Hardware Interrupts (HWI)

Introduction

In this chapter, we'll see what the EDMA can do when it finishes a transfer. We will discuss how the CPU’s interrupts work, how to configure the EDMA to interrupt the CPU at the end of a transfer, and how to configure the EDMA to auto-initialize.

Learning Objectives

1. CPU writes buffer with sine values
2. EDMA copies values from one buffer to another
3. When the EDMA transfer is complete
   - EDMA signals CPU to refill the buffer
   - EDMA re-initializes itself

Outline

- Hardware Interrupts (HWI)
  - Generating Interrupt with the EDMA
  - Enabling & Responding to HWI's
- EDMA Auto-Initialization
- Exercise
- Lab
- Optional Topics
Chapter Topics

Hardware Interrupts (HWI) ................................................................. 5-1

EDMA Interrupt Generation ............................................................... 5-3

Hardware Interrupts (HWIs) ............................................................... 5-6
  How do they work? ............................................................................ 5-7
  Interrupt Service Routines (ISRs) .................................................... 5-9
  Configuring HWI Objects ............................................................... 5-11
  Interrupt Initialization ................................................................. 5-13

EDMA Interrupt Dispatcher ............................................................... 5-14

EDMA Auto-Initialization ................................................................. 5-17
  6 Steps to Auto-Initialization ......................................................... 5-19

Summary .......................................................................................... 5-21
  Configuring EDMA Interrupts in 6 Easy Steps ............................... 5-22
  The EDMA ISR ............................................................................. 5-24
  The EDMA's CSL Functions ......................................................... 5-25

Exercise ......................................................................................... 5-26

Lab 5 ............................................................................................... 5-29
  Overview ..................................................................................... 5-29
  Lab Overview ............................................................................... 5-30

Optional Topics ............................................................................. 5-36
  Saving Context in HWIs .............................................................. 5-36
  Interrupts and the DMA ............................................................. 5-38
  EDMA Channel Chaining ............................................................ 5-41
  Additional HWI Topics .............................................................. 5-42
  Exercise 1 .................................................................................... 5-50
  Exercise 2 .................................................................................... 5-51
EDMA Interrupt Generation

EDMA channels can be configured to send interrupt signals to the CPU when they finish a transfer.

Generate EDMA Interrupt

What causes an EDMA channel to send an interrupt?
The channel’s “Transfer Count” going to zero

You can prevent (or enable) the channel from sending an interrupt...

The TCINT bit of each channel turns EDMA interrupt generation on and off.

Generate EDMA Interrupt (TCINT)

◆ Channel’s Options register allows you to enable/disable interrupt generation

Similar to the CPU’s interrupt recognition, the EDMA has flag/enable bits...
The CIPR register records which enabled (TCINT set) channels have finished. The CIER register controls which CIPR bits send an interrupt to the CPU.

**Generate EDMA Interrupt (TCINT)**

<table>
<thead>
<tr>
<th>Channel #</th>
<th>Options</th>
<th>CIPR</th>
<th>CIER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TCINT=0</td>
<td>0</td>
<td>CIER0 = 0</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>CIER1 = 0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>1</td>
<td>CIER8 = 0</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>0</td>
<td>CIER15 = 0</td>
</tr>
</tbody>
</table>

- The Channel Interrupt Pending Register (CIPR) records that an EDMA transfer complete has occurred.

**Generate EDMA Interrupt (TCC)**

<table>
<thead>
<tr>
<th>Channel #</th>
<th>Options</th>
<th>CIPR</th>
<th>CIER</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TCINT=1</td>
<td>0</td>
<td>CIER0 = 0</td>
</tr>
<tr>
<td>1</td>
<td>TCINT=0</td>
<td>1</td>
<td>CIER1 = 0</td>
</tr>
<tr>
<td>...</td>
<td>TCINT=1</td>
<td>1</td>
<td>CIER8 = 0</td>
</tr>
<tr>
<td>15</td>
<td>TCINT=0</td>
<td>0</td>
<td>CIER15 = 0</td>
</tr>
</tbody>
</table>

- Any channel can set any CIPR bit.
- Value in TCC bit field selects CIPR bit that will get set.
- Setting any CIPR bit allows for EDMA channel chaining (described later in Optional Topics).

The TCC field in the Options Register allows each channel to set any CIPR bit.
The Chip Support Library (CSL) has functions for manipulating the various bits used by the EDMA to control interrupt generation.

Passing a “-1” to EDMA_intAlloc() allocates any available CIPR bit, as opposed to allocating a specific bit.

For now, allocating any CIPR bit is OK. When using EDMA Channel Chaining, though, a specific CIPR bit must be used. In these cases, it is either a good idea to allocate the specific CIPR bits first, or plan out which channels will use which bits. Then use the EDMA_intAlloc() function to officially allocate (i.e. reserve) each CIPR bit. (Note, Channel Chaining is briefly discussed at the end of this chapter as an optional topic.)
Hardware Interrupts (HWIs)

If the EDMA can generate an interrupt, what has to be done in order for the CPU to recognize and respond to this interrupt? What is an interrupt anyway?

1. First, we examined how the EDMA generates interrupts to the CPU
2. Next, we explore how CPU interrupts (HWI's) work
How do they work?

Interrupts are very important in DSP systems. They allow the CPU to interact with the outside world.

**How do Interrupts Work?**

1. An interrupt occurs
   - EDMA
   - HPI
   - Timers
   - Ext pins
   - Etc.

2. Sets flag in IFR register
The IER register and the GIE bit in the Control Status Register allow users to enable and disable interrupts.

### Interrupting the CPU

<table>
<thead>
<tr>
<th>IFR</th>
<th>IER “Individual Enable”</th>
<th>GIE “Master Enable”</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDMAINT</td>
<td>Interrupt Flag</td>
<td>'C6000 CPU</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **EDMAINT Interrupt Flag Reg (IFR)**
  - bit set when int occurs
- **Interrupt Enable Reg (IER)**
  - enables individual int’s
  - \(\text{IRQ\_enable(IRQ\_EVT\_XINT2)}\)
  - \(\text{IRQ\_enable(IRQ\_EVT\_EDMAINT)}\)
- **Global Interrupt Enable (GIE) bit in Control Status Reg**
  - enables all IER-enabled interrupts
  - \(\text{IRQ\_globalEnable()}\)
  - \(\text{IRQ\_globalDisable()}\)

---

Here is a nice summary of how CPU interrupts work on the C6000.

