Department of Electrical and Electronic Engineering
Imperial College London

EE3-19, EE9AO10, EE9CS7-27, EE9SC5

Real Time Digital Signal Processing

Course homepage: http://learn.imperial.ac.uk

Lab 3 – Interrupt I/O

Paul D. Mitcheson
paul.mitcheson@imperial.ac.uk
Room 1112, EEE

Imperial College London

Objectives

- Help you to understand the concept of interrupts (this knowledge is essential for the remaining laboratories)
- Learn how to use interrupt driven code first for audio input then for audio output.
- Configure the signal generator so that its signal does not damage the audio inputs on the DSK.

Data sampling using interrupts

In this first exercise you will use interrupt driven code to sample an input waveform and output it again after full-wave rectification.

Creating a new project from the previous lab

- Using windows explorer make a copy of the ***lab2*** project folder you created in the previous lab and paste it under the ***RTDSPlab*** folder. This will create a folder called ***lab2 – copy***
- Rename the ***lab2 – copy*** folder to ***lab3***
- Navigate to ***RTDSPlab\lab3\RTDSP*** and delete ***sine.c***
- Download the ***lab3.zip*** zipfile from Blackboard. Unzip ***intio.c*** into ***RTDSPlab\lab3\RTDSP***
- Recycle the power on the DSK hardware and ensure the USB lead is connected.
- Start up CCS (if CCS is already open use **file→Switch Workspace** to the same effect). When asked to choose a workspace hit the browse button and select the ***lab3*** folder you just created. See figure 1.

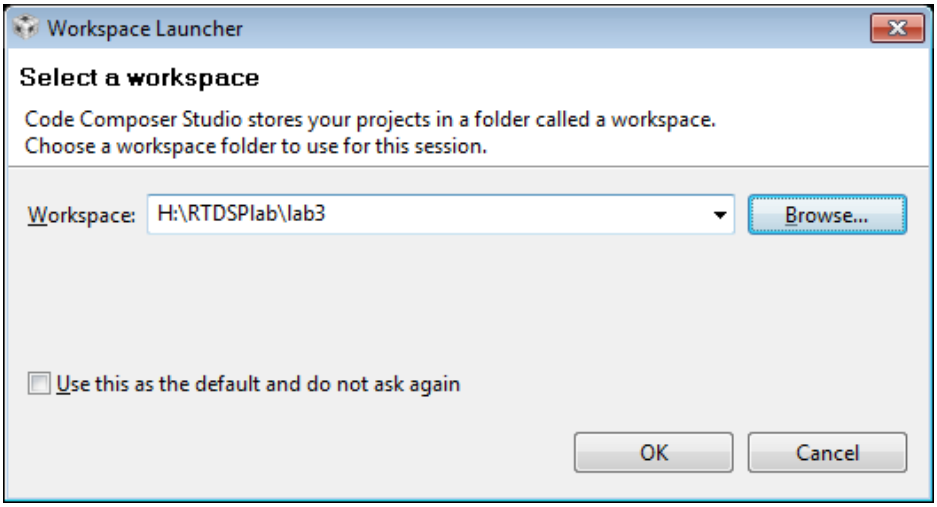

Figure 1: Selecting the lab3 project

- Hit **Ok**

Using the Configuration tool

If you cast your mind back to lab 2 we mentioned that the configuration tool is a graphical environment for modularly adding the software components you require for a particular project. For this laboratory we are going to extend the use the tool to manage some of the tasks required when using interrupts.

- The first step in configuring hardware interrupts is to setup the link between the physical interrupt and the ISR[1] (which you are yet to write). Click on the C/C++ projects icon on the bottom of the CCS window. Double click on the ***dsp_bios_.tcf*** file (shown in figure 2) file to open the configuration tool. Expand the **Scheduling** directory in the configuration window by clicking the **+** symbol. Now expand **HWI** in a similar manner. You should see a window listing a number of interrupts as of figure 3 (there are 16 different interrupts in total). Before you link the audio port interrupts to some code I want to explain this window to give you a clearer understanding of interrupts.

