Implementing a Double Buffered System

Introduction

In this module, we will discuss some different ways to handle system timing issues. We will define some terms that can be used to describe a system and its timing. We will also discuss a couple of different ways to solve timing issues. We'll take a brief look at optimization to see how it helps solve timing problems. We'll also learn the benefits of a double-buffered system and how to modify your current single buffered system into a double-buffered system.

Learning Objectives

The main purpose of this module is to help you implement a double-buffered system on a C6000 DSP.
Chapter Topics

Implementing a Double Buffered System.......................................................................................... 8-1

Chapter Topics.................................................................................................................................. 8-2

What is Real Time? ................................................................................................................................. 8-3
  Definition......................................................................................................................................... 8-3
  Single Buffer System Timing ........................................................................................................6 8-4
  Double Buffer System Timing.................................................................................................... 8-7

Implementing a Double Buffer System............................................................................................ 8-8
  How do you implement a double buffer system? ....................................................................... 8-8

Lab 8 ................................................................................................................................................... 8-9
  Part A – Double Buffering......................................................................................................... 8-11
What is Real Time?

DSPs are often used in "real-time" systems. "Real-time" systems need to calculate a correct answer in a given amount of time. So, how much time is "real-time"? The answer to this question is often very system dependent. But, there are some general concepts that we can explore that will apply to all "real-time" systems.

Definition

Here is a good general definition of "real-time". Again, the true definition can change from system to system. It basically boils down to "when do you get the data?" and "when do you need to be finished with it?''.

What is Real Time?

Most DSP algorithms benefit from "block processing" where you process multiple samples at once. Some algorithms, FFT for example, require blocks for processing. When processing samples, the CPU has to do a context save/context restore for each sample. When you buffer up samples, the context switch time is dramatically reduced. Also, most algorithms can be optimized to process blocks over samples by using techniques like loop-unrolling and packed data processing (or single instruction, multiple data). We don't discuss these topics much in this class, but the TMS320C6000 Optimization Workshop goes into great detail on these subjects.
**Single Buffer System Timing**

Let's take a closer look at the effect that data buffering has on the timing of a system.

![Lab 7 – Single Buffer System Timing Diagram](image)

- Block must be processed before In-16 arrives
- Processing time increases 16x due to buffer size
- Time constraint is the same: one sample period
- Computationally efficient, but increased latency

Why did we have to decrease our buffer size to get lab 7 to work?

The main point to notice here is that we have the same amount of time \( t_s \) to process a buffer that we had to process a single sample in the previous slide. Does it take longer to process a buffer than it does a single sample?
**A Broken System**

Since one sample period is all the system has to process the buffer, if the buffer size is too large, it may take too much time. This causes the system to break because it will start dropping samples and using buffers that may be discontinuous.

**Why Did the Last Lab Break?** ($t_p > t_s$)

- Processing 32 samples takes longer than processing 16
- The time to process the samples hasn't changed ($t_s$)
- There are 3 solutions to this problem
  1. Decrease buffer size (we did this at the end of lab 7)
  2. Decrease processing time ($t_p$) with optimization
  3. Increase the amount of available time for processing

Let’s see what Solution 2 (optimization) can do for us...

If the system is broken, there are two different ways to fix it:
- Decrease the amount of time needed to process a buffer (the first two solutions above)
- Increase the amount of time that the system has to process a buffer (double-buffering)
The Optimization Solution

One way to fix a system that is missing "real-time" is to decrease the amount of time needed to process a buffer. One way to do this is to optimize the code that does the processing. So, how large an effect can optimization have?

