Introduction

In this chapter, you will learn how to program the EDMA to perform a transfer of data from one buffer to another.

Learning Objectives

We will learn how to:

- Use the EDMA to transfer data from one location to another
Chapter Topics

Using the EDMA......................................................................................................................... 4-1

Chip Support Library (CSL) ............................................................................................................. 4-3

Enhanced Direct Memory Access (EDMA)....................................................................................... 4-4
  Introduction .................................................................................................................................... 4-4
  Overview ....................................................................................................................................... 4-5
  Definitions .................................................................................................................................... 4-5
  Example ....................................................................................................................................... 4-6
  Programming the EDMA (the traditional way) ............................................................................. 4-7

Using CSL ...................................................................................................................................... 4-7
  Looking more closely at the Config structure? .............................................................................. 4-8

EDMA Events – Triggering the EDMA ............................................................................................ 4-9

Exercise ........................................................................................................................................... 4-10

Lab 4 – Overview ............................................................................................................................ 4-13

Lab 4 ............................................................................................................................................... 4-14
  DMA (vs. EDMA) ......................................................................................................................... 4-20
  EDMA: Channel Controller vs. Transfer Controller ................................................................. 4-22
  QDMA ......................................................................................................................................... 4-23
  DAT (CSL module) ..................................................................................................................... 4-24
  EDMA: Alternate Option Fields ................................................................................................. 4-26
Chip Support Library (CSL)

- C-callable library that supports programming of on-chip peripherals
- Supports peripherals in three ways:
  1. Resource Management (functions)
     - Verify if periph is available
     - “Check-out” a peripheral
  2. Simplifies Configuration
     - Data structures
     - Config functions
  3. Macros improve code readability
- You still have to know what you want the peripherals to do, CSL just simplifies the code and maintenance

The best way to understand CSL is to look at an example...

<table>
<thead>
<tr>
<th>CSL Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache</td>
<td>Cache &amp; internal memory</td>
</tr>
<tr>
<td>CHIP</td>
<td>Specifies device type</td>
</tr>
<tr>
<td>CSL</td>
<td>CSL initialization function</td>
</tr>
<tr>
<td>DAT</td>
<td>Simple block data move</td>
</tr>
<tr>
<td>DMA</td>
<td>DMA (for '0x devices)</td>
</tr>
<tr>
<td>EDMA</td>
<td>Enhanced DMA (for '1x dev)</td>
</tr>
<tr>
<td>EMIF</td>
<td>External Memory I/F</td>
</tr>
<tr>
<td>EMIFA</td>
<td>C64x EMIF’s</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose Bit I/O</td>
</tr>
<tr>
<td>HPI</td>
<td>Host Port Interface</td>
</tr>
<tr>
<td>I2C</td>
<td>I²C Bus Interface</td>
</tr>
<tr>
<td>IRQ</td>
<td>Hardware Interrupts</td>
</tr>
<tr>
<td>McASP</td>
<td>Audio Serial Port</td>
</tr>
<tr>
<td>McBSP</td>
<td>Buffered Serial Port</td>
</tr>
<tr>
<td>PCI</td>
<td>PCI Interface</td>
</tr>
<tr>
<td>PLL</td>
<td>Phase Lock Loop</td>
</tr>
<tr>
<td>PWR</td>
<td>Power Down Modes</td>
</tr>
<tr>
<td>TCP</td>
<td>Turbo Co-Processor</td>
</tr>
<tr>
<td>TIMER</td>
<td>On-chip Timers</td>
</tr>
<tr>
<td>UTOPIA</td>
<td>Utopia Port (ATM)</td>
</tr>
<tr>
<td>VCP</td>
<td>Viterbi Co-Processor</td>
</tr>
<tr>
<td>XBUS</td>
<td>eXpansion Bus</td>
</tr>
</tbody>
</table>

General Procedure for using CSL

1. Include Header Files
   - Library and individual module header files
2. Declare Handle
   - For periph's with multiple resources
3. Define Configuration
   - Create variable of configuration values
4. Open peripheral
   - Reserves resource; returns handle
5. Configure peripheral
   - Applies your configuration to peripheral

Timer Example:

1. `#include <csl.h>`
   `#include <csl_timer.h>`
2. `TIMER_Handle myHandle;`
3. `TIMER_Config myConfig = {control, period, counter};`
4. `myHandle = TIMER_open(TIMER_DEVANY, ...);`
5. `TIMER_config(myHandle, &myConfig);`
Enhanced Direct Memory Access (EDMA)

Introduction

The EDMA is a peripheral that can be set up to copy data from one place to another without the CPU’s intervention. The EDMA can be setup to copy data or program from a source (external/internal memory, or a serial port) to a destination (e.g. internal memory). After this transfer completes, the EDMA can “autoinitalize” itself and perform the same transfer again, or it can be reprogrammed.
Overview

EDMA Overview

- Source
- Destination
- Transfer Count
- Options
- Index
- Count Reload
- Link Addr

EDMA Channel

- C64x has 64 channels
- C67x has 16 channels

EDMA requires transfer parameters
- Most obvious: Src, Dest, Count
- Each channel has also has options for:
  - Data size
  - Channel Priority
  - Autoinit (linking)
  - Inc/dec src & dest addresses

How much does the EDMA move?

Definitions

EDMA - How much to move

- Block
  - Frame 1
  - Frame 2
  - ...
  - Frame M

- Frame
  - Elem 1
  - Elem 2
  - ...
  - Elem N

- Element

Options

Source

Transfer Count

Destination

Index

Cnt Reload

Link Addr

# Frames (M-1)

# Elements (N)

ESIZE

00: 32-bits
01: 16-bits
10: 8-bits
11: rsvd
**Example**

How do we setup the six EDMA parameters registers to transfer 4 byte-wide elements from `loc_8` to `myDest`?

**EDMA Example**

<table>
<thead>
<tr>
<th>8-bit Values</th>
<th>myDest:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3 4 5 6</td>
<td>8 9 10 11 12</td>
</tr>
<tr>
<td>7 8 9 10 11 12</td>
<td>9 10 11</td>
</tr>
<tr>
<td>13 14 15 16 17 18</td>
<td>10</td>
</tr>
<tr>
<td>19 20 21 22 23 24</td>
<td>11</td>
</tr>
<tr>
<td>25 26 27 28 29 30</td>
<td>8 bits</td>
</tr>
</tbody>
</table>

**Addr Update Mode (SUM/DUM)**

- **00**: fixed (no modification)
- **01**: inc by element size
- **10**: dec by element size
- **11**: index

**ESIZE**

- **00**: 32-bits
- **01**: 16-bits
- **10**: 8-bits
- **11**: rsvd

**FS**

- **0**: Off
- **1**: On

**Options**

<table>
<thead>
<tr>
<th>Source</th>
<th>Transfer Count</th>
<th>Destination</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: 10</td>
<td>SUM 01</td>
<td>DUM 01</td>
<td>FS 1</td>
</tr>
</tbody>
</table>

**# Frames (less one)**

| 0 |

**# Elements**

| 4 |

Looking at the EDMA parameters one register at a time:

1. **Options**:
   - **ESIZE** should be self-explanatory based on our previous definitions.
   - **SUM** and **DUM** fields indicate how the source and destination addresses are to be modified between element reads and writes. Since our example above moves 4 consecutive byte elements and writes them to 4 consecutive locations, both **SUM** and **DUM** are set to `inc by element size`. In future chapters, we’ll use other values for them.
   - **Frame Sync (FS)** indicates how much data should be moved whenever the EDMA is triggered to run. In our case, since we want to move the whole frame of data when the CPU starts the EDMA channel, we should set **FS** = 1. Later, when we use the McBSP, we’ll want to change this value so the EDMA only moves one element per trigger event.

2. **Source**: Should have the source address of `loc_8`.

3. **Transfer Counter**: Will have the value 4. Actually, it is 0x 0000 0004.

4. **Destination**: gets the value of `myDest`.

5. **Index**: We’re not using the index capability in this chapter. We will discuss this in chapter 7.