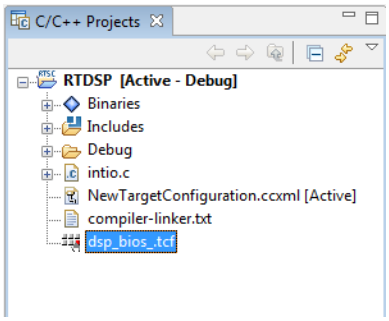

Figure 2: C/C++ Projects window showing tcf file.

[1] ISR: Interrupt Service Routine i.e. the code that is executed when an interrupt occurs.

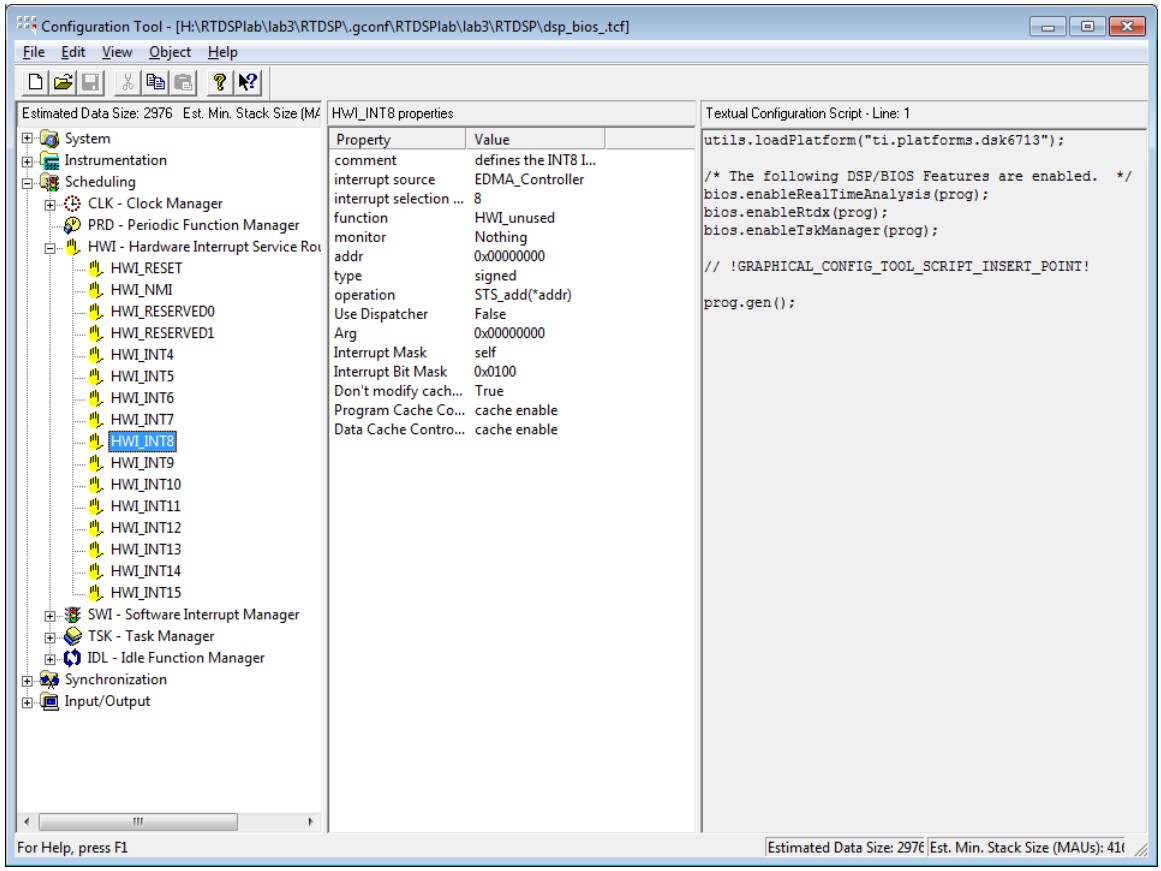

Figure 3: Interrupt management window in the BIOS configuration tool

- Right Click on the interrupt labelled **HWI_RESET** and choose **properties** from the pop up box. This interrupt occurs whenever the reset button on the DSK is pressed. Notice the text box called function (at the moment containing the default text **_c_int00**). In simple terms this window allows you to assign some code you have written within a function called **_c_int00** (or other name you have chosen) to the physical event[2] of clicking the reset button. Cancel this window as we are not interested in running code when the reset button is pressed but when a sample arrives at the audio port.

