions:
- init function
- IOM_mdBindDev
- IOM_mdUnBindDev
- IOM_mdControlChan
- IOM_mdCreateChan
- IOM_mdDeleteChan
- IOM_mdSubmitChan
- interrupt routine (isr)

Data Structures:
- BIOS Device Table
  - IOM function table
  - Dev param's
  - Global Data Pointer (device inst. obj.)
- Channel Params
- Channel Instance Obj.
- IOM_Packet (aka IOP)

➢ You will get a chance to examine several of these functions in the lab

What platforms does the DDK support?
IOM – Driver Development Kit (DDK)

The DDK (Driver Development Kit) is free of charge from TI and contains all of the necessary files, functions and structures to communicate to specific hardware on the development platforms listed below.

### Driver Developer Kit (DDK)

<table>
<thead>
<tr>
<th>Platform</th>
<th>Video Capture / Display</th>
<th>PCI</th>
<th>EMAC</th>
<th>McBSP</th>
<th>McASP</th>
<th>HiW UART</th>
<th>S/W UART</th>
<th>Utopia</th>
</tr>
</thead>
<tbody>
<tr>
<td>6711 DSK</td>
<td>External</td>
<td>External</td>
<td>✓</td>
<td>AD535</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6713 DSK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6416 VT1420</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6416 TEB</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6416 DSK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>DM642 EVM</td>
<td>✓ ✓ ✓ ✓ ✓ ✓ ✓ ✓</td>
<td>✓ ✓ ❑</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* We have only included C6000 systems in this table

- Provided Royalty Free (for use on TI DSP’s)
- Requires CCS v2.2 or greater
- To download, go to www dspvillage com and select Software ➔ Peripheral Drivers.

DDK v1.0
DDK v1.1
DDK v1.2
Reference Frameworks (RFx)

What is a Reference Framework?

- Application Framework for systems which integrate:
  - XDAIS algorithms
  - IOM Drivers
- Statically or Dynamically instantiates XDAIS algorithms
- Provides the ALGRF module which uses BIOS Memory Management
- Uses IOM to talk to codecs (or other hardware)

Blank Page Syndrome

- Who wants to start this way?
An Application Blueprint

- Does something useful
- Is easy to adapt and change
- Creates modules that can be reused
- Includes documentation and comments
- Written in portable, high-level language
- Has a well standardized file structure
- Uses various tools together (BIOS, IOM, RTA, etc)
- Is NOT a blank page

Reference Framework Characteristics

- Good Starterware
- Design-ready, reusable, C language source code
  - Not demo code
- A complete “generic” application running on TI DSK's
  - Supplied with “FIR type” eXpressDSP compliant algorithms
- Criteria to enable appropriate selection of RF level
- System Budgeting
  - Memory footprint
  - Instruction cycles
- Adaptation guide for adding algorithms, channels, and drivers
- An API Reference Manual for new (library) modules
- Consistent documentation in RF application notes
  - SPRA79x
  - eXpressDSP for Dummies
- RF1, RF3, RF5: Licensed with every TMS320 device - royalty free
Reference Frameworks

<table>
<thead>
<tr>
<th>Design Parameter</th>
<th>RF1</th>
<th>RF3</th>
<th>RF5</th>
<th>RF6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Configuration</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Dynamic Object Creation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Static Memory Management</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Dynamic Memory Allocation</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Recommended # of Channels</td>
<td>1 to 3</td>
<td>1 to 10+</td>
<td>1 to 100</td>
<td>1 to 100</td>
</tr>
<tr>
<td>Recommended # of XDAIS Algos</td>
<td>1 to 3</td>
<td>1 to 10+</td>
<td>1 to 100</td>
<td>1 to 100</td>
</tr>
<tr>
<td>Absolute Minimum Footprint</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Single/Multi Rate Operation</td>
<td>single</td>
<td>multi</td>
<td>multi</td>
<td>multi</td>
</tr>
<tr>
<td>Thread Preemption and Blocking</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Implements Control Functionality</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Supports</td>
<td>HWI</td>
<td>HWI, SWI</td>
<td>HWI, SWI, TSK</td>
<td>HWI, SWI, TSK</td>
</tr>
<tr>
<td>Implements DSPLink (DSP↔GPP)</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Total Memory Footprint (less algos)</td>
<td>3.5KW</td>
<td>11KW</td>
<td>25KW</td>
<td>tbd</td>
</tr>
<tr>
<td>Processor Family Supported</td>
<td>C5000</td>
<td>C5000, C6000</td>
<td>C5000, C6000</td>
<td>None</td>
</tr>
</tbody>
</table>

Planned, but not yet available

RF3 Block Diagram (out of the box)

- IOM Drivers for input/output
- Two processing threads with generic algorithms
- Split/Join threads used to simulate stereo codec.
  (On C6416/C6713 DSKs, how could we save cycles on split/join?)

