Introduction

In this module, you will learn how to use the BIOS scheduler and some additional debugging techniques provided by BIOS.

Learning Objectives

- Add a function to flash the LEDs and add a load
- Make “load” and “copy” operate simultaneously
Chapter Topics

DSP/BIOS Scheduling

Real-Time Problem
- Definition
- Possible Solutions
- HWI and SWI
- Tasks
- DSP/BIOS Threads - Summary

BIOS
- Enabling BIOS
- BIOS is
- Thread Scheduling
- SWI Properties
- Using a Mailbox
- Task Code Topology
- Periodic Functions
- How to Create A Periodic Function

Real-Time Analysis Tools
- Execution Graph, CPU Load Graph
- Statistics View, Message Log

Lab 9
- Part A
- Part B
- Part C
Real-Time Problem

Definition

Lab 9 Requirement - Abstract

Previous Requirement
- DSP pass-through and addSine

New Requirement
- Add function to flash LEDs and add load
- LED/Load independent of addSine/copy

Issues:
- Do we have enough bandwidth (MIPS)?
- Will one routine conflict with the other?

What are some possible solutions?
Possible Solutions

Possible Solution – while Loop

```c
main
{
    while(1) {
        addSine/copy
        LED/load
    }
}
```

- Put each routine into an **endless loop** under `main`
- Algos run at different rates:
  - `addSine/copy`: 94Hz
  - `LED/load`: 4Hz
- What if one algorithm starves the other for recognition or delays its response?

How are these problems typically solved?

Possible Solution - Use Interrupts (HWI)

- **An interrupt driven system** places each function in its own ISR

<table>
<thead>
<tr>
<th>Function</th>
<th>Period</th>
<th>Compute</th>
<th>CPU Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addSine/copy</code></td>
<td>11ms</td>
<td>7 μs</td>
<td>6%</td>
</tr>
<tr>
<td><code>LED/load</code></td>
<td>250 ms</td>
<td>100 ms</td>
<td>40%</td>
</tr>
</tbody>
</table>

46%

Interrupt is missed...

How could we prevent this?
Allow Preemptive Interrupts - HWI

- _Nested interrupts_ allow hardware interrupts to preempt each other.

- Use DSP/BIOS HWI dispatcher for context save/restore, and allow preemption
- Reasonable approach if you have limited number of interrupts/functions
- **Limitation:** Number of HWIs and their priorities are statically determined, only one HWI function for each interrupt

```c
main
{
  while(1);
}

Timer1_ISR
{
  addSine/copy
}

Timer2_ISR
{
  LED/load
}
```

What option is there besides _Hardware interrupts_?

Use Software Interrupts - SWI

- Make each algorithm an _independent_ software interrupt
- _SWI scheduling_ is handled by DSP/BIOS
  - HWI function triggered by hardware
  - SWI function triggered by software
    - for example, a call to `SWI_post()`
- Why use a SWI?
  - No limitation on number of SWIs, and priorities for SWIs are user-defined!
  - SWI can be scheduled by hardware or software event(s)
  - Defer processing from HWI to SWI

```c
main
{...
  // return to O/S;
}

DSP/BIOS
{addSine/copy
  LED/load
}
```

How do HWI and SWI work together?
**HWI and SWI**

**HWIs signaling SWIs**

- **HWI:**
  - Fast response to interrupts
  - Minimal context switching
  - High priority only
  - Can post SWI
  - Could miss an interrupt while executing ISR

- **SWI:**
  - Latency in response time
  - Context switch performed
  - Selectable priority levels
  - Can post another SWI
  - Execution managed by scheduler

**Tasks**

**Another Solution – Tasks (TSK)**

- DSPBIOS tasks (TSK) are similar to SWI, but offer additional flexibility
- TSK is more like traditional O/S task
- Tradeoffs:
  - SWI context switch is faster than TSK
  - TSK module requires more code space
  - TSKs have their own stack
- User preference and system needs usually dictates choice, easy to use both!