Notice that there isn't a specific interrupt labelled "audio port" in the interrupt list (as you may have been expecting). The DSP BIOS tool allows more flexibility than this. An audio port interrupt can be assigned to any of the interrupts **HWI_INT4** to **HWI_INT15**. The higher the interrupt in the list (i.e. the lower the number) means that the greater the priority of the interrupt.[3]

1. Choose **HWI_INT4** for your ISR by right clicking it and choosing **Properties**.

[2] Remember the term **event**, as it is extensively used in interrupt programming parlance, for example interrupt driven programs are sometimes called event-driven programs.

[3] This means that code bound to HWI_INT15 will relinquish execution (but complete later) to code bound to HWI_INT4 should the events attached to these interrupts occur around the same time. This is called pre-empting.

2. In the box labelled **interrupt source** browse the list for a while. Notice that there are many physical events that can be assigned to this interrupt. The one we require is **MCSP_1_Receive**[4] (do not confuse with MC**A**SP1). Choose this from the drop down list. This event occurs whenever the audio port has a sample ready to be read.
3. Enter the name of the ISR routine in the box labelled **function** overwriting the default text **HWI_unused.** In our case the ISR will be a C routine, so its name must have a leading underscore, e.g. **_YourChosenFunctionName**. (I chose ***_ISR_AIC***). Enter a function name of your choice (do **not** use **_ISR** since this is a reserved label). You will later write a C function with the name you have chosen that services the interrupt (and remember that in the C code you will prototype and define the function **without** using the "_" character). Lastly, click the **Dispatcher** tab then check the **Use Dispatcher** box so that the configuration tool will automatically take care of saving/restoring the context when an interrupt occurs. Click **Apply** then Click **OK** to remove the dialog box.
4. Choose **file→Save** from the (configuration tool) menu to save the BIOS configuration settings. This automatically creates a section of code based on the settings you have made called an interrupt vector table. (You would have had to explicitly write this code had you not used the BIOS configuration tool.)
5. You can now close the configuration tool.

[4] Again, you may have been expecting something that referred to the audio port specifically! MCSP stands for Multi Channel Serial Port which connects between the Audio port and the DSP chip. This system is more flexible as the MCSP can be configured to connect to other hardware via J3.

The C program shell

You are provided with a template program ***intio.c*** (which you added to the project earlier). Open this file in the CCS window and work out what the code does. The notes below and the comments in the code will help you with this task and indicate what level of understanding is required.

Execution starts at main(). Firstly, general hardware settings are set through the function called Init_hardware(). Briefly, this function:

1. Initializes the board support library through calls to the library function DSK6713_init() (All you need to understand is that this function must always be called first to reset all the hardware to the default settings).
2. Initializes the codec (audio port). First, it uses the function DSK6713_AIC23_openCodec() to open the on-board AIC23 codec (audio port). This function requires as an argument the configuration structure defined at the beginning of the program (called Config). The configuration structure is used by the function to set the audio ports bit resolution and sampling frequency (amongst many other things! The important settings to notice are that the bit resolution is 16 bit and the sampling frequency is set to 8K.
3. Next there are four calls to a library function called MCBSP_FSETS(). This general purpose function is for setting the registers on the multi-channel buffered serial port (this port connects between the AIC23 audio port and the DSP). The function uses pre-defined library symbols for the arguments. These functions ensure that the word length read from the multi channel buffered serial port is 32 bits wide and an interrupt is generated after each 32 bit data packet is available. We have provided a function to unpack this 32 bit data into 16 bit[5] left and right channels and then convert the two channels into a 16 bit mono signal.