How about using the EDMA’s channel sorting capability to replace the “Split” and “Join” SWI’s. This can be done since an IOM driver can be written to allow connections to multiple PIP’s. All of this means less CPU MIPs tied up with moving data – and thus they can be applied to your algorithms.
Lab 12 – Using SIO (Streams) and Drivers

In our earlier labs, we constructed the entire I/O interface by hand via the EDMA, McBSP and codec. The code we have written so far is about 80-90% of a driver. You now know what the “low-level” interface looks like. The next logical step is to add the few missing pieces to make our code into a driver that encapsulates the EDMA, McBSP and codec I/O interface.

Instead of creating a driver from our own code, it is much easier to take a driver that already exists and modify it to meet our system specs. This is what most people will do anyway. Knowing the low-level EDMA and McBSP structures, you can easily modify an existing driver to work in your own particular system.

So, we’re going to do this lab in two pieces:

First, we will use a canned “off-the-shelf” driver from the DDK (Driver Development Kit) which covers the I/O interface (EDMA, McBSP, codec) and then modify our processing code to communicate with the driver using Standard I/O (SIO, i.e. streams).

In the 2nd part of the lab, we will modify the existing driver to perform channel sorting and get it working with our new processing code. This will provide you with the full knowledge of how to use drivers in the C6000 world and modify them to your liking.

Lab12/12a – SIO and IOM

- Drop in an IOM driver and modify TSK to use SIO
- Modify IOM driver to perform channel sorting
Lab 12 Procedure

In the first lab, the driver from the DDK hands us interleaved data – as opposed to the channel sorting we’ve done all week long. So, we need to add “split” and “join” functions to properly talk to the off-the-shelf driver. We will add the necessary stream interface (SIO) to talk to the driver and see how the code runs. Let’s give it a try…

**Open Audioapp.pjt and Remove Existing Files & Code**

1. Reset the DSK, start CCS and open audioapp.pjt.

2. Remove files from the project.
   Remove the following files from the project:
   - codec.c
   - edma.c
   - mcbsp.c

   We don’t need these files because what they contain is already written in the DDK driver.

3. Delete code from main.c.
   Open main.c and remove the following lines (again, this code is not necessary because it is already contained in the driver):
   - #include <csl.h>
   - #include <csl_edma.h>
   - #include <csl_irq.h>
   - #include “edma.h”
   - #include “mcbsp.h”
   - #define PING 0
   - #define PONG 1
   - Void initHwi(void); //the ISR is inside the driver
   - Extern int pingOrPong
   - initMcBSP;
   - initEdma;
   - initHwi (both the call and the function)
   - McBSP_write …
Add the “off-the-shelf” Driver to your Project

4. Add the codec devParams to your project.
   Add the following file to your project. This file is located at
   c:\IW6000\labs\audioapp\IOM_orginal:

   dsk6416_codec_devParams.c
   OR
   dsk6713_codec_devParams.c

   This file is already located in your \audioapp directory.

5. Add a user-defined IOM driver device to your project.
   Open audioapp.cdb.

   Click on the + next to Input/Output. Click on the + next to Device Drivers. Right-click on
   User-Defined Devices and insert a new UDEV. Rename it to udevCodec.

   Right-click on this new user-defined device and select Properties. Modify the properties as
   follows. These names can be found in the header files for the chosen driver. Also, the func-
   tion table type is IOM_Fxns because the model we’re using is an IOM model. If using the
   6713DSK, replace “6416” with “6713” in the parameters below:

   init function:       _DSK6416_EDMA_AIC23_init
   function table ptr: _DSK6416_EDMA_AIC23_FXNS
   Function table type: IOM_Fxns
   device id:          0x00000000
   device params ptr:  _DSK6416_CODEC_DEVPARAMS
   device global data ptr:  0x00000000

   Click OK.

