EECS 452 Lab 6:
Interrupts, FFT and Graphics

In this lab we will transition to more directly useful work: FFTs, Interrupts and FPGA/processor communication/graphics.

1. Background

In lab 5 you had the chance to play with fairly trivial interrupt code. Ideally you learned some of the basics of interrupts, but this section will provide a more detailed overview.

At the highest-level interrupts can be thought of as a hardware mechanism for causing a function to be called. Specifically, when an event occurs the processor:

- Stops what it is doing
- Saves some state information so it can get back to what it was doing
- Starts running code associated with that event.

Once the code (function) associated with that event finishes it issues a "return from interrupt" instruction and it:

- Restores the state information it saved
- Jumps back to what it was doing

This process is actually fairly complex. At the least we need some way to determine which events are allowed to cause interrupts (enabling the interrupt) and a way to specify what to do when a given interrupt does occur.

Step one: When the event occurs

This step is actually fairly easy. Each event that can cause an interrupt has a single bit associated with it. If that event occurs, the associated bit becomes a "1". Note that this happens independently of what instruction the processor is currently executing. These bits are kept in two registers, "interrupt flag register 0" and "interrupt flag register 1". In the C5515 they are laid out as follows:

Figure 1-25. IFR0 and IER0 Bit Locations

15	14	13	12	11	10	9	8
RCV2	XMT2	SAR	LCD	PROG3	Reserved	PROG2	DMA
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0
PROG1	UART	PROG0	TINT	INT1	INT0	Reserved	
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

Figure 1-26. IFR1 and IER1 Bit Locations

15					11	10	9	8
Reserved						RTOS	DLOG	BERR
R-0						R/W-0	R/W-0	R/W-0
7	6	5	4	3	2	1	0	
I2C	EMIF	GPIO	USB	SPI	RTC	RCV3	XMT3	
R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	

LEGEND: R/W = Read/Write; R = Read only; *-n* = value after reset

It's probably not clear what all these events are, but they are described in the "C5515 DSP System Guide (p54-55), which is where the figures are taken from. In particular the TINT bit is set whenever a timer event occurs. One might use the following code to enable the timer interrupt:

```
//IER0 is memory location 0.
#define IER0            *(volatile Uint16*)(0x0000)
#define TINT_BIT 4

IER0 |= (1 << TINT_BIT);
```

Step two: Enabling interrupts

Just because a given event *can* cause an interrupt doesn't mean it *will.* We must first enable each potential interrupt source. This is done by setting the appropriate bit to a "1" in "Interrupt Enable Register 0" or "Interrupt Enable Register 1" (IER0 & IER1). These are laid out in exactly the same way the interrupt flag registers are (in fact you'll notice that the figures above are labeled as both IFR and IER). So having the event occur (and the flag bit therefore be a "1") isn't enough. The interrupt must be enabled.

But even enabling a *given* interrupt isn't enough—there is also a global interrupt enable. It is bit 11 of "ST1_55", yet another system register. That bit is called "INTM" which stands for "interrupt mask". If the bit is a "1" the interrupt will be masked (that is prevented from occurring). That bit is set to "1" (disabling all interrupts) on boot. See page 2-39 of the C5515 DSP CPU guide. This entire concept is illustrated in the following figure:

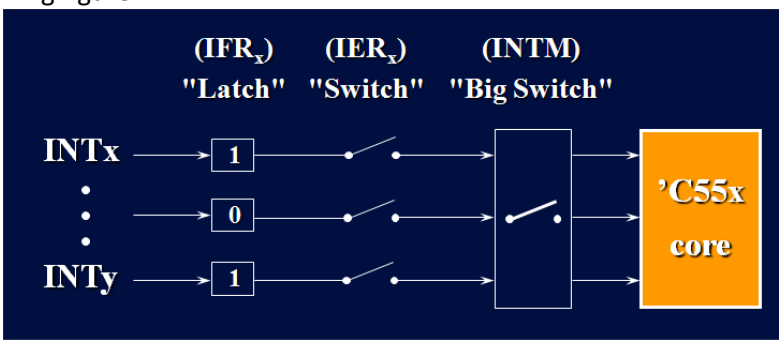


Figure 1: This figure illustrates how the IFR, IER, and INTM interact.

Typically one clears the INTM bit by using a macro called **`_enable_interrupts()`**.