6. **Reload/Linking**: Again, this capability is not used in this chapter. Rather we cover it in the next chapter.
Programming the EDMA (the traditional way)

**Programing the Traditional Way**

<table>
<thead>
<tr>
<th>EDMA Reg Values</th>
<th>Traditional Way to Setup Peripherals</th>
</tr>
</thead>
<tbody>
<tr>
<td>options 0x51200001</td>
<td>1. Determine register field values</td>
</tr>
<tr>
<td>source &amp;loc_8</td>
<td>2. Compute Hex value for register</td>
</tr>
<tr>
<td>count 0x00000004</td>
<td>3. Write hex values to register with C</td>
</tr>
<tr>
<td>dest &amp;myDest</td>
<td></td>
</tr>
<tr>
<td>index 0x00000000</td>
<td></td>
</tr>
<tr>
<td>rld:lnk 0x00000000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Options</th>
<th>Source</th>
<th>Destination</th>
<th>Index</th>
<th>Cnt Reload</th>
<th>Link Addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESIZE</td>
<td>SUM</td>
<td>SUM</td>
<td>FS</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>01</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td># Frames (less one)</td>
<td># Elements</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Is there an easier way to program these registers?

Using CSL

As shown below, we basically want to get the six 32-bit values we calculated for each register into the EDMA channel parameter location.

**CSL – An Easier Way to Program Peripherals**

Chip Support Library (CSL) consists of:

- **Data Types**
  - EDMA_Handle
  - EDMA_Config

- **Functions**
  - EDMA_config()
  - EDMA_setChannel()

- **Macros**
  - EDMA_OPT_RMK()
  - EDMA_SRC_OF()

```c
EDMA_Config myConfig = {
  0x51200001, // options
  &loc_8,     // source
  0x00000004, // count
  &myDest,    // destination
  0x00000000, // index
  0x00000000 // reload:link
};
```

EDMA_config()
Here are the 5 basic steps to accomplishing this using CSL:

### EDMA Programming in 5 Easy Steps

1. **Include the necessary header files**
   ```c
   #include <csl.h>
   #include <csl_edma.h>
   ```

2. **Declare a handle** (will point to an EDMA channel)
   ```c
   EDMA_Handle hMyChan;
   ```

3. **Fill in the config structure** (values to program into EDMA)
   ```c
   EDMA_Config myConfig = {
      EDMA_OPT_RMK(), ...
   };
   ```

4. **Open a channel** (requests any avail channel; and reserves it)
   ```c
   hMyChan = EDMA_open(EDMA_CHA_ANY, EDMA_OPEN_RESET);
   ```

5. **Configure channel** (writes config structure to assigned channel)
   ```c
   EDMA_config(hMyChan, &myConfig);
   ```

### Looking more closely at the Config structure?

You can see we used CSL macros (_RMK and _OF macros) to create the six 32-bit hex values. The beauty of these macros is how easy they are to read and write. This will come in handy when we need to debug our code, or later on when we need to maintain the code.

**EDMA Parameter Values**

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>options</td>
<td>0x51200001</td>
</tr>
<tr>
<td>source</td>
<td>&amp;loc_8</td>
</tr>
<tr>
<td>count</td>
<td>0x00000004</td>
</tr>
<tr>
<td>dest</td>
<td>&amp;myDest</td>
</tr>
<tr>
<td>index</td>
<td>0x00000000</td>
</tr>
<tr>
<td>rldcnt:lnk</td>
<td>0x00000000</td>
</tr>
</tbody>
</table>

- **_RMK** (register make) creates a single hex value from option symbols you select
- **_OF** macro performs any needed casting (and provides visual consistency)
- Highlighted in **BLUE** are the options discussed thus far (esize, sum, dum, fs, src, cnt, dst)

**EDMA_Config myConfig = {**

```c
EDMA_OPT_RMK(
   EDMA_OPT_PRI_LOW,
   EDMA_OPT_ESIZE_8BIT,
   EDMA_OPT_2DS_NO,
   EDMA_OPT_SUM_INC,
   EDMA_OPT_2DD_NO,
   EDMA_OPT_DUM_INC,
   EDMA_OPT_TCINT_YES,
   EDMA_OPT_TCC_OF(5),
   EDMA_OPT_LINK_NO,
   EDMA_OPT_FS_YES
),
   EDMA_SRC_OF(loc_8),
   EDMA_CNT_OF(0x00000004),
   EDMA_DST_OF(myDest),
   EDMA_IDX_OF(0),
   EDMA_RLD_OF(0)
);
```
EDMA Events – Triggering the EDMA

How do you trigger an EDMA channel to run, that is, start copying a block of memory?
- Channels must receive a start event in order to run
- Set an event by writing to the EDMA’s Event Register (ER)

Conveniently, a CSL function can set the ER bit for us:

EDMA_setChannel(hMyChan)

<table>
<thead>
<tr>
<th>EDMA Event Input ER</th>
<th>EDMA Channels</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>...</td>
</tr>
</tbody>
</table>

In Chapter 6 we will show how to use interrupt events to trigger the EDMA. This will come in handy when we use the McBSP to tell the EDMA when to transfer a value to it, or when to pick up a value from its receive register.
Exercise

Exercise 1 (Takes 20 Minutes)

- Using the space provided in Student Notes, write the code to initialize the EDMA.
- Here’s a few Hints:
  - Follow the 5 steps we just discussed for writing CSL code
  - Here are the config values for options not yet discussed:
    - Low priority (PRI)
    - Single dimensional source & dest (2DS, 2DD)
    - Set TCC to 0
    - TCINT to off
    - LINK to no
    - Set reload and index values to 0

Exercise 1, Steps 1-2

1. Specify the appropriate include file(s):

2. Declare an EDMA handle named \textit{hEdma}.

3. Fill out the values for \textit{gEdmaConfig} so that it moves the contents of \textit{gBuf0} to \textit{gBuf1}.
Exercise 1, Step 3: EDMA_Config

```c
EDMA_Config gEdmaConfig = {
    EDMA_OPT_RMK(
        EDMA_OPT_PRI_ , // Priority?
        EDMA_OPT_ESIZE_ , // Element size?
        EDMA_OPT_2DS_  , // Is it a 2 dimensional src?
        EDMA_OPT_SUM_  , // Src update mode?
        EDMA_OPT_2DD_  , // Is it a 2 dimensional dst?
        EDMA_OPT_DUM_  , // Dest update mode?
        EDMA_OPT_TCINT_ , // Cause EDMA interrupt?
        EDMA_OPT_TCC_OF( ) , // Transfer complete code?
        EDMA_OPT_LINK_ , // Enable linking (autoinit)?
        EDMA_OPT_FS_   , // Use frame sync?
    ),
    EDMA_SRC_OF( ) , // src address?
    EDMA_CNT_OF( ) , // Count = buffer size
    EDMA_DST_OF( ) , // dest address?
    EDMA_IDX_OF( 0 ) , // frame/element index value?
    EDMA_RLD_OF( 0 ) // reload
};
```

Exercise 1, Steps 4-6

4. Request any available EDMA channel from CSL to perform the transfer:

5. Configure the EDMA channel you opened:

6. How would you trigger this channel to run?
*** this page is VERY blank ***
Lab 4 – Overview

Lab 4 – Programming the EDMA

Goals:
1. CPU generates 32 sine values into gBuf0
2. EDMA transfers 32 elements from gBuf0 to gBuf1

Goals of the lab:

- To use CSL to set up the EDMA for copying buf0 to buf1. This will be done programmatically as discussed in the material.
Lab 4

Understanding Coding/Naming Conventions

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

2. Open main.c

You can open a file by double-clicking on it in the Project View window. You may have to expand the source files folder to find it.

3. Review coding conventions.
   - Take a look at the prototypes and global variables. You’ll notice that each uses titleCase, meaning that the first word is lower case and the concatenated second word has the first character capitalized. Titlecase is suggested for user-defined functions as well as global variables. Example: gBuf.
   - Constants are entirely capitalized (no underscores) – notice the constant BUFFSIZE.
   - CSL Functions: the CSL API uses a specific naming convention. The generic form of a CSL function looks like: MOD_function( ). For example, when using the EDMA module, its open function appears as:

     EDMA_open( )

     EDMA is capitalized because it is the module name. The function (such as “config”, “intEnable”, or “open”) is in titleCase and separated by an underscore.
   - CSL Data Types: take the generic form MOD_DataType. That is, along with the module name, the type is separated by an underscore. Also, notice that titleCase is used here, too, with one exception; the first letter after the underscore is Capitalized.
   - To distinguish global variables from locals, we will use a small “g” prefix. Globals do not use underscores. (The small “g” is not required, but it’s a common practice.)
   - Handles (or pointers to our resources) will normally begin with a lower case prefix of “h”. (Not required, but again, it’s a common practice.)

These conventions match TI’s software development guidelines, and are similar to Microsoft’s naming conventions. For the most part, understanding and using these conventions will help clarify everyone’s code. Hopefully they’ll quickly become second nature.