<table>
<thead>
<tr>
<th></th>
<th>No Opt</th>
<th>Opt (-gp -o3)</th>
<th>Fast RTS &amp; Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>C6713</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(225 MHz)</td>
<td>2400 cycles</td>
<td>1024 cycles</td>
<td>Not Needed</td>
</tr>
<tr>
<td></td>
<td>10.7 µs</td>
<td>4.5 µs</td>
<td></td>
</tr>
<tr>
<td><strong>C6416</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(1 GHz)</td>
<td>9600 cycles</td>
<td>8000 cycles</td>
<td>3600 cycles</td>
</tr>
<tr>
<td></td>
<td>9.6 µs</td>
<td>8 µs</td>
<td>3.6 µs</td>
</tr>
</tbody>
</table>

- Without Optimization, we don’t have enough time to process 32 samples (double the processing time)
- The C6713 is much more efficient because it is floating-point
- The Fast RTS (run-time support) library is an optimized floating point library for C64x and C62x

* Approximate numbers obtained with CCS profiler.
The time to copy the data is NOT included.

Note: The C64x is slower? Why? These benchmarks are for the sine wave generator that we have been using in the labs. Is this algorithm a fixed- or floating-point algorithm? It is a floating-point algorithm. The C64x is a fixed-point processor, while the C67x is a floating-point processor. The C64x has to call floating-point library routines that emulate floating-point on a fixed-point device. These routines are not available to the C Compiler for optimization. This reduces its efficiency dramatically.

It is easy to see how big an effect optimization has on system timing. The optimization used here is very basic, and there are other steps that could have been taken to further optimize these routines. Even with basic optimization, the performance of these routines can be dramatically improved.

The Fast RTS Library for the C62x and C64x processors contain optimized floating-point routines that can help these processors deal with floating-point much more efficiently. These libraries can be downloaded from our web site, www.dspvillage.com.
Double Buffer System Timing

Another solution to system timing issues is to increase the amount of time that the algorithm has to process a buffer. This can be done by using two buffers instead of one. This is called a double-buffered system of a ping-pong buffer system.

3. Double Buffer System Timing

- Time constraint is now buffer length \( t_B \) (NOT sample period \( t_S \))
- Processing is the same as the single buffer system
- Latency is increased, but it is deterministic
- Simultaneous receive, process, transmit
- Also called Ping/Pong buffering

How do we implement a double buffered system?

Notice that the time allowed to process a buffer is no longer sampling period \( t_S \). It is now the sampling period times the length of the buffer \( t_B \). This extra time can be used to reduce the amount of optimization that needs to be done, increase the buffer size for more efficiency, or simply allow for changes later on.

Are there any consequences of double-buffering that should be considered? Sure, it takes more memory and it adds more latency. So, this is something else to add to the engineering balance sheet.

The concept of double-buffering can also be extended to included more than two buffers. This is very common in different kinds of systems where there is a lot of data and latency is not a big issue (i.e. video).
Implementing a Double Buffer System

Let's make double-buffering easy to implement by breaking it down step by step.

How do you implement a double buffer system?

**Implementing a Double Buffer System (1)**

- Add a second buffer to receive and transmit:

![Buffers diagram]

- In the HWI, add a variable to check status of ping/pong:

```java
if (pingpong == 0) {
    copy RcvPing to XmtPing
    pingpong = 1;
} else {
    copy RcvPong to XmtPong
    pingpong = 0;
}
```

**Implementing a Double Buffer System (2)**

- For the EDMA, we need to create two reload entries (ping and pong) for both receive and transmit (receive only shown below):

![EDMA configuration diagram]

- Psuedo Code
  - Allocate reload entries for Ping and Pong
  - Src = DRR (McBSP0)
  - EDMA_config (…)
  - Link: channel → Pong, Pong → Ping, Ping → Pong
Lab 8

Let's go off and apply all of the new knowledge that we have learned. In Lab 8, we'll take the single-buffered system that we've had and make it double-buffered.

**Open Audioapp.pjt**

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

**Add Load to the Single Buffer System**

As we discussed earlier, there is a limited amount of time to process the input buffer. What we want to do is add a load of NOPs inside the HWI that we can use to determine how much delay the application can handle before breaking. We can use a function named `load(loadValue)` to add the simulated load. This function is contained in a file named `load_6416.asm` or `load_6713.asm`.