Step three: Where to go

This part is probably the trickiest. Once we *get* and interrupt and it's *enabled* and thus allowed to go to the processor core, how does the core know what to do? The answer is that we create a table in memory that says where to go for each interrupt we care about (assumedly that would be any that we enabled). The start of the table looks like this (taken from Page 53 of the C5515 DSP System Guide):

Table 1-32. Interrupt Table

NAME	SOFTWARE (TRAP) EQUIVALENT	RELATIVE LOCATION (HEX BYTES) (1)	PRIORITY	FUNCTION
RESET	SINT0	0x0	0	Reset (hardware and software)
NMI(2)	SINT1	0x8	1	Non-maskable interrupt
INT0	SINT2	0x10	3	External user interrupt #0
INT1	SINT3	0x18	5	External user interrupt #1
TINT	SINT4	0x20	6	Timer aggregated interrupt
PROG0	SINT5	0x28	7	Programmable transmit interrupt 0 (I2S0 transmit or MMC/SD0 interrupt)

So the table entry for the timer interrupt is 0x20 (32) bytes from the start of the table. Each table entry has 8 bytes allocated to it. We only use bytes 1-3. We set those bytes to the address of the function you wish to have called when that event occurs[1].

But where is the head of that table? Great question! The answer is it's programmable. There are two special purpose registers that store the top 16 bits of the table. Wherever those register point, that's where the table is (the bottom 8 bits of the address are all zeros). We normally have the exact location selected by one of our configuration files and have it passed to us as a function address named Reset(). Yes, that's pretty convoluted, but it's TI's standard way of handling this. So we get code that looks like this:

```
#define IVPD          *(volatile Uint16*)(0x0049)
#define IVPH          *(volatile Uint16*)(0x004A)
Uint32 reset_loc = (Uint32)Reset;
IVPD = reset_loc >> 8;
IVPH = reset_loc >> 8;
```

Why do we have both IVPD and IVPH? That's actually fairly complex and has to do with multi-processor/operating systems issues*. In general we just keep them both the same value[2].

Now that the table has been located somewhere in memory, we just need to set the address we want to branch to in the case of an interrupt in the right location. Say that when the timer interrupt occurs we want to branch to an instruction named "Timer_Handler()". We would do the following (assuming we had set reset_loc correctly as in the code above):

```
#define TINT   0x20
*((Uint32*)((reset_loc + TINT)>>1)) = (Uint32)Timer_Handler;
```

That code is fairly complex. In particular what we are doing is putting the address of Timer_Handler into the location reset_loc+TINT. But the right shift by 1 is a bit weird—why are we dropping the least significant bit? The answer is that the TI processor addresses *instructions* using byte (8-bit) address and *data* as word (16-bit) address. So we need to convert this program address to a data address. Yes, it's weird.

[1] Bytes 4-7 can be assembly instructions (useful to optimize the runtime of the interrupt) and byte 0 can be used to set the stack mode. You can read spru599d.pdf page 20 if you want to learn about stack modes…

[2] Interrupt vectors for interrupts 0-15 and 24-31 are relative to IVPD. Interrupt vectors for interrupts 16-23 are relative to IVPH.

One other note: the function to be called must be declared to be of type interrupt and must return a void. It may not take arguments.

```
interrupt void Timer_Handler()
{
      //stuff goes here

}
```

2. Pre-lab

In this pre-lab you will be prepared for working with interrupts, working with TI's FFT functions and dealing FFT windowing.

Interrupts

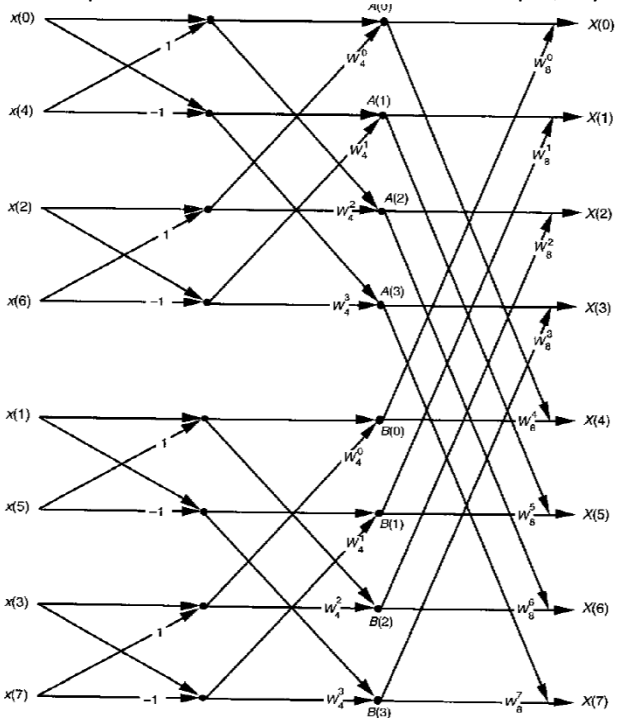
The previous section went into some detail about interrupts on the C5515. In order to demonstrate your understanding, answer the following question.