6. Add a DIO-Class Driver to your Project.
   Under DIO-Class Driver, insert a new DIO called dioCodec. Make sure its properties are as
   follows:

   use callback version of DIO function table: unchecked
   device name:        udevCodec
   channel parameters: 0x00000000

   Close and save the cdb.
7. **Add the DDK directory to your search path.**
   Add the following path to your include search path. Select:
   
   Project → Build Options → Compiler Tab
   
   Then select the Preprocessor category. Locate the Include Search Path and add the following to the path:
   
   \[\text{C:}\backslash\text{CCStudio\_v3.1\backslash ddk\backslash include}\]
   
8. **Add the device driver library files to your project.**
   Add the following 2 device driver library files to the project. You will find these files located at: \text{c:}\backslash\text{CCStudio\_v3.1\backslash ddk\backslash lib}.
   
   **6416DSK:**
   
   \[\text{dsk6416\_edma\_aic23.l64}\]
   \[\text{c6x1x\_edma\_mcbsp.l64}\]
   
   **6713DSK:**
   
   \[\text{dsk6713\_edma\_aic23.l67}\]
   \[\text{c6x1x\_edma\_mcbsp.l62}\]
   
**Build and Fix Any Errors**
9. **Build your project and fix any typos/errors.**
   Build the project. You should have a couple of errors concerning PING and pingOrPong (we’ll fix that in a moment). Fix any other typos or errors and rebuild if necessary.
**Examine sioFunctions.c**

10. Add sioFunctions.c to your project and examine its contents.

Add sioFunctions.c to your project and examine the functions in the file. This file was written by the authors of the workshop to encapsulate all of the SIO functions necessary to communicate with the driver. In your own system, you will need similar functions to create and prime the streams for whichever driver you are using.

---

**Note:** 6713 USERS: in sioFunctions.c, change the 2 occurrences of ISRAM to IRAM.

---

The four functions are:

- createStreams( ) – creates the input and output SIO streams hooked to the appropriate DIO, size and attributes
- primeStreams( ) – allocates the dynamic memory buffers for ping and pong. MEM_calloc is the BIOS API that dynamically creates these buffers in any heap.
- splitBuff() – the canned driver hands the processing code interleaved data (LRLR) instead of channel sorting it like we have before. So, a splitBuff() function is required to split the (L)eft and (R)ight data channels.
- joinBuff() – after processing is complete, we need to join the L and R buffers back together.

**Make Other Code Modifications**

11. Add header files to main.c.

Open main.c for editing.

Add the following include files. <sio.h> is the header file that contains the APIs necessary for using streams:

- #include <std.h>
- #include <sio.h>

12. Delete the allocations of the buffers (ping and pong) in main.c.

If you noticed in sioFunctions.c, this file allocates the buffers used by SIO – so, we don’t need to allocate them in main.c anymore.

Delete the 8 global variables creating the rcv and xmt ping/pong buffers.

13. Delete the initialization of the buffers

Delete the for loop and int i in main( ) that zeroes out the buffers. For the time being we’ll just deal with the single buffer of noise.
14. Move the SINE_init and initAlgs calls to the prolog of the TSK in processBuffer().
Cut (don’t delete) the 2 SINE_init statements and the initAlgs statement and paste them just above the while(1) in processBuffer(). This puts them in the prolog of the TSK. main() should now be completely empty.

15. Remove files from the project
There are 4 lines of code creating the source and dest pointers at the beginning of processBuffer(). Delete these 4 allocations and add the following two pointer declarations:

- short *source;
- short *dest;

We only need two pointers at this time because L and R are combined.

16. Delete the if/else construct for pingOrPong in processBuffer().
Delete the entire if/else pingOrPong construct just below SEM_pend in processBuffer().

We no longer need to know whether we are processing ping or pong because the streams handle that protocol for us. We simply issue 4 streams and the driver hands back ping, then pong, then ping, etc.

17. Add the call to the stream functions to create/prime the streams.
Add the following 2 calls in processBuffer() between initAlgs() and while(1){

- createStreams();
- primeStreams();

This code is also in the TSK’s prolog and will only run at initialization.