### How do Interrupts Work?

1. **An interrupt occurs**
   - DMA
   - HPI
   - Timers
   - Ext pins
   - Etc.

2. **Sets flag in IFR register**

3. **CPU acknowledges interrupt and ...**
   - Stops what it is doing
   - Turn off interrupts globally
   - Clears flag in register
   - Saves return-to location
   - Determines which interrupt
   - Calls ISR

4. **ISR (Interrupt Service Routine)**
   - Saves context of system*
   - Runs your interrupt code (ISR)
   - Restores context of system*
   - Continues where left off*

* Must be done in user code, unless you choose to use the DSP/BIOS HWI dispatcher

Note, the DSP/BIOS HWI Dispatcher is discussed later (on page 5-12).
Interrupt Service Routines (ISRs)

The Interrupt Service Routine is the function that gets called when an interrupt occurs. The ISR contains the instructions for what needs to be done when a given interrupt occurs.

Please fill-in the code that needs to be run in our system, when the EDMA finishes transferring a block of sine wave values:

```c
void edmaHWI()
{

}
```

**Hint:** Just fill in the functions that need to run. Don’t worry about the arguments, for now. Though, you’ll need to come up with the function arguments when coding the ISR in the upcoming lab.
The ISR should perform two actions:

- Refill the buffer with new sine values.
- Trigger the EDMA to run again, thus moving the new sine values.

**Interrupt Service Routine**

*What do we want to happen when the EDMA interrupts the CPU?*

```c
void edmaHWI()
{
    SINE_blockFill();
    EDMA_setChannel();
}
```
Configuring HWI Objects

C6000 interrupts are very configurable; and thus, very flexible and powerful.

- **C6000 has 16 hardware interrupts (HWI)**
- When multiple interrupts are pending, they are serviced in the order shown
- Each interrupt object is associated with an:
  - Interrupt source
  - Interrupt service routine

Using the DSP/BIOS Configuration Tool, it is easy to configure each HWI object’s **Interrupt Source** and **ISR function**. These settings can also be handled via CSL functions, but the Config Tool is much easier to use.

**Note:** Since the Config Tool expects an assembly label, you need to place an “_” (underscore) in front of any C function name that is used – as shown above.
The HWI object allows you to select the HWI dispatcher. This is found on the 2nd tab:

**Configure HWI Object**

The HWI Interrupt Dispatcher takes care of saving and restoring the context of the ISR.

**HWI Interrupt Dispatcher**

The HWI dispatcher is plugged into the interrupt vector table. It saves the necessary CPU context, and calls the function specified by the associated HWI object. Additionally, it allows the use of DSP/BIOS scheduling functions by preventing the scheduler from running while an HWI ISR is active.
Interrupt Initialization

Several concepts have been introduced up to this point. Let's take a moment to make sure that you understand how to setup the CPU to receive a given interrupt.

Enable CPU Interrupts

◆ **Exercise:** Fill in the lines of code required to enable the EDMAINT hardware interrupt:

```c
void initHWI(void)
{

}
```
EDMA Interrupt Dispatcher

The EDMA Interrupt Dispatcher, which is completely different from the HWI Dispatcher that we talked about earlier, helps us solve a very basic problem.

### EDMA ISR Problem

- How many EDMA channels? 16 (or 64)
- How many EDMA interrupt service routines could exist? 16 (or 64)
- How many EDMA interrupts? 1

Since there is only one EDMA ISR, the CIPR bits can be used to tell which EDMA channels have actually completed transfers and need to be serviced.

### Which Channel?

Since there is only one EDMA interrupt to the CPU, how does the CPU know which channel caused it?

Two methods:
1. Test each CIPR bit using: `EDMA_intTest(bit #)`
2. Automate testing each CIPR bit using `EDMA Interrupt Dispatcher`
To use the EDMA Interrupt Dispatcher, the EDMA interrupt vector needs to be setup to call the dispatcher.

- Previously, our EDMAINT vectored directly to our Interrupt Service Routine
- Can you think of a problem this might create?
  - What if two different EDMA channels cause an interrupt?
  - Do you want all channels to use the same ISR? (Not very convenient)
- To solve this problem, CSL provides a simple **EDMA Interrupt Dispatcher**

The EDMA Interrupt Dispatcher figures out what channels have finished and calls the function that has been associated with each CIPR bit that’s been set.

The source code for the EDMA dispatcher is provided (as is the source code for all of CSL). Upon examination you’ll find that the EDMA dispatcher reads both the CIPR and CIER registers. It then calls a function for any CIPR bit = 1, whose respective CIER bit is also = 1.

How do we know which function is associated with which channel (i.e. CIPR bit)?
The EDMA Interrupt Dispatcher needs to be told what function to call for each of the CIPR bits that we want to cause an interrupt to the CPU. This is referred to as "hooking" a function into the EDMA Interrupt Dispatcher. And thus, the CSL function is called EDMA_intHook().

The EDMA_intHook function has two arguments, the CIPR bit number and the function to be called when it’s set by a completed EDMA channel.

For simplicity, the example shown above specifies a CIPR bit with just the number “8”. Most likely, though, you will use a variable to represent the CIPR bit number. A variable is a better choice as it can be set when using the EDMA_intAlloc() function to reserve a CIPR bit for an EDMA channel.
EDMA Auto-Initialization

Interrupting the CPU is nice for keeping the EDMA and CPU in sync. This allows the CPU to know when to perform an action based upon EDMA activity, such as refilling the sine-wave buffer.

But, how does the EDMA channel get reprogrammed to perform another block transfer?

The CPU could go off and program the EDMA for a new transfer during the ISR. Are there any negatives to this? Yes, it takes valuable CPU time. What if we could tell the EDMA what job to do next; that is, in advance?

When the Transfer is Complete …

<table>
<thead>
<tr>
<th>Options</th>
<th>Source</th>
<th>Transfer Count</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
<td>0x5</td>
<td>0</td>
<td>0x15</td>
</tr>
<tr>
<td>Count Reload</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Link Addr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

When TC (transfer count) reaches 0:

- Channel stops moving data
- EDMA can send interrupt to CPU (just discussed)
- Which registers have changed since EDMA was started?
  - Source, Destination, Count

Notice that the EDMA channel registers actually change as the transfer takes place. The source address, destination address, and the transfer count are good examples of values that may change as the transfer occurs. If these values have changed, they can't be used to do the same transfer again without being refreshed.
The EDMA has a set of "reload" registers that can be configured like an EDMA channel. Each channel can be linked to a reload set of registers. In this way, the values in the reload registers can be used to "reload" the “used” EDMA channel.