[5] If a larger bit resolution is required the settings on both the AIC23 (by modifying the config structure) and the MCBSP will need to change (by modifying the MCBSP_FSETS() function calls). Larger bit resolutions will not be explored in this laboratory.

Interrupt settings are configured through the function init_HWI() which does the following:

1. All of the interrupts are disabled before changes are made to interrupt configurations, this is done with the library function IRQ_globalDisable()
2. The IRQ_nmiEnable() enables the non-maskable interrupt. (This interrupt is used by the debugging tools.)
3. IRQ_Map() maps the physical event of a the MCBSP receive interrupt to a specific interrupt priority, in a similar manner to the setting you made in the BIOS Configuration tool. You should ensure that the interrupt priority number here (currently **4**) matches the interrupt number (HWI_INT**4**) you chose when setting up the configuration file.
4. This particular interrupt (MCBSP receive) is then enabled through the function IRQ_Enable() by using a symbol (IRQ_EVT_RINT1) that represents the event generating the interrupt.
5. All interrupts are then enabled through the function call IRQ_globalEnable(). It is important that all interrupt initializations you require are performed before the call to IRQ_globalEnable(), since as soon as interrupts are globally enabled an interrupt could potentially occur.

Finally the program enters an endless while loop:

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

The program waits here for interrupts to occur.

When an interrupt occurs the context of this while loop is saved (i.e. the current program counter position, and register values) and the ISR is then loaded and executed (this is called a context switch). After the ISR code has completed this while loop is then re-loaded from exactly the position it was pre-empted.[6]

[6] You could if you wished add some code in the while loop to do some other task, for example flashing LED's on the board and it would seem as if the processor was doing two tasks at once; servicing the audio ports and flashing the LED's.

Exercise 1: Interrupt service routine

You should now write a function that services the interrupt (don't forget to write the prototype for the function also!) This function should have the same name as the function name you assigned in the configuration file above (remember that you do not want to use the "_" in the name at this point). This function should perform the following operations:

1. Read in a sample from the codec. For this purpose you can use the special function we have provided for you called mono_read_16Bit(). This simply reads a Left and Right sample from the audio port, divides the amplitude of them by two and then sums the samples together to provide a mono input. This function returns this mono result as a 16 bit integer.
2. Full-wave rectify the sample.
3. Write out the rectified value. Again you can use the special function mono_write_16Bit(samp). Note that the integer argument samp must not exceed 16 bits. This function simply takes the argument samp and sends copies of it to the left and right channels to achieve a mono output.

When you have finished programming build the project **Project→Rebuild Active Project** Once the project has successfully compiled and linked, it will create a file ***intio.out*** that is the DSK executable file. Make sure you have successfully compiled the program with no errors before continuing on to the next step.

Run the program

- Use the menu **Target→Debug Active Project** to open the debugger, connect to the hardware and load the ***intio.out*** file into the memory of the DSK hardware.
- Run the program using **Target→Run**

Connecting a signal source to the hardware and running the program.

It is important that you read the next section carefully. Failure to do so may result in damage to the hardware. You may use either a hardware signal generator or an software signal generator (recommended as it is more convenient and takes less space)

Do not under any circumstances connect the signal generator to the audio port without setting the waveform (to a suitable voltage level and frequency) on an oscilloscope first! This is especially important if you are using a dedicated signal generator to source the signal. A software signal generator is provided on the PC (which is a safer option as the signal level is restricted in amplitude). The line **inputs** on the audio port can take a maximum $\mathbf{2V_{RMS}}$ and the sampling frequency has been set to 8 KHz. So set the signal generator to a sine of frequency between 10 Hz and 3.8 KHz and not exceeding 5.66 V peak to peak. You should know why these limits are chosen.

Ensure that this signal is split into two and connected to the left and right line inputs. (If you are using the software signal generator then connecting a stereo mini jack cable from the PC to the line in jack will achieve this). Measure the signal on one of the line output connectors. You should see a full-wave rectified signal if you programmed the code successfully.