What are the major differences between SWI and TSK?
SWIs and TSKs

- Similar to hardware interrupt, but triggered by SWI_post()
- All SWIs share system software stack

SWI

start

“run to completion”

end

TSK

start

SEMs_post

Pause (blocked state)

end

- SEM_post() triggers execution
- Each TSK has its own stack, which allows them to pause (i.e. block)

DSP/BIOS Threads - Summary

DSP/BIOS Thread Types

<table>
<thead>
<tr>
<th>Priority</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HWI</td>
<td>Used to implement ‘urgent’ part of real-time event, triggered by hardware interrupt, HWI priorities set by hardware</td>
</tr>
<tr>
<td></td>
<td>SWI</td>
<td>Use SWI to perform HWI ‘follow-up’ activity, SWI’s are ‘posted’ by software, multiple SWIs at each of 15 priority levels</td>
</tr>
<tr>
<td></td>
<td>TSK</td>
<td>Use TSK to run different programs concurrently under separate contexts, TSK’s are usually enabled to run by posting a ‘semaphore’ (a task signaling mechanism)</td>
</tr>
<tr>
<td></td>
<td>IDL</td>
<td>Multiple IDL functions, runs as an infinite loop, like traditional while loop</td>
</tr>
</tbody>
</table>

C6000 Integration Workshop - DSP/BIOS Scheduling
Enabling BIOS

Enabling BIOS – Return from main()

main
{ ...
// return to BIOS
}
DSP BIOS
addSine/copy
LED/load

- The while() loop we used earlier is deleted
- main() returns to BIOS IDLE allowing BIOS to schedule events, transfer info to host, etc
- A while() loop in main() will not allow BIOS to activate

BIOS provides several capabilities...
BIOS is ...

DSP BIOS Consists Of:

- **Real-time analysis tools**: Allows application to run uninterrupted while displaying debug data.
- **Real-time scheduler**: Preemptive thread management kernel.
- **Real-time I/O**: Allows two-way communication between threads or between target and PC host.

Thread Scheduling

Priority Based Thread Scheduling

User sets the priority...BIOS does the scheduling

How do you create a SWI and set priorities?
**BIOS**

---

**SWI Properties**

---

**Managing SWI Priority**

- Drag and Drop SWIs to change priority
- Equal priority SWIs run in the order that they are posted

---

How do you pass information to SWIs?
Using a Mailbox

Pass Value to SWI Using Mailbox

- Each SWI has its own mailbox
- Why pass a value? Allows SWI to find out “who posted me”
- SWI_or() ORs value into SWI’s mailbox and posts SWI to run
- SWI_getmbox() inside SWI reads status of mailbox
- Other posts that use SWI mailbox:
  - SWI_inc(), SWI_dec(), SWI_andn()

Task Code Topology

- Initialization (runs once only)
- Processing loop - option: termination condition
- Suspend until unblocked
- Perform desired DSP work...
- Shutdown (runs once - at most)

- TSK can encompass three phases of activity (prolog, processing, epilog)
- TSKs can be blocked by using: SEM_pend, MBX_pend, SIO_reclaim, and several others (suspend execution until unblocked)
- TSKs can be unblocked by using: SEM_post, MBX_post, SIO_issue, etc.
Periodic Functions

Periodic functions run at a specific rate in your system:
- e.g. LED/load requires 4Hz
- Use the CLK Manager to specify the DSP/BIOS CLK rate in microseconds per “tick”
- Use the PRD Manager to specify the period (for the function) in ticks
- Allows multiple periodic functions with different rates
- Can be used to model a system (various functions w/loading)

Let’s use the Config Tool to create a periodic function...

How to Create A Periodic Function

Creating a Periodic Function
RealTime Analysis Tools

Execution Graph, CPU Load Graph

Built-in Real-Time Analysis Tools

- Gather data on target (3-10 CPU cycles)
- Send data during BIOS IDLE (100s of non-critical cycles)
- Format data on host (1000s of host PC cycles)
- Data gathering does NOT stop target CPU

Execution Graph

- Software logic analyzer
- Debug event timing and priority

CPU Load Graph

- Analyze time NOT spent in IDLE
Statistics View, Message Log

Built-in Real-Time Analysis Tools

Statistics View
- Profile routines w/o halting the CPU
- Capture & analyze data without stopping CPU

Message LOG
- Send debug msgs to host
- Doesn’t halt the DSP
- Deterministic, low DSP cycle count
- More efficient than traditional printf()

LOG_printf (&logTrace, "addSine ENabled");
Lab 9

In this lab, we’re going to change our copy routine to a SWI, add a routine to blink the LEDs and analyze other parts of our code using DSP/BIOS tools.