Add a Second Buffer to the System

4. In main.c, add a second buffer for use as the EDMA destination

Per the system diagram, we need to create another buffer to be used as the destination of our EDMA transfer. We currently have just one buffer (gBuf) that was used to hold the sine values that we graphed in lab2.

Change the name of the current buffer to gBuf0 (search and replace all occurrences).

Declare a second global buffer, the same size as gBuf0, and name it gBuf1. gBuf0 will be the source of our EDMA transfer and gBuf1 will be the destination.
**Initialize the EDMA via CSL**

Our goal is to set up an EDMA channel to copy one buffer to another. The following steps will get the EDMA to transfer just once. Later in the lab, we’ll add the autoinitiation capability.

We will be using the Chip Support Library (CSL) to perform setup and initialization (most of the code you’ll need comes from the paper exercise). Refer to the 5-step CSL procedure for programming the EDMA from the discussion – and the paper exercise you did just before the lab. We’re going to follow the first 5 steps of the procedure and save the autoinit step until later.

If you need additional help, you can refer to the CSL Reference Manual (SPRU401) under Help → Users Manuals in CCS.

We are going to put all of the code that initializes the EDMA into a separate file to keep it all nice and organized. We have provided a simple file to start with called edma.c.

5. **Add edma.c to your project**
   The file, edma.c, is located in c:\iw6000\labs\audioapp\.

6. **Open edma.c and inspect it**
   There's not much exciting here right now, but we'll add a lot of code to this file by the day's end.

   We're going to add code to this file to initialize and configure the EDMA to do a transfer. We will basically be following the 5 step procedure that we outlined earlier. Please refer back to this procedure to help you keep track of what you are doing.

7. **Add the two header files necessary for CSL and the EDMA APIs (Step 1 of 5)**
   In edma.c, our code will reference the functions and data-structures from these libraries (<csl.h> and <csl_edma.h>). Make sure you add them in the correct order. These should be the first #include statements in main.c

8. **Declare the EDMA Handle in edma.c (Step 2 of 5)**
   Add a global EDMA handle, named hEdma, to the global variables area of your program in edma.c. We will use this handle to point to and initialize the channel registers.

9. **Copy the Starter EDMA Config Structure**
   Rather than typing the whole structure from scratch, we have provided a structure for you that is almost completely filled in (see comments at the top of the file).

   Copy the structure from the commented area to the global variables area of edma.c just beneath the declaration for the EDMA handle. Change the name of the structure from variableName to gEdmaConfig.

   Notice: The TYPE definition EDMA_Config uses an uppercase C for “C”onfig. This is the naming standard for CSL’s typedefs, i.e. MOD_Config, where MOD is the module name EDMA. (As opposed to the “config” function that uses a small “c”.)
10. Fill in the OPTIONS register of EDMA Config Structure (Step 3 of 5)

This code configures the EDMA using CSL’s _RMK and _OF macros. The _RMK macro is used to set up an EDMA Options register value. We use the _OF macros to initialize the other five EDMA registers in the config structure.

Fill in the structure based upon the following requirements for the EDMA transfer.

**Hint:** If you need some help filling in the values, you may find some hints by accessing Help → Users Manuals and looking at the CSL Reference Guide (SPRU401).

Search the .pdf file for **EDMA_OPT_field_symval.** You can find tips here on how to fill in the config structure.

Set the Options (OPT) register using the _RMK macro as follows:

- Low Priority
- 16 bit Elements
- 1-dimensional source
- Source Increments
- 1-dimensional destination
- Destination Increments
- Do NOT cause a transfer complete interrupt (later in the lab, we’ll change this)
- Set a transfer complete code of 0 (we will change this using EDMA_intAlloc later…)

- Set the transfer complete code upper bits (TCCM) to the default value
- Set the cause alternate transfer complete interrupt to default
- Set the value of the alternate transfer complete code to the default value
- Set the peripheral device transfer source to default
- Set the peripheral device transfer destination to default

- Disable linking of event parameters (we’ll change this in order to auto-initialize)
- Use Frame Synchronization

**Note:** If you are using the C67x, make sure to comment out the four fields that are specific to the C64x.
11. Now, set the other registers as follows:

- **Source** is `gBuf0`.
- Set **Count** to the buffer’s size. Use the defined constant at the top of the file.
- **Destination** is `gBuf1`.
- No **Index** needed, set to 0. Not used unless the DUM and SUM use IDX (index).
- Set **Reload** (and **Link**) to 0, for now. We’ll change this dynamically in the code.