18. Delete SEM_pend() and replace it with the _reclaim’s and splitBuff().
Delete the SEM_pend() statement and replace it with the following 3 lines:

- SIO_reclaim(inStream,(Ptr*)&source,NULL);
- SIO_reclaim(outStream,(Ptr*)&dest,NULL);
- splitBuff(source,BUFFSIZE,sourceL,sourceR);
19. Add `joinBuff()` and the `_issue’s` at the end of `processBuffer()`.

There are 3 closing braces at the end of `processBuffer()`. In between the 1st and 2nd brace, add the following 3 lines:

- `joinBuff(destL,destR,BUFFSIZE,dest);`
- `SIO_issue(outStream,dest,BUFFSIZE*4,NULL);`
- `SIO_issue(inStream,source,BUFFSIZE*4,NULL);`

20. Declare buffers for the streams.

Add the following 5 lines to the globals area:

- `short sourceL[BUFFSIZE];`
- `short sourceR[BUFFSIZE];`
- `short destL[BUFFSIZE];`
- `short destR[BUFFSIZE];`
- `extern SIO_Handle inStream,outStream;`

**Build and Run the Final Code**


Build and load.

If you get a msg that says that CCS cannot find “divu.asm”, just ignore it. In Debug mode, CCS will scan all of the source files so that you can perform mixed mode (C/asm) debug. The DDK had a file called “divu.asm” that doesn’t exist anymore. This will be fixed in a future build.

Click Run. The music should sound pretty good (other than the fact that you should be sick of listening to the same midi file by now).
Lab12a: Modifying the Driver

Again, there are two main pieces we deal with when developing a system: I/O and processing. In the previous lab, we used the off-the-shelf driver for the 6416/6713DSK and changed our processing code to communicate with that specific driver. The canned driver didn’t do any channel sorting and, likely, most systems will require some kind of channel sorting.

Now that our processing code uses streams to hook to the driver, let’s now MODIFY the existing driver to perform channel sorting. We are going to change the low-level code of the driver to do exactly what we want it to do. We’ve worked with the low-level EDMA configurations before, so we have enough information to proceed.

**Browse the Driver Files**

22. Close/save any projects, close CCS, power cycle the DSK, open CCS again.

23. Browse the driver files.

Using CCS, select:

File → Open

and browse the \audioapp folder. Find the folder called IOM original. These are the original DDK driver files – only renamed with “audioapp” since we’ll be modifying them. Also, the appropriate #include statements have been changed in order to accommodate the name changes of the files. Examine the following screen capture of the IOM original folder for future use and the exact spelling of all the filenames:
Like all I/O mini-drivers, the dsk6416_edma_aic23 driver uses channels and ports. Open the file `dsk6416_edma_aic23_audioapp.c` (or 6713 equivalent) and examine it.

**mdBindDev()** – configures the AIC23 codec as well as the McBSP and binds them to the driver as a port.

**mdCreateChan()** – configures the EDMA channels to transport data from the SIO buffer to the McBSP (output) or from the McBSP to the SIO buffer (input), i.e. between the stream and the port that was created by `mdBindDev()`. The AIC23 and McBSP configurations do not need to be changed. However, the EDMA config structure will need to be modified to perform channel sorting.

**mdSubmitChan()** – submits packets from the SIO stream to the driver to be placed in a queue for linking into the EDMA. Since the EDMA handles the transport of samples to and from the McBSP into and out of SIO stream buffers, the properties of the EDMA channel will need to be modified in order to add de-interleaving to the driver.

**mdDeleteChan()** – you might suspect that this function might be affected, as it’s related to the EDMA. However, this function only frees the EDMA resource to be used by the system, and is not dependent upon the mode in which the EDMA was previously operating – so it remains unchanged.

24. Examine `mdCreateChan()` in `dsk6416_edma_aic23_audioapp.c` (or 6713 equivalent).

Find the `mdCreateChan()` function and examine it.

The `mdCreateChan()` function call of `dsk6416_edma_aic23_audioapp.c` is used to create an initialization structure for the EDMA channel that will be opened as well as configuring a parameter structure to be passed to the generic `C6x1x_edma_mcbsp` driver.