When the Transfer is Completes …

Essentially, the EDMA has its own 2KB parameter RAM split between channels & reloads

When TC (transfer count) reaches 0:
- EDMA can reload the channel’s parameters from one of the many Reload sets
- Each Reload set consists of six 32-bit values
- Link Address points to the next Reload setup
- Auto-Init, Reload, and Linking all refer to the same EDMA feature

The reload register sets can also be linked to other reload sets; thus a linked-list can be created.

Creating a “Linked-List” of Transfers

- Offloads CPU ... can reinitialize all six registers of an EDMA channel
- Next transfer specified by Link Address
- Perform simple re-initialization or create linked-list of events
- Useful for ping-pong buffers, data sorting, circular buffers, etc.
6 Steps to Auto-Initialization

Here is a nice 6-step procedure for setting up EDMA Auto-Initialization.

**Reloading an EDMA channel in 6 Steps**

1. Choose LINK option
2. Allocate handle for reload values
3. Allocate a reload set
4. Configure reload set (with same values as original channel)

And the 5th step is ...

**Steps 5 & 6: Set the Link Address fields**

5. `EDMA_link(hMyHandle, hMyReload)`
6. `EDMA_link()` pokes hMyReload address into Link Address field
Here’s a code summary of the six steps required for setting up a channel for linking:

### Reloading an EDMA channel in 6 Steps

1. **Modify your config to enable linking:**
   ```c
   EDMA_OPT_RMK(  
       ...  
       EDMA_OPT_LINK_YES, ... ),
   ```
2. **Create a handle to reference a Reload location:**
   ```c
   EDMA_Handle hMyReload;
   ```
3. **Allocate a Reload location** (reserve a reload set; -1 for “any”)
   ```c
   hMyReload = EDMA_allocTable(-1);
   ```
4. **Configure reload set** (writes config structure to reload set)
   ```c
   EDMA_config(hMyReload, &myConfig);
   ```
5. **Update Link Address fields** (modifies field in chan, not myConfig struct)
   ```c
   EDMA_link(hMyHandle, hMyReload);
   EDMA_link(hMyReload, hMyReload);
   ```
Summary

Here is the complete flow of EDMA interrupts, from EDMA channel to CPU:

### Generate EDMA Interrupt Summary

<table>
<thead>
<tr>
<th>EDMA Channels</th>
<th>EDMA Int Generation</th>
<th>CPU Interrupts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel #</td>
<td>Options</td>
<td>CIPR</td>
</tr>
<tr>
<td>0</td>
<td>TCINT=1 TCC=0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>TCINT=0 TCC=0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>TCINT=1 TCC=1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>TCINT=0 TCC=15</td>
<td>0</td>
</tr>
</tbody>
</table>

- **Set EDMA to generate an interrupt to the CPU:**
  1. (CIPR) Reserve CIPR bit using EDMA_intAlloc()
  2. (Options) TCINT = 1
  3. (CIER) Set CIER bit to match TCC value

- **Set CPU to respond to interrupt from EDMA**
  1. (IER) Enable individual EDMA interrupt
  2. (CSR GIE) Enable interrupts globally

While the flow from EDMA completion to CPU interrupt may be a bit involved, it provides for an extremely flexible, and thus capable, EDMA controller. (In fact, the EDMA is often called a co-processor due to its extreme flexibility.)
Configuring EDMA Interrupts in 6 Easy Steps

In the first step of this procedure we use introduce a new CSL macro: _FMK

**EDMA Interrupts (6 steps)**

1. Modify EDMA Config structure for TCINT & TCC
   - `EDMA_OPT_TCINT_YES`, //set channel to interrupt CPU
   - `EDMA_OPT_TCC_OF(0)`, //set TCC in code

   ```c
   gTcc = EDMA_intAlloc(-1); //reserve TCC (0-15)
   myConfig.opt |= EDMA_FMK(OPT, TCC, gTcc); //set TCC in myConfig
   ```

   **What does the _FMK macro do?**

   _FMK builds a 32-bit mask that can be used to OR a value into a register. In our case, we’re using it to put the CIPR value allocated by EDMA_intAlloc into the TCC field of the Options register. Note, it is important that the previous value for TCC have been set to “0000” when using the OR command shown above. This is why we set TCC = 0 in the global EDMA configuration.

   **CSL’s _FMK macro (field make)**

   ```
   EDMA Options Register
   Peripheral | Register | Field | Value
   ---------------
   TCC
   19 16
   0011
   gTCC = 3
   ```

   ```c
   EDMA_FMK(OPT, TCC, gTCC) = 0x00030000
   ```

   Some additional notes for _FMK:
   - Before you can ‘or’ gTCC into the TCC bit field, it must be shifted left by 16 bits (to make it line up).
   - While is easy to write a right shift by sixteen bits in C, you must know that the TCC field is 4-bits wide from bits 19-16. The _FMK macro already knows this (so we don’t have to look it up.)
   - Worse yet, without _FMK, everyone who maintains this code must also know the bit values for TCC. (Or they’ll have to look it up, too.)
   - _FMK solves this for you. It creates a 32-bit mask value for you. You need only recall the symbol names: Peripheral, Register, Field, and Value.
Here is the complete summary of the 6-step procedure for setting up an EDMA channel to interrupt the CPU.

### EDMA Interrupts (Part 1)

1. **Modify EDMA Config structure for TCINT & TCC**
   - `EDMA_OPT_TCINT_YES, //set channel to interrupt CPU`
   - `EDMA_OPT_TCC_OF(0), //set TCC in code`

   ```c
   gTcc = EDMA_intAlloc(-1); //reserve TCC (0-15)
   myConfig.opt |= EDMA_FMK(OPT,TCC, gTcc); //set TCC in myConfig
   ```

2. **Hook the ISR to the appropriate TCC value**
   - `EDMA_intHook(gTcc, myISR);`

3. **Set the appropriate bit in the CIER register**
   - `EDMA_intEnable(gTcc); // must match chosen TCC value`

### EDMA Interrupts (Part 2)

4. **Include the header file**
   - `#include <csl_irq.h>`

5. **Set the appropriate bit in the IER register**
   - `IRQ_enable(IRQ_EVT_EDMAINT);`

6. **Turn on global interrupts**
   - `IRQ_globalEnable(); // turn on interrupts globally`

---

**When the transfer completes...what happens?**
The EDMA ISR

Here is the summary for how a function is run, which is associated with the completion of an EDMA channel.