2. **Add load_6416.asm or load_6713.asm to your project**

   The file should be located in the `c:\iw6000\labs\audioapp` directory.

3. **In edma.c, create a global integer variable named loadValue and initialize it to 1**

   We will use this variable to impose a simulated load of 1 microsecond on the system. The argument passed to the `load()` function represents the number of microseconds of load to add.
4. **Call load( ) from edmaHwi( )**

   Inside the `edmaHwi()` function, just after the if statement that tests the `xmtdone` and `revdone` local variables, add the following function call:

   ```
   load(loadValue);
   ```

   This function will take the argument passed to it (`loadValue`) and add approximately 1 microsecond of load per increment of `loadValue`.

5. **Rebuild and run your code**

   Let's see what effect this 1 microsecond delay has on our single-buffered system.

6. **Use the DIP switch to turn on the Sinewave**

   How does the system sound? If everything is working "correctly", it should sound fine. Why?

   When we made the buffers smaller at the end of lab 7, it bought us some time. (It also made the application work!) So now the question is "How much time do we have left before it breaks?". As we saw in the presentation, we don't have a whole lot of time left without using optimization. Let's use the load() function to get an idea of how much time we have left over.

7. **Add loadValue to a Watch Window**

   Find `loadValue` in `edmaHwi( )`, highlight it, right-click and select Add to Watch Window.

8. **Increment the loadValue by 1 until the system breaks (audio sounds bad)**

   Click in the text area next to the `loadValue` label in the Watch Window. This will select the current value for `loadValue` (1) and allow you to change it. Try incrementing the `loadValue` to 2. How does the music (and sine wave) sound?

   **Note:** You need to make sure that the sine wave is turned on for this part. If it is not turned on, you should be able to add quite a bit more load because the system is not generating the sine wave (which takes CPU cycles and time).

   Keep incrementing the `loadValue` until you hear the system break (ours broke around 10 for the 6416 and 6 for the 6713). How much load can the system handle before it starts to sound bad? _______________

   Now, that we know how to break the system (that's the easy part), let's leave the load in our code and add another buffer to our system. Using a double buffer system will give us a whole buffer time of samples instead of just the period between two samples. In other words, with a double buffer system, this load will be insignificant.

9. **Halt the DSK**
Part A – Double Buffering

In order to add double buffers we need to:

- Create 4 new buffers (a receive and transmit for both left and right channels)
- Allocate 2 new EDMA reload locations
- Change the EDMA initialization code to initialize the receive and transmit channels as well as the two reloads that go along with each (ping and pong).
- Change the EDMA hardware interrupt function so that it can keep up with what buffers (ping or pong) to process
- Use the load( ) function to see how much extra load the system can now handle
- Increase the buffer size to make future processing more efficient

Note: If you struggled with Lab 8 and couldn’t get it to work, copy the files from \solutions for c64x\lab8 or \solutions for c67x\lab8 into your \audioapp directory and begin with the next step shown below.

Add Double Buffers

10. Add new buffers and change the names of your current buffers

   In the global variables area, you need to change the buffer names to look like this:
   
   ```c
   short gBufRcvLPing[BUFFSIZE];
   short gBufRcvRPing[BUFFSIZE];
   short gBufRcvLPong[BUFFSIZE];
   short gBufRcvRPong[BUFFSIZE];
   short gBufXmtLPing[BUFFSIZE];
   short gBufXmtRPing[BUFFSIZE];
   short gBufXmtLPong[BUFFSIZE];
   short gBufXmtRPong[BUFFSIZE];
   ```

   Note: Don’t forget that the order of the buffers is important. Due to the way we are using the EDMA for channel sorting, the buffers for the Right channel need to follow immediately after their corresponding Left channel buffers.