The dsk6416_edma_aic23_audioapp driver of the DDK is built over a more generic EDMA/McBSP driver, whose function calls are located in `C6x1x_edma_mcbsp.c` (which has been renamed to `C6x1x_edma_mcbsp_audioapp.c` for the purposes of this lab). This is a common practice for extending a generic device driver (which only configures the I/O devices of the DSP in question) to a specific device driver which may incorporate external devices, such as the AIC23 codec.
**Add Channel Sorting (Indexing) to the EDMA Config**

25. **Modify the if/then/else construct to use the EDMA’s indexing feature.**

Modify the if/then/else statement that follows the definition of the EDMA configuration structure to:

```c
if (mode == IOM_INPUT) {
    edmaCfg.opt |= EDMA_FMK(OPT, DUM, EDMA_OPT_DUM_IDX);
} else {
    edmaCfg.opt |= EDMA_FMK(OPT, SUM, EDMA_OPT_SUM_IDX);
}
```

This will change both source and destination update modes to use element/frame indexing which is critical for channel sorting.

26. **Examine mdSubmitChan().**

The second thing we need to examine is `mdSubmitChan()`. There is no mdSubmitChan() function call in dsk6416_edma_aic23_audioapp. Instead, the IOM_fxs table links in the mdSubmitChan() function call of the underlying C6x1x_edma_mcbsp driver.

Save and close `dsk6416_edma_aic23_audioapp.c` (or 6713 equivalent).

Open `C6x1x_edma_mcbsp_audioapp.c`.

Scroll to the `mdSubmitChan()` function call in `C6x1x_edma_mcbsp_audioapp.c` (it is near the bottom of the file). There are a number of if() statements testing for various commands. This is how the DIO layer implements such commands as `SIO_flush()` and `SIO_abort()` – they are passed to the mini-driver via the cmd element. At the end of this function is a statement which conditionally links the incoming packet directly into the next EDMA transfer or, if there is already a waiting packet linked in, places it on a queue to be linked later. The `linkpacket()` function is an internal function of the driver (i.e. not exposed via the IOM_fxs table) and is the heart of the `mdSubmitChan()` function call in terms of linking SIO buffers into the EDMA channel.

27. **Declare two new variables in the linkpacket() function.**

In the file `C6x1x_edma_mcbsp_audioapp.c`, scroll to the `linkpacket()` function (about mid way in the file).

Declare two new variables of type int in the declaration phase of the function:

```c
int elemPerChan, elemMaus;
```

They do not need to be initialized.
28. Add code to calculate elemMaus and elemPerChan.

Locate the line in the code which displays the comment:

/* Load the buffer pointer into the EDMA */

Directly before this line of code (and, more importantly, after the `pramPtr` variable has been initialized), insert the following code in order to calculate the number of Minimum Addressable Units (bytes for the C6000) in each element (for us it will be two because we are using shorts, but this code is more general) as well as the number of elements in each channel (again, for us this will be the transfer count divided by two because we have a left and a right channel, but let’s write the driver more generally.)

```c
elemMaus = EDMA_FGETH(pramPtr, OPT, ESIZE) + 1;
if(elemMaus == 3)
    elemMaus = 4;

elemPerChan = (packet->size) / elemMaus / chan->tdmChans;
```

Note: `chan->tdmChans` is an element in the channel object which does not yet exist. We will add this in later and initialize it in the `mdCreateChan()` function call.
29. Set up auto initialization and indexing for EDMA channel sorting.

Locate the following piece of code within the function, a few lines further down:

```c
/*
 * Load the transfer count into the EDMA. Use the ESIZE
 * field of the EDMA job to calculate number of samples.
 */
EDMA_RSETH(pramPtr, CNT, (Uint32) packet->size >>
(2 - EDMA_FGETH(pramPtr, OPT, ESIZE)));
```

Remove or comment out the `EDMA_RSETH` command above and replace it with the following:

```c
EDMA_FSETH(pramPtr, CNT, FRMCNT, elemPerChan - 1);
EDMA_FSETH(pramPtr, CNT, ELECNT, chan->tdmChans);
EDMA_FSETH(pramPtr, RLD, ELERLD, chan->tdmChans);
EDMA_FSETH(pramPtr, IDX, ELEIDX, packet->size / chan->tdmChans);
EDMA_FSETH(pramPtr, IDX, FRMIDX, elemMaus - packet->size * (chan->tdmChans - 1) / chan->tdmChans);
```

30. Declare a new variable called `tdmChans`.

The variable `tdmChans` in the code above, does not currently exist as part of the ChanObj structure (the instance object which is created every time a channel is opened). Previously the channels did not perform channel sorting, so there was no reason to have this parameter in the object.