The flow described above is specific to the upcoming lab exercise. Though much of it is generic, two of the steps are specific:

- The lab asks you to setup autoinitialization for the channel we’re using. This may, or may not, be what you need in another system.
- The final step triggers the EDMA to run using the EDMA_setChannel() function. Often this is done automatically by interrupt events. In Lab 5, we will use the _setChannel function, but the next lab uses the McBSP to trigger the EDMA to run.
The EDMA's CSL Functions

With so many EDMA control registers, and so many CSL functions, we thought a summary which correlated the functions to the EDMA registers they act upon might be helpful.

### EDMA Functions (Which Registers they Affect)

<table>
<thead>
<tr>
<th>Function</th>
<th>Register</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDMA_setChannel</td>
<td>ESR</td>
<td>(sets ESR bit which sets corresponding ER bit)</td>
</tr>
<tr>
<td>EDMA_clearChannel</td>
<td>ECR</td>
<td>(sets ECR bit which clears corresponding ER bit)</td>
</tr>
<tr>
<td>EDMA_getChannel</td>
<td>ER</td>
<td>(Event Register)</td>
</tr>
<tr>
<td>EDMA_enableChannel</td>
<td>EER</td>
<td>(Event Enable Register)</td>
</tr>
<tr>
<td>EDMA_disableChannel</td>
<td>ER</td>
<td>(Event Disable Channel)</td>
</tr>
<tr>
<td>EDMA_intAlloc</td>
<td>CCER</td>
<td>(Chan Chaining Enable Reg)</td>
</tr>
<tr>
<td>EDMA_intFree</td>
<td>CCPR</td>
<td>(Chan Chaining Enable Reg)</td>
</tr>
<tr>
<td>EDMA_intTest</td>
<td>CIPR</td>
<td>(Chan Interrupt Pending Reg)</td>
</tr>
<tr>
<td>EDMA_intClear</td>
<td>CIER</td>
<td>(Chan Interrupt Enable Reg)</td>
</tr>
<tr>
<td>EDMA_enableChaining</td>
<td>CCER</td>
<td>(Chan Chaining Enable Reg)</td>
</tr>
<tr>
<td>EDMA_disableChaining</td>
<td>CCPR</td>
<td>(Chan Chaining Enable Reg)</td>
</tr>
</tbody>
</table>

Here’s the same summary, but we’ve added the function’s arguments and return values.
Exercise

Exercise 1 (Review)

- Complete the following Interrupt Service Routine.
  Here’s a few hints:
  - Follow the code outlined on the “EDMA ISR” slide.
  - Don’t forget, though, that our exercise (and the upcoming lab) uses different variable names than those used in the slide’s example code.
  - To “fill the buffer”, what function did we use in Labs 2 and 4 to create a buffer of sine wave data?

```c
void edmaHi(void)
{
    SINE_blockFill(gBuf0, BUFFSIZE); // Fill buffer with sine data
    EDMA_setChannel(hEdma); // start EDMA running
};
```

Exercise 2: Step 1

1. Change gEdmaConfig so that it will: (Just cross-out the old and jot in the new value)
   - Interrupt the CPU when transfer count reaches 0
   - Auto-initialize and keep running

```c
EDMA_Config gEdmaConfig = {
    EDMA_OPT_PRI_LOW,     // Priority?
    EDMA_OPT_ESIZE_16BIT, // Element size?
    EDMA_OPT_2DS_NO,      // 2 dimensional source?
    EDMA_OPT_SUM_INC,     // Src update mode?
    EDMA_OPT_2DD_NO,      // 2 dimensional dest?
    EDMA_OPT_DUM_INC,     // Dest update mode?
    EDMA_OPT_TCINT_NO,    // Cause EDMA interrupt?
    EDMA_OPT_TCC_OF(0),   // Transfer complete code?
    EDMA_OPT_LINK_NO,     // Enable link parameters?
    EDMA_OPT_FS_YES       // Use frame sync?
};
```
Exercise 2: Steps 2-4

2. Reserve “any” CIPR bit (save it to gXmtTCC). Then set this value in the gEdmaConfig structure.

3. Allow the EDMA’s interrupt to pass through to the CPU. That is, set the appropriate CIER bit. (Hint: the TCC value indicates which bit in CIPR and CIER are used)

4. Hook the ISR function so it is called whenever the appropriate CIPR bit is set and the CPU is interrupted.

Exercise 2: Steps 5

5. Enable the CPU to accept the EDMA interrupt. (Hint: Add 3 lines of code.)

```c
void initHwi(void)
{

}
```

Please continue on to the next page.
6. Declare a handle for an EDMA reload location and name it `hEdmaReload`:

7. Allocate one of the Reload sets: (Hint: `hEdmaReload` gets this value)

8. Configure the EDMA reload set:

9. Modify both the EDMA channel and the reload set to link to the reload set of parameters:
Lab 5

Overview

In lab 5, you'll have an opportunity to test everything that you have learned about interrupts and auto-initialization.

Lab 5 – Programming the EDMA

Pseudo Code
1. CPU generates 32 sine values into buf0
2. EDMA transfers 32 elements from buf0 to buf1
3. EDMA sends “transfer complete” interrupt to CPU
4. Go to step 1

Goals of the lab:

- To use CSL to configure the EDMA interrupt to the CPU in order to generate another buffer full of sine wave values.
- To change the configuration of the EDMA so that it uses auto-initialization to setup the next transfer.
Lab Overview

This lab will follow the basic outline of the discussion material. Here's how we are going to go about this:

- First, we're going to configure the CPU to respond to interrupts and set up the interrupt vector using the .cdb file. We're going to configure the CPU to call the EDMA dispatcher that will call our function to process the EDMA interrupt.
- Next, we'll write the function that we want the EDMA dispatcher to call.
- Then, we'll change some setting in the EDMA configuration and the initEdma( ) code. One thing that we'll definitely need to do is to tell the EDMA dispatcher to call the function that we wrote in the previous step.
- Finally, we'll configure the EDMA channel to use auto-initialization.

Configure the CPU to Respond to Interrupts

How does the CPU know what to do when the interrupt occurs? Where does code execution go? We need to tell the CPU that when the EDMA interrupt occurs, we want it to call the EDMA interrupt dispatcher. The EDMA dispatcher will then see what interrupts have occurred and call the configured functions.

During this part of the lab, we will be somewhat following the "6-step procedure to program the EDMA to interrupt the CPU" outlined on pages 5-21 to 5-23. Feel free to flip back and review that material before trying to write the code.
1. Reset the DSK, start CCS and open audioapp.pjt.