11. Initialize all four transmit buffers to zero

   In main( ), add/modify the initialization code to zero BOTH transmit buffers (ping and pong).
Modify EDMA Handles, Configuration, Initialization

12. Add two #defines to help us keep track of what we’re doing:

Add the following definitions to edma.c.

#define PING 0
#define PONG 1

13. In edma.c, change the external references to the buffers

Near the top of edma.c, there should be 4 references to the old buffers (without ping/pong). Change these references so that they match the declarations in main.c.

14. Add New EDMA Handles

Both receive and transmit are going to require 3 EDMA handles each: one for the channel’s handle (hEdmaRcv), like before; one for the ping configuration; and the last for the pong configuration. So, you should have the following handles declared:

EDMA_Handle hEdmaRcv, hEdmaReloadRcvPing, hEdmaReloadRcvPong;
EDMA_Handle hEdmaXmt, hEdmaReloadXmtPing, hEdmaReloadXmtPong;

15. Modify EDMA Configurations

Locate the EDMA configuration gEdmaConfigRcv. Change the destination address from:

gBufRcvL to gBufRcvLPing

The reason we are setting this to gBufRcvLPing, is because we want to be receiving this buffer while we are transmitting gBufXmtLPing. This gets the double buffered system off to a good start. When gBufRcvLPing is full, we’ll copy gBufRcvLPing to gBufXmtLPing and transfer it. We’ll also copy the corresponding R Channel buffers that have been sorted by the EDMA for us.

Locate the EDMA configuration gEdmaConfigXmt. Change the source address from:

gBufXmtL to gBufXmtLPing

Now, the initial transmit channel is set up to transfer from gBufXmtLPing to the destination (soon to be DXR). The initial receive channel is set up to transfer from the source (soon to be DRR) to gBufRcvLPing.
16. Modify the receive EDMA channel initialization

Locate the `initEdma()` function. Add/change the following code for the receive EDMA channel initialization. Most of these changes should go from top to bottom in your code:

- For the `EDMA_allocTable()` function, change the reload handle to `hEdmaReloadRcvPong` and allocate another reload handle for receive's Ping.

- The first `EDMA_config()` is fine…it sets up the initial channel configuration. Change the second `_config` to use the new name `hEdmaReloadRcvPing`. Now the initial transmit channel and the receive's Ping reload entry are configured. Next, we’ll tackle the receive's Pong reload entry.

- For the receive's Pong reload entry, we need to change the destination address for the transfer. For Pong, we will be transferring from the McBSP’s DRR to the receive's Pong Buffer (`gBufRcvLtPong`). Add the following two lines of code just beneath the second `_config` for receive:

  ```c
  gEdmaConfigRcv.dst = EDMA_DST_OF(gBufRcvLPong);
  EDMA_config(hEdmaReloadRcvPong, &gEdmaConfigRcv);
  ```

  We continue to use the original `gEdmaConfigRcv` configuration and simply modify a few elements of the structure just before running `_config`. This is typically how it’s done. However, if you wanted to, you could have created two more complete `EDMA_Config` structures.

- Let’s finish the receive side by modifying/adding the `EDMA_link()` function calls. If you look back at the discussion material, you’ll see exactly how to link the channel to the reload tables and so forth. The initial channel links to pong, pong links to ping and ping links to pong. Remember, `EDMA_link()` API changes the link address field in the channel and reload table’s register set. You’ll need the following three `EDMA_link()`’s to accomplish this:

  ```c
  EDMA_link (hEdmaRcv, hEdmaReloadRcvPong);
  EDMA_link (hEdmaReloadRcvPong, hEdmaReloadRcvPing);
  EDMA_link (hEdmaReloadRcvPing, hEdmaReloadRcvPong);
  ```

- You’re now finished with the receive side.