Find the definition of the ChanObj structure at the beginning of C6x1x_edma_mcbsp_audioapp.c and add the following variable to the structure:

```c
Int tdmChans;
```

Position within the structure doesn’t matter, but for consistency with how the solutions are built, insert it directly after the `tcc` element in the structure.
31. Initialize tdmChans within the mdCreateChan() function.

The value of tdmChans needs to be initialized. The proper place for this is in mdCreateChan(). Scroll to the portion of this function labeled:

```c
/* initialize the channel structure */
```

and insert the following line of code among the other initializations:

```c
chan->tdmChans = params->tdmChans;
```

This will initialize the value of tdmChans within the channel object using the number of channels which is passed from the calling function. Fortunately, tdmChans is already an element in the parameter passing structure of this function call, so no further modifications need to be made.

32. Save and close C6xlx_edma_mcbsp_audioapp.c.

**Build the New Library: myDriver.lib**

33. Create a new project to build the new library.

Create a new project in your \audioapp directory call myDriver. Type in the project name myDriver and then browse to the \audioapp directory so that the .pjt file ends up in the \audioapp directory. Select a Project Type of Library (.lib). Click Finish.

34. Add the driver files to your project.

Add the following files to your project from the \IOM original folder:

All 4 C files from the \IOM original folder (for whichever DSK you are using):

- `dsk6416 Edma_aic23_audioapp.c`
- `dsk6713 Edma_aic23_audioapp.c`
- `dsk6416_aic23_audioapp.c`
- `dsk6713_aic23_audioapp.c`
- `c6xlx Edma_mcbsp_audioapp.c`
- `dsk6416_codec_devParams.c`
- `dsk6713_codec_devParams.c`
35. **Change #include statements.**

Open `dsk6416_codec_devParams.c` (or `6713` equivalent) and change the following:

```c
#include <dsk6416_edma_aic23.h>    to    #include <dsk6416_edma_aic23_audioapp.h>
#include <aic23.h>                               to     #include <aic23_audioapp.h>
```

**Note:** DSK6713 users will use “6713” instead of “6416” above.

Save and close the file.

36. **Add \IOM original to the Include Search Path.**

Select:

```
Project → Build Options → Compiler Tab
```

Click the **Preprocessor** Category. Add the following path to the Include Search Path:

```c
C:\iw6000\labs\audioapp\IOM original
```

Under the **Pre-Define Symbol** box, add the following symbol:

```c
;CHIP_6416  (or ;CHIP_6713 for 6713 users)
```

Click OK.

37. **Build your new library file and fix any errors.**

Build your project and fix any errors. CCS had created a library file for us containing everything we need for the driver to operate called `myDriver.lib`. Close the `myDriver` project.
Lab12a: Modifying the Driver

Remove Old Driver/Source Files and add myDriver.lib

38. Remove the old driver library files and source files from audioapp.pjt.

Open your audioapp project and remove the following libraries and source files from it (or the 6713 equivalent filenames):

- c6x1x_edma_mcbsp.l64
- dsk6416_edma_aic23.l64
- dsk6416_codec_devParams.c

39. Add the new library file (myDriver.lib) to your project from audioapp\Debug folder.

Make the last few Code Adjustments

40. Add back in some control code to main.c.

Open main.c for editing. We’ll have to add back some of the left/right control code, since the buffers are now sorted again.

Add the following 4 lines to the start of processBuffer() (do not delete the declarations for source and dest that are already there):

- short *sourceL;
- short *sourceR;
- short *destL;
- short *destR;

41. Remove splitBuff() and replace it with new code.

Remove the call to splitBuff() and replace it with the following 4 lines of code:

- sourceL = source;
- sourceR = source + BUFFSIZE;
- destL = dest;
- destR = dest + BUFFSIZE;

42. Remove the call to joinBuff().
Lab12a: Modifying the Driver

**Build – Load – Run – Save**

43. **Build, load and run your code.**
   Everything should work perfectly. If not, fix any errors and rebuild/load/run.

44. **Copy project to preserve your solution.**
   Using Windows Explorer, copy the contents of:
   c:\iw6000\labs\audioapp\*.* TO c:\iw6000\labs\lab12

45. **When you're done playing, halt the processor and close CCS.**

You’re done.