2. Open the CDB file and click the + sign next to Scheduling

3. Click the + sign next to HWI – Hardware Interrupt Manager
   A list of hardware interrupts will appear. Hardware interrupt #8 (HWI_INT8) is the EDMA interrupt to the CPU (by default).

4. Right-click HWI_INT8 and select Properties

5. Change the function name to _EDMA_intDispatcher
   The hardware interrupt vector table is written in assembly, so the underscore is required to access the C function, EDMA_intDispatcher( ), which is provided by CSL.

6. Use the HWI Dispatcher
   Click on the Dispatcher tab and check the Use Dispatcher checkbox. Click OK. Close and Save.

   We are actually using two dispatchers here as we discussed in the material. The HWI dispatcher that we configured with the check box takes care of context save/restores for the ISR routine. The EDMA dispatcher figures out which EDMA interrupts to the CPU need to run and calls the functions to handle them.

**Initializing Interrupts**

We need to set up two things: (1) enable the CPU to respond to the EDMA interrupt (IER) and (2) turn on global interrupts (GIE). Refer to the discussion material which outlines the 5-step CSL procedure for initializing an HWI.

7. Add a new function called initHwi( ) at the end of your code in main.c
   We will use this function to initialize hardware interrupts. We will add a call to it in main( ) in few steps.

8. Add a call to IRQ_enable( ) in initHwi( ) to enable the EDMA interrupt to the CPU
   This connects the EDMA interrupt to the CPU via the IER register.

9. Enable CPU interrupts globally and terminate the initHwi() function
   Add the CSL function call that enables global interrupts (GIE). Add a closing brace to the function to finish it off.

10. Add the proper include file for interrupts to the top of main.c in the "include" area

11. Add a call to initHwi( ) in main( ) after the call to initEdma( )
**Writing the ISR Function**

We need to set up the EDMA to cause a CPU interrupt when it finishes transferring a buffer (i.e., when the transfer count reaches zero – this is the transfer complete interrupt). We then will set up a CPU Interrupt Service Routine (ISR for short) to fill the source buffer with new sine values and kick-off the EDMA to transfer them to the other buffer.

12. **Review the Pseudo Code for Our System**

Here is a summary of the code that you will need to write in the ISR function. The steps to write this code will follow.

So, the new code will look something like this:

- Init the EDMA to fire an interrupt when it completes
- Init the CPU to recognize the EDMA’s interrupt
- Enter the infinite while loop
- While running in the infinite while loop
  - When our EDMA interrupt (HWI) occurs, code execution goes to the ISR
  - In the ISR, the buffer is filled with new sine values and the EDMA copy is triggered
  - We re-enter the while loop. When the copy is done, the EDMA causes another CPU interrupt ... and so on ...

**Hint:** Whenever the instructions ask you to “add a new function”... don’t forget to prototype it! We’ve already added it to the header file for you for inclusion in other files.

13. **Add a new function called edmaHwi() at the end of your edma.c code**

This function will serve as our Interrupt Service Routine (ISR) that will get called by the EDMA interrupt dispatcher. The EDMA interrupt dispatcher passes the CIPR bit of the EDMA channel that caused the interrupt to the edmaHwi() routine. We will not be using this argument for now, but we will need it later. So, go ahead and write the function with the argument in the definition like this:

```c
void edmaHwi(int tcc)
```
14. Copy SINE_blockFill( ) and EDMA_setChannel( )

Every time the ISR occurs, we want to fill a buffer and trigger the EDMA to copy the buffer. So, copy the code that calls SINE_blockFill( ) and EDMA_setChannel( ) routines from main() to the ISR function edmaHwi() you just created.

Make sure that you copy these function calls. Do not delete them from main(). The calls are needed in main() to “prime the pump” (i.e. get the whole process started). If we don't do this, the ISR will never run because the first buffer never gets transferred. So, leave the calls in main().

Use a closing brace to complete the edmaHwi() ISR.

15. Create an external reference to SINE_Obj

The SINE_blockFill( ) function refers to the SINE_Obj created in main.c. So, we need to create an external reference to it much like we did to the buffers in the previous lab.

16. Add sine.h to edma.c

Add a #include statement for sine.h to edma.c to take care of the prototype for SINE_blockFill() and the SINE_Obj data type.

**Configuring the EDMA to Send Interrupts**

While you have just setup the CPU to respond to interrupts properly … currently, the EDMA is not setup to send interrupts. We need to modify the EDMA config structure to tell the EDMA to send an interrupt when it completes a transfer. We also need to modify the initEDMA( ) code to make some other changes in order to initialize interrupts properly.

17. Turn on the EDMA interrupt in the EDMA config structure

Change TCINT field to YES. This will cause the EDMA to trigger an interrupt to the CPU.

18. Create a Global Variable to store to TCC Value

We don’t really care which TCC value gets used – it’s arbitrary.

Create a global variable (of type short) named gXmtTCC.

**Modify initEdma( )**

19. Configure the EDMA Channel to use a TCC Value

Configure the channel using your new variable. (It’s a two step process.)

- Inside the initEdma function (after the _open) set gXmtTCC equal to “any” TCC value as shown in the discussion material.
- Then set the actual TCC field (in the configuration) to this value.

This reserves a specific TCC value so that no other channel can use it.

After referring to the material, you hopefully came up with these two steps to be added to initEdma( ):

```c
  gXmtTCC = EDMA_intAlloc(-1);
  gEdmaConfig.opt |= EDMA_FMK(OPT, TCC, gXmtTCC);
```
20. **Hook the edmaHwi( ) function into the EDMA Interrupt Dispatcher**

The EDMA Interrupt Dispatcher automatically calls a function for each of the CIPR bits that get set by an EDMA interrupt and that are enabled.

We need to tell it what function to call when the transmit interrupt fires. The transmit interrupt is going to assert a given CIPR bit when it occurs. So, we need to tell the EDMA Interrupt Dispatcher which function is tied to that CIPR bit. Refer back to the lecture material if you can't figure out which API call to use here, or how to use it. Don't forget about online help inside CCS as well. Add this code anywhere in the initEdma( ) function that makes sense to you.

21. **Clear any spurious interrupts and enable the EDMA interrupt**

At the end of the `initEdma()` function in `edma.c`, add the following calls to clear the EDMA’s channel interrupt pending bit associated with the channel we’re using (i.e. clear the appropriate CIPR bit). Also, enable the EDMA interrupt (i.e. set the required CIER bit). Note, the same TCC value used earlier is required for both these operations.

```c
EDMA_intClear(gXmtTCC);
EDMA_intEnable(gXmtTCC);
```

**Initialize the Channel’s Link Address**

Now that we've got interrupts all set up, let's configure the channel to auto-initialize each time it completes. In addition to interrupting the CPU, this will be done each time the EDMA channel completes a transfer.