17. Modify the transmit side EDMA initialization

In a similar fashion, modify the transmit side. Reference the discussion materials which has a nice drawing of what the receive side should look like. If necessary, add to the drawing your own comments (i.e. how each channel/reload table links to each other and what the src/dest addresses are for each transfer). This will help you make these modifications with fewer mistakes:

- make sure you have two transmit reload entries (ping and pong)
- configure the channel and reload entries (don’t forget to set up the source addr)
- link the transmit channel and reload entries properly
Modify the edmaHwi()

18. Set up the status flag to check ping or pong

Locate the edmaHwi() function. Add a local, static variable called pingOrPong and initialize it to PING.

19. Add four local pointers

We are going to manage which buffers get processed by using pointers and a very simple if/else statement. In edmaHwi( ), create four local pointers:

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

20. Add the proper if statement control code

Now that we have the local pointers created, we can create a very simple if/else statement to have them point to the correct buffers. For the PING case, we want them to point to the receive and transmit PING buffers. For the PONG case, we want them to point to the receive and transmit PONG buffers. After each case, we will need to switch the pingOrPong variable. We'll need to add this code inside the if statement that tests both the rcvInt and xmtInt flags. We only want to execute this code when we are going to process a buffer. The pseudocode looks something like this:

   if (pingOrPong == PING) {
      sourceL = gBufRcvLPing
      sourceR = gBufRcvRPing
      destL = gBufXmtLPing
      destR = gBufXmtRPing
      pingOrPong = PONG
   }
   else {  // pingOrPong must equal PONG
      sourceL = gBufRcvLPong
      sourceR = gBufRcvRPong
      destL = gBufXmtLPong
      destR = gBufXmtRPong
      pingOrPong = PING
   }

Note: If you’re uncomfortable with adding this control logic to the code, just copy it from the solution and continue.
21. **Change two SINE_add( )'s and the two copyData( )'s**

   When the code finishes executing the if/else statement that we just added, the active buffers are pointed to by the four local pointers that we added: sourceL, sourceR, destL, destR. This makes it easy to change the processing functions, the two SINE_add( )'s and the two copyData( )'s. Modify these functions to use the active pointer names instead of the globals that we have been using.

   **Hint:** gBufRcvL should become sourceL, gBufXmtL should become destL, etc.

---

**Build and Run**

22. **Build the project, debug, and load it the DSK**

23. **Turn on the DIP switch to add the sine wave**

   We want it in the system to do a comparison with the first part of this lab.

24. **Run the code**

   You should hear audio playing from your speakers and the sine “noise” added to it. In other words, the result of this lab is identical to the previous lab in terms of what your ear can hear.

25. **Increase the load on the system**

   As you can tell, the load() function is adding an insignificant amount of load. Before, using the single buffer system, a loadValue of a few microseconds broke the single buffer. Now that we have a double-buffered system, how much load will now break the audio stream? 10 times as much? 100 times as much? Try loadValue = 100 (adding 100 microseconds of load). Wow, it still works. In this system, using a double buffer allows more than 20 times the headroom than a single buffered system. Ours broke around 190uS.
**Increase Buffer Size**

26. **Use the extra headroom to process bigger buffers, increase the buffer size to 512**

   We are currently using pretty small buffers. Most of the time, it is beneficial to process big blocks of data. So, let's increase the buffer size of our system to 512 in both main.c and edma.c.

27. **Rebuild and run your code**

   Does it still work, even with the much larger buffers? It should. Feel free to play around with the loadValue. You should be able to increase it to really high values with the larger buffers. Now we are ready to do some real processing of this data.

28. **Remove the load**

   Comment out the load(loadValue) statement in your code and the loadValue declaration. Since we've proven that the double buffer system is more robust, we will not be using the load( ) function any longer to add a simulated load in the HWI.

29. **Save all of your changes**

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

   Using Windows Explorer, copy the contents of:
   
   c:\iw6000\labs\audioapp\*.*  TO  c:\iw6000\labs\lab8

   ![STOP](image)

   You’re done.