Open the Project

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

Add a SWI to Our Code

2. Open up main.c and, at the end of the code, add a new function and prototype named `processBuffer()`

   We are going to use `processBuffer()` as the function called by a SWI. It will do all of the processing that we have been doing in `edmaHwi()`. Don’t forget to prototype this new function.

3. Move pointers from the old HWI to the new `processBuffer()` function

   Remember the four local pointers that we created in the last lab: `sourceL`, `sourceR`, `destL`, and `destR`? We need to move **(not copy)** these pointers from `edmaHwi()` to `processBuffer()`. **DO NOT** move the static variable `pingOrPong`.

   The `edmaHwi()` function is in `edma.c` and `processBuffer()` should be in `main.c`. 
4. **Copy code from the old HWI to processBuffer( )**

The processBuffer( ) function will contain logic that is similar to the current HWI, so let’s borrow some of its code to get processBuffer( ) written. Locate the `edmaHWI()` routine in `edma.c`. **Copy** the following pieces of code from `edmaHWi( )` to `processBuffer( )`. Did we mention that you need to *copy* this stuff, **not move** it??

- if/else logic code that uses `pingOrPong`
- the SINE_add( ) function calls and its if statement
- the 2 calls to `copyData( )`

We are going to write the code for the function first, then when the code is written, we will add a SWI object to the .cdb file to call `processBuffer( )`.

5. **Delete code in edmaHWI()**

In `edmaHWI()`, delete the code inside the if `(pingOrPong)` and else statements that does the pointer assignments. Leave the code that modifies `pingOrPong`. We are going to replace some of this code with posting the SWI. The status variable, `pingOrPong`, will help us set the right mailbox value.

Delete the data copy routines and the “if” block that does the SINE_add statements.

All you should be left with is the code that tests the rcvDone and the xmtDone values inside an if statement, the two if statements that check these flags, and something like this:

```c
if (pingOrPong == PING) {
    pingOrPong = PONG;
} else {
    pingOrPong = PING;
}
```

This code should be inside the if statement that tests the rcvInt and the xmtInt values. We only want to use this code when both interrupts have occurred.
6. **Trigger the SWI to run and set a mailbox value in edmaHwi()**

In the `edmaHwi()` routine, you should now have the three if statements: one that checks for the receive interrupt, one that checks for the transmit interrupt, and the third that checks the two flags. Inside the third if statement you should have the `if (pingOrPong)/else statement and nothing else (maybe a few leftover braces). Inside the `if (pingOrPong), post a SWI named `processBufferSwi` (which we will create in a few more steps) and send it the mailbox value of PING. Inside the `else`, post the same SWI, but send it PONG. Make sure the code to swap the value of `pingOrPong` to the other state is still in place after posting the SWI.

**Note:** The APIs for posting a SWI expect a pointer to the SWI object (i.e. a handle). So, make sure and pass the **address** of the structure itself (the SWI_Obj) in the API call.

---

**Modify the processBuffer() Routine**

When the `processBufferSwi` is posted inside the HWI, it will post the routine, `processBuffer()`, to run. The `processBuffer()` function needs to do the following:

- get the mailbox value (is it PING or PONG ?)
- if PING, assign the source and destination pointers to the PING buffers, just like we did in the `edmaHwi()` before
- if PONG, assign the source and destination pointers to the PONG buffers
- add the sine values if the DIP switch is set
- copy the left and right buffers (using `copyData()`)

7. **Add a status flag to processBuffer() in main.c**

In `processBuffer()`, add a new status flag with a type of `Uint32` (instead of int), and name it `pingPong`. This new status flag, `pingPong`, will be used as the mailbox value.

8. **Get the mailbox value**

Use the proper API to get the mailbox value from the SWI’s object and place it in the variable `pingPong`. This should be your first line of code in the `processBuffer()` function.
9. **Change the if/else to use pingPong**

Modify the if/else statement in processBuffer( ) to use pingPong instead of pingOrPong.

The rest of this function should now be written and complete. First, you grab the mailbox value from the SWI object. If it is PING, you set up the PING buffers to be processed. Else, if PONG, you set up the PONG buffers to be processed. If the DIP switch is on, the SINE_add() functions are called. Finally, the assigned buffers are copied.

**Note:** Don’t forget to eliminate the two instructions that change the status of pingPong (held over from the HWI code). These instructions are no longer needed.