We will be following the "6 Steps to Auto-Initialization" procedure outlined earlier. Please feel free to refer back to this material to help you understand this part of the lab.

22. **Enable the link parameters**

Change the LINK field to YES in the EDMA Configuration Structure. This will cause the channel to link to a reload entry and refresh the channel with its original contents – this is called autoinitialization. The next few steps will set up the channel’s link address to the reload entry.

23. **Add another global EDMA handle named hEdmaReload to edma.c**

24. **Initialize the new reload entry handle**

In `initEdma()`, add the following API call to initialize the reload handle (hEdmaReload) to ANY reload entry location:

```c
hEdmaReload = EDMA_allocTable(-1);
```

You can see an example of this in the discussion material. This handle points to the reload entry that we will initialize with the original channel's EDMA config structure.
25. **Configure the Reload Entry**

We have already configured the channel registers using `EDMA_config`. You now need to configure the reload entry using the same configuration and API (different handle):

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

26. **Link the channel and reload entry to the reload handle**

After the channel finishes the first transfer, we need to tell it where to link to for the next transfer. We need to link the channel to the new reload entry handle (acquired in the previous step) AND we need to link the reload entry to itself for all succeeding transfers. This is the basis of autoinitialization. Use the proper API to link the channel to the reload entry and use that same API to link the reload entry to itself. Go ahead and add this code to `initEdma()`.

**Build and Run**

27. **Build/load the project and fix any errors**

28. **Run the code, then halt and verify that both buffers contain sine values.**

Graph `gBuf0` and `gBuf1` – do they look like sine waves? They might look a bit funny based on when you hit “Halt”. At this point, we have verified that the buffers are being written to at least once. However, we have not verified that they are being written repeatedly. So, let’s try a CCS technique to verify this. Unfortunately, this will have an affect on real-time operation…but we’ll discover a workaround for this later in the BIOS discussion.

29. **Set a breakpoint in the `edmaHwi()` function.**

Open `edma.c` and look in the `edmaHwi()` function. Set a breakpoint anywhere inside the `edmaHwi()` function. Make sure you can see a graph of `gBuf0` or `gBuf1`.

30. **Animate your code**

   Click the *Animate* button:

   on the vertical tool bar. You should see your buffers and your graph update continuously. If so, halt your code.

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

   Using Windows Explorer, copy the contents of:

   c:\iw6000\labs\audioapp\*. * TO c:\iw6000\labs\lab5

   ![STOP]

   You're Done
Optional Topics

Saving Context in HWIs

Interrupt Keyword

```c
main()
...

interrupt occurs

next instruction
...
```

```c
interrupt myISR(void);
context save ...
context restore ...
B IRP;
```

Interrupt Keyword
- When using the `interrupt` keyword:
  - Compiler handles register preservation
  - Returns to original location
- No arguments (void)
- No return values (void data type)

The HWI dispatcher...

HWI Dispatcher

```c
main()
...

interrupt occurs

next instruction
...
```

```c
HWI Dispatcher:
context save
context restore
```

```c
void myISR(arg1);
----
----
----
return;
```

Dispatcher
- Uses standard (unmodified) C function, which allows the use of algorithms from an object file (library)
- Required when interrupt uses DSP/BIOS scheduler functions
- Easy to use -- simple checkbox
- Simple way to nest interrupts
- Saves code space -- since all share one context save/restore routine

Comparing the two...
Optional Topics

HWI Dispatcher vs. Interrupt Keyword

1. HWI Dispatcher
   - Allows nesting of interrupts
   - Saves code space
   - Required when ISR uses BIOS scheduler functions
   - Allows an argument passed to ISR

2. Interrupt Keyword
   - Provides highest code optimization (by a little bit)

Notes:
- Choose HWI dispatcher and Interrupt keyword on an interrupt-by-interrupt basis

Caution:
For each interrupt, use only one of these two interrupt context methods

3. Write ISR’s using Assembly Code

   .include “hwi.s62"

   myASM_ISR:
   
   HWI_enter C62_ABTEMPS, 0, 0xffff, 0
   
   Your ISR code …
   
   HWI_exit C62_ABTEMPS, 0, 0xffff, 0

   - If using Assembly, you can either handle interrupt context/restore & return with the HWI dispatcher, or in your own code
   - If you don’t use the HWI Dispatcher, the HWI _enter/_exit macros can handle:
     - Context save (save/restore registers)
     - Return from interrupt
     - Re-enable interrupts (to allow nesting interrupts)

   HWI enter: Modify IER and re-enable GIE
   HWI exit: Disable GIE then restore IER
Interrupts and the DMA

DMA Interrupt Generation

**DMA: Interrupt Generation**

Generate Interrupt to CPU When:
- Split XMT Overrun (SX)
- Frame complete (FRAME)
- Start xfr last frame (LAST)
- Block xfr completes (BLOCK)
- WSYNC drop (WDROP)
- RSYNC drop (RDROP)

CND = true (1)
IE = int enable

**DMA: Interrupt Generation**

- **DMA_INT** signal generates CPU interrupt if enabled in IER
- During ISR, CPU may need to check DMA’s Secondary Control register to determine cause of DMA interrupt
- CPU must clear CND bit in Secondary Control register

CND = true (1)
IE = int enable
DMA Reload Process

**DMA: Auto Reload**

- Use auto-reload to automatically reload the DMA channel for next block of transfers
- Unlike EDMA only 3 registers can be reloaded:
  - Source Address
  - Destination Address
  - Transfer Counter

Three steps are required to use this feature:

1. **START:** 11b (start w/auto-init enabled)

**DMA Start bits:**

- 00: Stop
- 01: Start w/o auto-init
- 10: Pause
- 11: Start w/auto-init

**DMA Global Registers**

- Count Reload A
- Count Reload B
- Index A
- Index B
- Address A
- Address B
- Address C
- Address D

**DMA: Auto Reload**

![Diagram of DMA Reload Process]

- Use auto-reload to automatically reload the DMA channel for next block of transfers
- Unlike EDMA only 3 registers can be reloaded:
  - Source Address
  - Destination Address
  - Transfer Counter

Three steps are required to use this feature:

1. **START:** 11b (start w/auto-init enabled)
2. **SRC/DST RELOAD:** specifies which global address register (B, C, D or none)
3. **CNT RELOAD:** specifies which global count reload register (A, B)

**DMA Start bits:**

- 00: Stop
- 01: Start w/o auto-init
- 10: Pause
- 11: Start w/auto-init

**DMA Global Registers**

- Count Reload A
- Count Reload B
- Index A
- Index B
- Address A
- Address B
- Address C
- Address D
### DMA/EDMA Comparison

#### 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 + 1 for HPI</td>
<td>16 channels + 1 for HPI</td>
<td>64 channels + 1 for HPI</td>
</tr>
<tr>
<td>Sync</td>
<td>element + frame</td>
<td>element + frame + 2D (block)</td>
<td></td>
</tr>
<tr>
<td>CPU Interrupts</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt Conditions</td>
<td>six: 3 for Count, 3 for errors</td>
<td>Count = 0</td>
<td></td>
</tr>
<tr>
<td>Reload (Auto-Init)</td>
<td>~2</td>
<td>69</td>
<td>21</td>
</tr>
<tr>
<td>Chain Channels</td>
<td>None</td>
<td>4 channels (8-11)</td>
<td>64 channels</td>
</tr>
<tr>
<td>Priority</td>
<td>4 fixed levels</td>
<td>2 prog levels</td>
<td>4 prog levels</td>
</tr>
</tbody>
</table>

---

Optional Topics

**Technical Training Organization (TT O)**

5 - 40 C6000 Integration Workshop - Hardware Interrupts (HWI)
EDMA Channel Chaining

When one channel completes, it can trigger another to run.

C67x: only channels 8-11 can be used for chaining.

C64x: all channels can be chained.

To chain channels:
1. \text{CIPR} # must match Channel #
2. \text{CIER} can be 0 or 1
3. \text{CCER} # must be 1
4. \text{EER} # must be 1

What's the difference between EDMA Auto-Initialization and EDMA Channel Chaining?

Alternate Transfer Chaining (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 (64x only)
        EDMA_OPT_ATCC_DEFAULT,   // Alternate Transfer Complete Code (64x only)
        ...
    )
};
```

- Similar to EDMA channel chaining, but an event/interrupt is generated after each intermediate transfer (i.e. each request sent to the Transfer Controller).

- This allows you to send an event sync signal to a chained EDMA channel (or CPU interrupt) at the end of each transfer.

- By having both ATCC and TCC, it allows two different sync signals to be generated. One at the end of each transfer, another at the end of all transfers.

- Useful for very large transfers. Rather than triggering a big transfer request that would tie up a bus resource for too long, a transfer can be broken into many, smaller intermediate requests. (See the EDMA documentation for examples of this.)
Additional HWI Topics

NMIE

Enabling Interrupts

What events/conditions are required to recognize an interrupt?

![Diagram showing IER and CSR_GIE connections]

Interrupt Enable Register (IER)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Reserved</td>
</tr>
<tr>
<td>15-14</td>
<td>Reserved</td>
</tr>
<tr>
<td>12-10</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>Reserved</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
</tbody>
</table>

- To enable each int, write “1” to IE bit
- IER bits are NOT affected by the value in global interrupt enable (GIE)

```c
// To enable, then disable the timer0 int
IRQ_enable(IRQ_EVT_TINT0);
IRQ_disable(IRQ_EVT_TINT0);
```
NMIE - NMI Enable?

- NMIE enables the non-maskable interrupt (NMI)
- Exists to avoid unwanted NMI's occurring after RESET and before system is initialized
- NMIE must be enabled for any interrupts to occur
- Once enabled, NMI is non-maskable
- Enable NMIE just before exiting your boot routine
- NMIE is automatically set before main() when CDB file is included in the project

External Interrupt Pins

- To generate a valid interrupt signal, hold INTx low for 2+ cycles, then high for 2+ cycles
- Interrupt is latched on rising edge of CLKOUT1 following a rising edge of INTx (if above timing is met)
- Interrupt is recognized by the CPU one cycle later
Optional Topics

# External Interrupt Polarity

Allows change of polarity for the external interrupts EXT_INT4-7

<table>
<thead>
<tr>
<th>XIP7</th>
<th>XIP6</th>
<th>XIP5</th>
<th>XIP4</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

0 = low to high (default)
1 = high to low (inverted)

## Mapped Interrupt Registers Address (hex)

- Interrupt Multiplexor (High) 019C_0000
- Interrupt Multiplexor (Low) 019C_0004
- External Interrupt Polarity 019C_0008
Interrupt Vectors

Interrupt Vectors

RESET:

```
mvkl _c_int00,b0
mvkh _c_int00,b0
bb 0
nop
nop
nop
nop
...
```

```
20h
80h
A0h
0h
200h
```

```
_c_int00
boot.c
```

HWI_RESET Properties

- Vector table can be relocated ...

Vector Table Pointer (ISTP)

- ISTP is located in CPU
- ISTB field points to vector table
- Allows you to locate Interrupt Vector Table on any 1K boundary
- Configure with in CDB file or use IRQ_setVecs()

Locates vector table on any 1K boundary