- **Why is the full rectified waveform centred around 0V and not always above 0V as you may have been expecting? (see figure 4 for clues)**

[1]

- **Note that the output waveform will only be a full-wave rectified version of the input if the input from the signal generator is below a certain frequency. Why is this? You may wish to explain your answer using frequency spectra diagrams. What kind of output do you see when you put in a sine wave at around 3.8 kHz? Can you explain what is going on?**

[10]

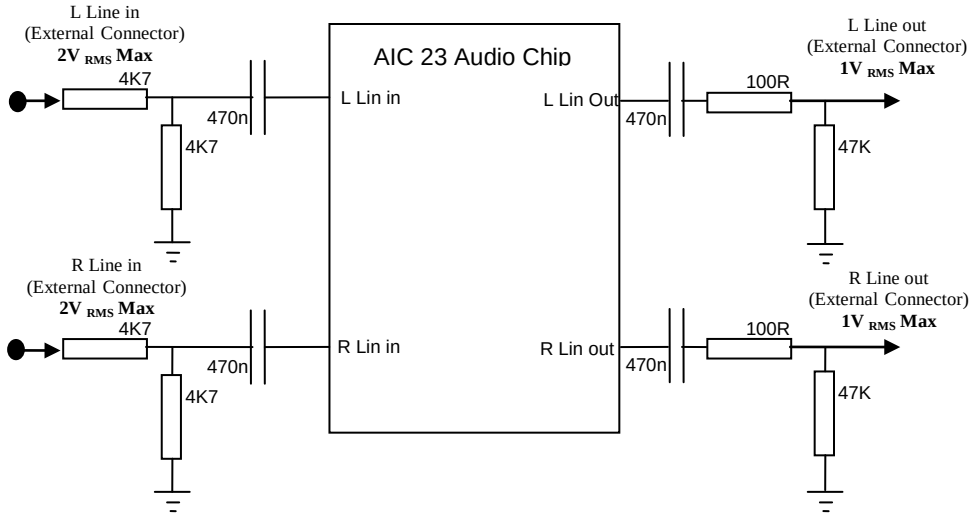

Figure 4: AIC23 Audio chip external components (adapted from TMS320C6713 Technical ref (page A-14, 2003 revision A)

Exercise 2: Interrupt-driven sine wave

Modify the program from Exercise 1 so that within the interrupt routine a sine wave is generated using the lookup-table method from Lab 2. As well as modifying the interrupt service routine, you will need to change the interrupt source so that the ISR is entered when the MCBSP is ready to write a sample (rather than when it has received a new sample). This will require the following modifications:

1. Modify the configuration file (in the configuration file browse tool) so that the ISR is associated with the MCSP_1_Transmit rather than MCSP_1_Receive interrupt.
2. Modify the function init_HWI to reflect the changes you made to the configuration file. Note that a transmit interrupt is denoted by the symbol IRQ_EVT_XINT1.
3. Rectify this sine wave before outputting it.

Make sure you have the code working and set up properly before moving onto the next lab. This lab will provide a template for the next lab so it is essential that you have the code working.

Deliverables

You do not have to write a formal report including abstract, conclusions etc for this lab. You are, however, required to write a tidy report which provides program listings and evidence (e.g. graphs) that you have done the exercises. Ensure you cover the points made below.

Marks are awarded as follows:

- The 2 questions asked above in **Arial Bold Text**

  [11]

- Paragraph explaining the operation of your code for both exercises, using code snippets if required.

  [7]

- Scope traces showing your code operates as expected and showing the edge cases of the frequency of operation.

  [3]

- Full readable code listing with comments (as an appendix to your report).

  [4]

Please submit your assignment formatted as a pdf/word/other turnitin compatible document via the blackboard upload page for lab3.

Revision History

11th Feb 2009
8th Feb 2010 – Added mark allocation information
3rd Nov 2010 – Rewritten for CCS v4 and Win 7
3rd Jan 2012 – Minor tweaks
2nd Jan 2013 – Minor tweaks