12. Add external references for `gBuf0` and `gBuf1`

Since `gBuf0` and `gBuf1` are declared in main.c, we need to add external references to them so that the code generation tools know how to go find them. The easiest way to do this is to copy the code that creates the two buffers from main.c to edma.c and add the C keyword `extern` in front of them.

**Initializing the EDMA**

13. Add code to the `initEdma` function in edma.c

In `edma.c`, find the function called `initEdma()`. Notice that this function is already prototyped for you.

14. Inside `initEdma()`, open the EDMA channel (Step 4 of 5 Easy Steps)

Inside this function, add a call to the CSL function that opens an EDMA channel. Use the handle that we created earlier. Pick *any* channel (hint) and reset the channel when it’s opened.

15. Configure the EDMA channel (Step 5 of 5 Easy Steps)

Next, use a CSL function to configure the channel with the Config structure you created earlier.

You have now completed the `initEdma()` function.

**Modifying main( )**

16. Add `initEdma()` call to main( ) in main.c

Now that the function is created, we need to call it. Add a call to `initEdma()` in the main() function just below the call to `SINE_init(…)`.

17. Include `edma.h` in main.c

Since we are calling a function that is located in another file, we need to reference it in the calling file, main.c. We have provided a header file to do this for you, edma.h. Feel free to open edma.h and check out what it has in it.
18. Tell the EDMA Channel to Transfer the Buffer

Call EDMA_setChannel( ) after SINE_blockFill( ) in main.c (and before the while loop) to initiate an EDMA transfer for the hEdma channel. This function was discussed toward the end of the chapter (as part of the EDMA ISR topic).

We are using this function in place of a synchronization (i.e. trigger) event. The next lab uses the McBSP to trigger the EDMA transfers. Which is a lot more fun.

19. Add CSL header files to main.c

Since we are using a CSL function for the EDMA in main.c (EDMA_setChannel()), we need to add the two necessary header files to main.c. Add #include statements for <csl.h> and <csl_edma.h> to main.c. Make sure to add these files in this order and put them above the other header files in main.c.

Build and Run Code to Check Operation

20. Set the DSP clock speed for DSK6416 (1000MHz), DSK6713 (225MHz)

Open audioapp.cdb. Click on the + next to System. Right click on Global Settings and select Properties. Change the DSP Speed to 1000MHz for the 6416DSK and 225MHz for the 6713DSK. Click OK and close/save the .cdb file.

21. Add CHIP_6416 or CHIP_6713 to the Project → Build Options

The CSL code that we added to initialize the EDMA needs to know what chip we are using. It uses this information to decide how many EDMA channels we have, which peripherals we have, etc.

To give it this information, we need to define a build time constant. Select Project → Build Options. Under Category, select Preprocessor. Next to the Pre-Define Symbol (-d) text box, add: ;CHIP_6416 or ;CHIP_6713 depending on your target (as shown below).

Your build options should now look something like this:
22. **Build/load your code and fix any errors.**

23. **Run your code**

   Looking at main(), you’ll notice that all we are doing is:
   - Initializing the EDMA channel
   - Filling the source buffer (gBuf0); then
   - Telling the EDMA to transfer that buffer to the destination (gBuf1)

   Afterwards, the code drops into the while loop and does nothing.

   Our main intent is to see if the EDMA config structure is set up properly and that the EDMA actually does one transfer. Once this is working, the next step is to cause the EDMA transfer repeatedly. We’ll do this by adding a hardware interrupt to our system and configuring our channel for auto-initialization in the next chapter.

24. **Halt the processor and graph gBuf0 and gBuf1.**

   After halting the CPU, graph (as you did in lab 2) the source buffer (gBuf0) and the destination buffer (gBuf1) to make sure they match. If not, debug your code and re-verify. Here’s a reminder for how to do the graphs:
   
   View → Graph → Time/Frequency

   Modify the following values:
   - Graph Title   gBuf0
   - Start Address  gBuf0
   - Acquisition Buffer Size  32
   - Display Data Size  32
   - DSP Data Type  16-bit signed integer
   - Sampling Rate  8000

   Click OK when finished.