```
ISTB field
```

Use CDB file to set ISTP ...
Let CDB setup ISTP

What does the Vector Table look like?

New Interrupt Vector Table

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>ISFP Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>0x000</td>
</tr>
<tr>
<td>NMI</td>
<td>ISTB + 0x020</td>
</tr>
<tr>
<td>INT4</td>
<td>ISTB + 0x080</td>
</tr>
<tr>
<td>INT5</td>
<td>ISTB + 0x0A0</td>
</tr>
<tr>
<td>INT6</td>
<td>ISTB + 0x0C0</td>
</tr>
<tr>
<td>INT7</td>
<td>ISTB + 0x0E0</td>
</tr>
<tr>
<td>INT8</td>
<td>ISTB + 0x100</td>
</tr>
<tr>
<td>INT9</td>
<td>ISTB + 0x120</td>
</tr>
<tr>
<td>INT10</td>
<td>ISTB + 0x140</td>
</tr>
<tr>
<td>INT11</td>
<td>ISTB + 0x160</td>
</tr>
<tr>
<td>INT12</td>
<td>ISTB + 0x180</td>
</tr>
<tr>
<td>INT13</td>
<td>ISTB + 0x1A0</td>
</tr>
<tr>
<td>INT14</td>
<td>ISTB + 0x1C0</td>
</tr>
<tr>
<td>INT15</td>
<td>ISTB + 0x1E0</td>
</tr>
</tbody>
</table>
**HWI Interrupt Selector**

There are 12 configurable interrupts

Most C6000 devices have more than 12 interrupt sources

The interrupt selector allows you to map any interrupt source to any HWI object

Side benefit is that you can change the hardware interrupt priority
Optional Topics

### Interrupt Selection

<table>
<thead>
<tr>
<th>Sel #</th>
<th>C6701 Sources</th>
<th>Interrupt Multiplexer High (INT10 - INT15)</th>
<th>Interrupt Multiplexer Low (INT4 - INT9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b</td>
<td>(HPI) DSPINT TINT0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001b</td>
<td>TINT1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010b</td>
<td>SD_INT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011b</td>
<td>EXT_INT4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100b</td>
<td>EXT_INT5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101b</td>
<td>EXT_INT6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110b</td>
<td>EXT_INT7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111b</td>
<td>DMA_INT0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000b</td>
<td>DMA_INT1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001b</td>
<td>DMA_INT2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010b</td>
<td>DMA_INT3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011b</td>
<td>XINT0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100b</td>
<td>RINT0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101b</td>
<td>XINT1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110b</td>
<td>RINT1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Interrupt Selector registers are memory-mapped
- Configured by HWI objects in Config Tool
- Or, set dynamically using IRQ_map()

---

### Interrupt Selection

<table>
<thead>
<tr>
<th>Sel #</th>
<th>C6701 Sources</th>
<th>Interrupt Multiplexer High (INT10 - INT15)</th>
<th>Interrupt Multiplexer Low (INT4 - INT9)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000b</td>
<td>(HPI) DSPINT TINT0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001b</td>
<td>TINT1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010b</td>
<td>SD_INT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0011b</td>
<td>EXT_INT4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0100b</td>
<td>EXT_INT5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0101b</td>
<td>EXT_INT6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0110b</td>
<td>EXT_INT7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111b</td>
<td>DMA_INT0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000b</td>
<td>DMA_INT1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001b</td>
<td>DMA_INT2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010b</td>
<td>DMA_INT3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011b</td>
<td>XINT0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100b</td>
<td>RINT0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101b</td>
<td>XINT1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110b</td>
<td>RINT1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Interrupt Selector registers are memory-mapped
- Configured by HWI objects in Config Tool
- Or, set dynamically using IRQ_map()
**CPU Interrupt Registers**

**Return Pointers (IRP/NRP)**

- When interrupt serviced, address of next execute packet placed in IRP or NRP register
- At the end of interrupt service routine, branch to IRP/NRP:

```plaintext
31 0

IRP (interrupt)  B.S2 IRP ;return, PGIE→GIE
R,W,+x
NOP 5

31 0

NRP (NMI)  B.S2 NRP ;return, NMIE = 1
R,W,+x
NOP 5
```

- **IRQ_set** (sets ISR bit which sets corresponding IFR bit)
- **IRQ_clear** (sets ICR bit which clears corresponding IFR bit)
- **IRQ_map**
- **IRQ_config**
- **IRQ_test**
- **IRQ_enable**
- **IRQ_disable**
- **IRQ_restore**

- **ISR** (Interrupt Set Register)
- **ICR** (Interrupt Clear Register)
- **IFR** (Interrupt Flag Register)
- **IER** (Interrupt Enable Register)
- **IRP** (Interrupt Return Pointer)
- **IRP** (Non-maskable Int. Return Ptr.)
- **ISTP** (Interrupt Service Table Ptr.)

- **IRQ_setVecs** or Use Config Tool
Solutions to Paper Exercises

Exercise 1

Enable CPU Interrupts

Exercise 1: Fill in the lines of code required to enable the EDMAINT hardware interrupt:

```c
void initHWI(void)
{
    IRQ_enable(IRQ_EVT_EDMAINT);
    IRQ_globalEnable();
}
```
Exercise 2

Exercise 2: Step 1

1. Change gEdmaConfig so that it will: (Just cross-out the old and jot in the new value)
   - Interrupt the CPU when transfer count reaches 0
   - Auto-initialize and keep running

   ```
   EDMA_Config gEdmaConfig = {
       EDMA_OPT_RMK(  
           EDMA_OPT_PRI_LOW,     // Priority?
           EDMA_OPT_ESIZE_16BIT, // Element size?
           EDMA_OPT_2DS_NO,      // 2 dimensional source?
           EDMA_OPT_SUM_INC,     // Src update mode?
           EDMA_OPT_2DD_NO,      // 2 dimensional dest?
           EDMA_OPT_DUM_INC,     // Dest update mode?
           EDMA_OPT_TCINT_OPT,   // Cause EDMA interrupt?
           EDMA_OPT_TCC_OF(0),   // Transfer complete code?
           EDMA_OPT_LINK_NO,     // Enable link parameters?
           EDMA_OPT_FS_YES       // Use frame sync?
       ),
       ... };  
   ```

Exercise 2: Steps 2-4

2. Reserve “any” CIPR bit (save it to gXmtTCC). Then set this value in the gEdmaConfig structure.

   ```
   gXmtTCC = EDMA_intAlloc(-1);
   gEdmaConfig.opt |= EDMA_FMK (OPT, TCC, gXmtTCC);
   ```

3. Allow the EDMA’s interrupt to pass through to the CPU.
   That is, set the appropriate CIER bit.
   (Hint: the TCC value indicates which bit in CIPR and CIER are used)

   ```
   EDMA_intEnable(gXmtTCC);
   ```

4. Hook the ISR function so it is called whenever the appropriate CIPR bit is set and the CPU is interrupted.

   ```
   EDMA_intHook(gXmtTCC, edmaHWI);
   ```
Exercise 2: Steps 5

5. Enable the CPU to accept the EDMA interrupt. (Hint: Add 3 lines of code.)

```c
#include <csl_irq.h>

void initHwi(void)
{
    IRQ_enable(IRQ_EVT_EDMAINT);
    IRQ_globalEnable(void);
}
```

Exercise 2: Steps 6-9 (EDMA Reload)

6. Declare a handle for an EDMA reload location and name it `hEdmaReload`:

   ```c
   EDMA_Handle hEdmaReload;
   ```

7. Allocate one of the Reload sets: (Hint: `hEdmaReload` gets this value)

   ```c
   hEdmaReload = EDMA_allocTable(-1);
   ```

8. Configure the EDMA reload set:

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

9. Modify both the EDMA channel and the reload set to link to the reload set of parameters:

   ```c
   EDMA_link(hEdma, hEdmaReload);
   EDMA_link(hEdmaReload, hEdmaReload);
   ```