10. **Copy #defines for PING and PONG from edma.c to main.c**

We are now using these definitions in both places to help with the control code.

**Add the SWI to the System**

11. **Add the SWI to the CDB File**

Open the configuration file. Click on the + next to Scheduling and then the Software Interrupt Manager. Insert a new SWI object and name it, processBufferSwi. Set this new SWI to run the processBuffer function when posted. Set the initial mailbox value to “0” (PING).

Save the CDB file.

**Note:** Notice the naming convention here. The object is called processBufferSwi to denote that it is a SWI object that calls a function named processBuffer( ). Be careful not to use the same name for both of these. This will cause a symbol problem in the linker because there are two different addresses (one for the SWI object structure and another for the code) for the same label.

**Add Necessary Header Files**

12. **Add the BIOS generated header file to main.c**

The processBufferSwi object that we created in the steps above is referenced in both main.c and edma.c. BIOS provided a header file to help resolve this reference (and other BIOS references). The file is named audioappcfg.h.

Include this header file in both main.c and edma.c.

13. **Add BSL header files to main.c**

Since we moved the BSL call that reads the DIP switch from edma.c to main.c, we need to include the necessary header files in main.c. Go ahead and add these files to main.c.
Build and Run

14. Header File sanity check

Before you build, you might want to check to make sure that you've added all of the appropriate header files to the appropriate source files. Here is a short list to remind you which source files should have which header files at this point. If you don't have the right header files in the right place, you can get a bunch of build errors.

<table>
<thead>
<tr>
<th>Source Files</th>
<th>Header Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.c</td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
<tr>
<td>edma.c</td>
<td>&quot;dsk6713.h&quot; or &quot;dsk6416.h&quot;</td>
</tr>
<tr>
<td>mcbsp.c</td>
<td>&quot;edma.h&quot;</td>
</tr>
<tr>
<td>&lt;csl.h&gt;</td>
<td>&quot;dsk6713.h&quot; or &quot;dsk6416.h&quot;</td>
</tr>
<tr>
<td>&lt;csl_edma.h&gt;</td>
<td>&quot;sine.h&quot;</td>
</tr>
<tr>
<td>&lt;csl.mcbsp.h&gt;</td>
<td>&quot;codec.h&quot;</td>
</tr>
<tr>
<td>&lt;csl_irq.h&gt;</td>
<td>&quot;mcbsp.h&quot;</td>
</tr>
<tr>
<td>&quot;sine.h&quot;</td>
<td>&quot;dsk6713_dip.h&quot; or &quot;dsk6416_dip.h&quot;</td>
</tr>
<tr>
<td>&quot;edma.h&quot;</td>
<td>&quot;dsk6713_dip.h&quot; or &quot;dsk6416_dip.h&quot;</td>
</tr>
<tr>
<td>&quot;mcbsp.h&quot;</td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
<tr>
<td>&quot;dsk6713.h&quot;  or &quot;dsk6416.h&quot;</td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
<tr>
<td>&quot;dsk6713_dip.h&quot; or &quot;dsk6416_dip.h&quot;</td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
<tr>
<td>&quot;audioappcfg.h&quot;</td>
<td>&quot;audioappcfg.h&quot;</td>
</tr>
</tbody>
</table>
15. **Build, debug and run your code**

16. **Did anything happen?**

   Can you hear music? No? Well, unless you remembered to remove your while() loop, your code shouldn’t work. Remember? BIOS requires you to remove the while() loop and “return” from main. Once this occurs, BIOS will begin scheduling the SWIs and any other BIOS activities.

   Remove the while() loop (allowing the code to return from main() and fall into BIOS).

17. **Rebuild and run**

   You should hear the music again. If not, go back and debug some more or ask your instructor for help.
Part A

Note: If you struggled with getting Lab 9 to work, simply copy the files from \solutions for c64x\lab9 or \solutions for c67x\lab9 into your lab9 directory and begin at the next step shown below.

Add a Periodic Function

Next, we will add a periodic function that toggles the LEDs on the DSK (using another BSL API call) and to load the system with a series of NOPs.

18. Add a new function to your code called blinkLeds()

Open and inspect blinkTheLeds.c in the audioapp folder. This code blinks the LEDs in a sequential pattern and adds a load to the system. If you are using the 6713 DSK, change the dsk6416_led.h include appropriately as well as each call to the BSL library (change each instance of 6416 to 6713).