   To do the graph for gBuf1, follow the same steps except change the graph title and start address to gBuf1.

   Once the graphs match, you have successfully programmed the EDMA to transfer data from one buffer to another.

25. **Copy project to preserve your solution.**

   Using Windows Explorer, copy the contents of:
   
   c:\iw6000\labs\audioapp\*.*  TO  c:\iw6000\labs\lab4

   You’re Done
Optional Topics

DMA (vs. EDMA)

- 4 Channels with fixed priority
- 1 *extra* channel dedicated to HPI
- Global registers shared by all channels

**Single Frame Transfer**

8-bit Pixels

```
 1 2 3 4 5 6
 7 8 9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
```

(Source: mem_8)

h_line: 8 9 10 11

8 bits

**ESIZE**
00: 32-bits
01: 16-bits
10: 8-bits
11: rsvd

**SRC/DST DIR**
00: no modification
01: inc by element size
10: dec by element size
11: index

**START**
00: Stop
01: Start
10: Pause

---

Lab 4

Technical Training Organization

C6000 Integration Workshop - Using the EDMA
## DMA / EDMA Comparison

<table>
<thead>
<tr>
<th>Features:</th>
<th>DMA</th>
<th>C67x EDMA</th>
<th>C64x EDMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channels</td>
<td>4 channels</td>
<td>16 channels</td>
<td>64 channels</td>
</tr>
<tr>
<td></td>
<td>+ 1 for HPI</td>
<td>+ 1 for HPI</td>
<td>+ 1 for HPI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ Q-DMA</td>
<td>+ Q-DMA</td>
</tr>
<tr>
<td>Sync</td>
<td>◆ element</td>
<td>◆ element</td>
<td>◆ element</td>
</tr>
<tr>
<td></td>
<td>◆ frame</td>
<td>◆ frame</td>
<td>◆ frame</td>
</tr>
<tr>
<td></td>
<td>◆ 2D (block)</td>
<td>◆ 2D (block)</td>
<td></td>
</tr>
<tr>
<td>Priority</td>
<td>4 fixed levels</td>
<td>2 prog levels</td>
<td>4 prog levels</td>
</tr>
</tbody>
</table>
EDMA: Channel Controller vs. Transfer Controller

We often describe the EDMA as a traditional DMA peripheral. While this description works conceptually, the EDMA is actually made of two blocks:

- **EDMA channel controller**: Reads transfer parameters from channel location in parameter RAM and sends request to Transfer Controller.
- **Transfer Controller**: Moves blocks of data as requested.

### EDMA Channel Controller vs. EDMA Transfer Controller

**EDMA**
- **EDMA Channel Controller**
  - Channels:
  - Reloads:
  - 2K parameter RAM
- **Transfer Controller**
  - **Transfer engine**
  - Takes move requests from EDMA, QDMA, and cache

**Transfer Controller**
- **Program Cache**
- **CPU**
- **Data Cache**
- **L2 SRAM**
- **EDMA Transfer Controller**
- McBSP’s
- HPI
- EMIF
- Etc.
QDMA

**EDMA Channel Controller**
- Channels:
- Reloads:
  - 2K parameter RAM

**QDMA**
- Options
  - Source
  - Count
  - Destination
  - Index

**Transfer Controller**
- Transfer engine
- Takes move requests from EDMA, QDMA, and cache

**QDMA**
- Sends a single block transfer request
- Starts (i.e. sends transfer request) when last register is written to; it doesn't work with (interrupt) events
- No auto-init, therefore it does not have Reload:Linking register (this feature is discussed in the next chapter)
- Transfer request goes directly to the Transfer Controller
## DAT (CSL module)

<table>
<thead>
<tr>
<th>DAT</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Block copy module</strong></td>
</tr>
<tr>
<td>• Simply moves (or fills) a block of data</td>
</tr>
<tr>
<td>• No sync or ints are provided</td>
</tr>
<tr>
<td><strong>DAT Functions</strong></td>
</tr>
<tr>
<td>• DAT_busy</td>
</tr>
<tr>
<td>• DAT_close</td>
</tr>
<tr>
<td>• DAT_copy</td>
</tr>
<tr>
<td>• DAT_fill</td>
</tr>
<tr>
<td>• DAT_open</td>
</tr>
<tr>
<td>• DAT_setPriority</td>
</tr>
<tr>
<td>• DAT_wait</td>
</tr>
<tr>
<td>• DAT_copy2d</td>
</tr>
</tbody>
</table>

**DAT is device independent** |
- Implemented for all C5000/C6000 devices |
- It uses whatever DMA capability is available |
- Uses QDMA, when available
void myDat(void) {
    #define BUFFSZ 4096
    static Uint8 BuffA[BUFFSZ]; Uint8 BuffB[BUFFSZ];
    Uint32 FillValue, XfrId;

    DAT_open(DAT_CHAANY, DAT_PRI_HIGH);

    FillValue = 0x00C0FFEE; /* Set the fill value */
    XfrId = DAT_fill(BuffA, BUFFSZ, &FillValue); /* Perform the fill operation */
    DAT_wait(XfrId); /* Wait for completion */

    XfrId = DAT_copy(BuffA, BuffB, BUFFSZ); /* copy A -> B */
    ... if (DAT_busy(XfrId) == 0) then /* Check if copy completed, yet */
        printf("Not done yet");
    ...
    DAT_close();
}
# EDMA: Alternate Option Fields

**EDMA “Alternate” Options (C64x only)**

```c
EDMA_Config gEdmaConfig = {
    EDMA_OPT_RMK( ... 
    //EDMA_OPT_TCCM_DEFAULT, // Transfer Complete Code Upper Bits (64x only)
    //EDMA_OPT_ATCINT_DEFAULT, // Alternate TCC Interrupt (c64x only)
    //EDMA_OPT_ATCC_DEFAULT, // Alternate Transfer Complete Code (c64x only)
    //EDMA_OPT_PDTS_DEFAULT, // Peripheral Device Transfer Source (c64x only)
    //EDMA_OPT_PDTD_DEFAULT, // Peripheral Device Transfer Dest (c64x only)
    ... 
```

PDTS/PDTD allows EDMA to use the EMIF’s PDT capability, that is it allows the EDMA to transfer directly to/from a peripheral to external memory.

**Alternate Transfer Chaining**

- TCCM, ATCINT, ATCC
- Discussed as an optional topic in Chapter 5

---

## Solutions to Paper Exercises

### Exercise 1, Steps 1-2

1. Specify the appropriate include file(s):
   ```c
   #include <csl.h>
   #include <csl_edma.h>
   #include “sine.h”
   ```

2. Declare an EDMA handle named `hEdma`.
   ```c
   EDMA_Handle hEdma;
   ```

3. Fill out the values for `gEdmaConfig` so that it moves the contents of `gBuf0` to `gBuf1`.
   ```c
   see next slide ... 
   ```
Exercise 1, Step 3: EDMA_Config

```c
EDMA_Config gEdmaConfig = {
    EDMA_OPT_RMK(  
        EDMA_OPT_PRI_LOW, // Priority?  
        EDMA_OPT_ESIZE_16BIT, // Element size?  
        EDMA_OPT_2DS_NO, // Is it a 2 dimensional src?  
        EDMA_OPT_SUM_INC, // Src update mode?  
        EDMA_OPT_2DD_NO, // Is it a 2 dimensional dst?  
        EDMA_OPT_DUM_INC, // Dest update mode?  
        EDMA_OPT_TCC_OF( 0 ), // Transfer complete code?  
        EDMA_OPT_LINK_NO, // Cause EDMA interrupt?  
        EDMA_OPT_TCC_OF( 0 ), // Transfer complete code?  
        EDMA_OPT_FS_YES // Use frame sync?  
    ),
    EDMA_SRC_OF( gBuf0 ), // src address?  
    EDMA_CNT_OF( BUFFSIZE ), // Count = buffer size  
    EDMA_DST_OF( gBuf1 ), // dest address?  
    EDMA_IDX_OF( 0 ), // frame/element index value?  
    EDMA_RLD_OF( 0 ) // reload  
};
```

Exercise 1, Steps 4-6

4. Request any available EDMA channel from CSL to perform the transfer:

   ```c
   hEdma = EDMA_open(EDMA_CHA_ANY, EDMA_OPEN_RESET);
   ```

5. Configure the EDMA channel you opened:

   ```c
   EDMA_config(hEdma, &gEdmaConfig);
   ```

6. How would you trigger this channel to run?

   ```c
   EDMA_setChannel(hEdma);
   ```