Add this file to your audioapp.pjt.

19. Add the new periodic function, blinkLeds(), to your CDB file

Open the configuration file and insert a new periodic object called blinkLedsPrd that calls the blinkLeds() function every 250 ticks. Click OK. Right click on the CLK manager, select properties and ensure that the default setting of 1000 microseconds/int is set. This sets the “tick” rate for BIOS and all periodic functions can be set up to fire after X number of ticks have expired. We’ll use the default setting of 1000 (or, 1 millisecond) for this lab. Click OK.

20. Build and Run.

Make sure that DIP switch #1 is down to enable the sine wave. Build and run your code. Do you see the LEDs flashing? What does your audio sound like? Would you purchase an audio system that sounded like that? 😊 Hmm. You should be hearing some problems in the audio – some noise perhaps. What do you think might be the problem? Well, if you know what it is…don’t fix it just yet. Let’s use some real-time analysis (RTA) tools to observe the operation of our code and debug it step by step. At the end, we’ll discover the exact reason why the audio stream has been basically broken.
Part B

Use Real-Time Analysis Tools

Next, we’ll use a few tools that might help us understand what is going on in our code. A few of these tools, such as the CPU load graph and execution graph are “ready to go” and require no additional coding efforts to use. The other tools require minimal code to work, such as LOG_printf().

21. Turn on the CPU Load Graph

On the menu bar, select:

DSP/BIOS → CPU Load Graph, or, click

The load graph will appear. You may want to float this window so that you can move it around and resize it to your liking. The load should be approximately 22%. So, you’ll notice that the CPU is not anywhere near 100% loaded, yet the audio still sounds unacceptable. Something is interrupting the audio stream.

22. View the Execution Graph

On the menu bar, select:

DSP/BIOS → Execution Graph, or, click

The execution graph shows the different threads in the system and when they occur relative to the other events. Remember, this graph is not based on time, but on events, i.e. when “something” happens (like when a SWI or periodic function runs). This graph is sometimes useful in helping you debug the timing of your system. In our case, the “problem” with the audio doesn’t show up necessarily in the execution graph.

23. Use the Statistics View

On the menu bar, select:

DSP/BIOS → Statistics View, or, click

The Statistics View gives detailed timing information about each of the DSP/BIOS threads in your system. Take a look at the processBufferSwi thread. The interesting element to observe is the max field. This number tells us the maximum amount of instructions that the processBuffer() function takes from the time that the SWI is posted until the time that it takes to finish executing. If this max number is ever greater than the size of your buffers times the sample rate, then your system has missed real-time. This is obvious when you listen to the audio. If you would prefer to see the statistics view based on time, rather than instructions, you can change the properties of the display.
24. **Change priority of the processBuffer() SWI vs. the periodic function**

OK. So, what’s the solution? Have you thought about how the processBuffer() function is prioritized? Is the periodic function set at a higher, lower, or equal priority? Let’s take a look. Open the audioapp.cdb file. Click on the + sign next to Scheduling. Click on SWI – Software Interrupt Manager. In the right hand window, you’ll see the SWI priority list where 0 is the lowest and 14 is the highest priority. What is the current setting? Which is more important – the processing of the audio or the blinking of the LEDs? Assuming that the answer is “the audio”, we need to set its priority higher than the LED blinking. By the way, if SWIs are set at the same priority, they execute in a first in, first out fashion.

Click and drag processBufferSwi to Priority 2 and release it. The audio is now higher priority.

Close the .cdb file.

25. **Build, Load and Run**

Your audio should sound MUCH better now and the LEDs should be blinking normally. The “enable sine” switch should also work flawlessly.

**Use LOG_printf() to display status of the DIP switch**

The LOG_printf() function is a very efficient means of sending a msg from your code to the CCS display screen. It is used during debug to send a msg to the PC host saying “we got to the filter ISR” or “the status of the DIP switch is UP”, etc. Let’s use this tool to send a msg to CCS’s display and state whether the sine is enabled or disabled.

26. **Add a trace buffer to your system**

Open the .cdb file and click on the + sign next to Instrumentation. Right click on the LOG – Event Log Manager and click Insert LOG. Rename LOG0 to logTrace. Save the .cdb file.

27. **Add LOG_printf() to the logic around addSineSorted()**

In the processBuffer( ) function, modify your code so that it looks like this:

```c
if (DSK6416_DIP_get(0) == 0) {    // DIP switch 0 is on (down)
    SINE_add(&sineObjL, sourceL, BUFFSIZE);
    SINE_add(&sineObjR, sourceR, BUFFSIZE);
    LOG_printf(&logTrace, "addSine ENabled");
} else {
    LOG_printf(&logTrace, "addSine DIISabled");
}
```

28. **Remove the load function**

Because we don’t need this load function anymore, comment out the call to load() from blinkTheLeds.c and REMOVE load_6416.asm (or  load_6713.asm) from your project.

29. **Rebuild and run**
30. View the Message Log

Select:

DSP/BIOS → Message Log, or, click

Move and resize to your liking. The log name will default to logTrace. If you had multiple trace buffers, you could select them here. You should see a series of msgs in the window. Right click in the Message Log window and select *Automatically scroll to the end of buffer*. Toggle the DIP switch up and down and see what happens. If you’re not getting any msgs, make sure your code is running and you hear the audio.

When finished, move on to part C where we will switch the SWI to a TSK…
Part C

Using a TSK Instead of a SWI

Now, we’re going to switch the SWI to a TSK. There are several things that a TSK can “block” or pend on – in this case, we’re going to use a semaphore. Because TSKs do not have a mailbox (like SWIs do), we need to use a global variable to pass the status of pingOrPong between the HWI and processbuffer( ). However, using a global variable means that the status of PingOrPong changes instantly.

The first time we enter the edmaHwi( ), the PING buffers are full. So, we want to post PING to the TSK. We must, however, switch the state of pingOrPong before doing the SEM_post because the global variable changes instantly (vs. using the mailbox within the SWI). So, we need to initialize pingOrPong to PONG, then switch it back to PING prior to the first SEM_post…so it processes PING when PING is ready.

31. Remove the processBufferSwi object.

Open the .cdb file and under Scheduling, delete the processBufferSwi object.

32. Add a TSK called processBufferTsk.

Under Scheduling, insert a new TSK and name it processBufferTsk. Change the TSK function to _processBuffer.

33. Add a semaphore for the TSK.

Under Synchronization, insert a new SEM called processBufferSem. When finished, close and save the .cdb file.

34. Change the SWI or statements to use SEM_post.

In edma.c, find edmaHwi( ). Replace the entire if/else construct for pingOrPong with the following (note the arrows are what you need to change):

```c
if (xmtDone && rcvDone)
{
    pingOrPong = !pingOrPong;
    SEM_post(&processBufferSem);
    rcvDone = 0;
    xmtDone = 0;
}
```

35. Make pingOrPong a global variable

In order for processBuffer( ) to “see” the pingOrPong variable, we need to make it global. In edmaHwi( ), delete the assignment for pingOrPong and add it to the global variables are of edma.c:

```c
int pingOrPong = PONG;
```
36. Replace SWI_getmbox with SEM_pend.

Open main.c and find processBuffer(). Replace SWI_getmbox with the following. SYS_FOREVER is the timeout value for the semaphore – i.e. we are waiting “forever” for the semaphore to post.

\[
\text{SEM_pend}(&\text{processBufferSem, SYS\_FOREVER);}
\]

37. Change pingPong variable to pingOrPong.

In processBuffer(), delete the assignment of pingPong. Where pingPong is used in the code, replace it with pingOrPong (now that the SWI mailbox doesn’t exist – we are using pingOrPong as a global). In the global declarations area of main.c, add the following extern to use pingOrPong:

\[
\text{extern int pingOrPong;}
\]

38. Insert a while(1) statement in processBuffer().

This while( ) statement will enclose the entire code inside processBuffer( ). Add the following:

\[
\text{while(1) }
\]
\[
\{
\}
\]
\[
\text{SEM_pend...}
\]
\[
\text{if (pingOrPong == ...}
\]
\[
\text{...}
\]
\[
\text{...}
\]
\[
\text{copyData(sourceR, destR...}
\]

39. Build your code and fix any errors.

40. Once you have a clean build – load/run. Everything should operate normally.

42. Copy project to preserve your solution.

Using Windows Explorer, copy the contents of:

c:\iw6000\labs\audioapp\*.*  TO  c:\iw6000\labs\lab9

You’re done.