Q1. Write a short C program that prints the word "hi" every time the I2S2 transmit interrupt occurs. Don't worry about what that interrupt is, but do be sure to enable it and otherwise set up all the parts needed to get it to work. Model your answer off of the code we used in lab 5.

Q2. In your own words, explain why interrupts are useful and common on both modern desktop processors and embedded processors. This should take a couple of paragraphs.

TI's FFT functions in the DSPlib

One interesting issue with doing FFTs is that it is generally the case that the input is in one order but the output is in a different order. For example, if you look at the butterfly network shown below, the outputs are in "normal" order but the inputs are not.



Q3. The order these inputs are found is often called "bit reversed" why is that named used? What bits are reversed?

Q4. Consider doing an 8-point FFT on complex numbers using the cfft() function in DSPlib. List all 16 outputs (real and imaginary parts) in the order they would come out. Your answer should be of the form R0, C0, R5, C5, R2, C1, etc.

Q5. Explain what cbrev32() does. Explain why that's helpful. (This is pretty straightforward).

Q6. The rfft32() function takes nx real inputs and generates nx/2 complex outputs (and thus takes the same amount of memory for the input and output even though the input is restricted to only real numbers). Why is this possible? Why doesn't it apply to the cfft() function?

Q7. What is "scaling" and why do some of the TI dsplib FFT functions support it? How much do they scale by at each stage and why does that make sense? How do you get rfft32() to use scaling?

Q8. This question asks you to think about doing FFTs on real-time audio data. For this problem, assume we are sampling 16-bit inputs at 48KHz and that our processor runs at 100MHz.

a. About how many CPU cycles do we have between our samples?
b. How long (in cycles) would we expect it would take cfft() to handle an input with 1024 entries? Can we do this between two data samples? What is the largest radix 2 FFT (if any) we could do between samples?
c. If we want to do a 1024 FFT after every 1024 samples, how can we manage that without missing some of the audio samples?

Chebyshev window file generation

Q9. Why do we use windowing when working with FFTs? Provide a fairly detailed answer (a couple of paragraphs). Include some discussion about leakage and how windowing affects leakage.

Q10. Write a MATLAB script to generate 3 files containing values of the Chebyshev window with 100 dB relative side-lobe attenuation. You will need these values in the lab so the files could be C header files or just text files but then you will need to read in these values in your code. Use the chebwin() function in MATLAB. Values of N and the files names to be used are:

N	name
512	cheby100N512.txt
1024	cheby100N1024.txt
2048	cheby100N2048.txt

Multiply the chebwin values by 32767 then round in order to generate 16-bit Q15 values. These can be written out as integer values.

In-lab

During this lab you will not only get hands-on experience with the FFT, you will also get an introduction to FPGA/processor communication and yet more experience with interrupts. First, let's get a bit of background information about the C5515's audio system.

On our C5515 ezDSP stick, the audio chip (AIC3204) is connected to the C5515 via an I2S bus.

Q1. What is an I2S bus?

Q2. In the "C5515 DSP System Guide" the I2S0 interrupt settings are discussed. If the external bus selection register is set correctly:

a. Is the flag register for this interrupt IFR0 or IFR1? Which bit of that register?
b. What is the table offset for that interrupt?

Part 1: Using TI's FFT

In the pre-lab you looked over the use of the DSPlib's FFT functions. We have written a bit of code (found on Ctools) which takes data in via the stereo in and then outputs one component of the FFT on one channel of the stereo out. Answer the following questions about this code.

Q3. What are we doing in the I2S handler (I2S_ISR)? What is going out on the left channel? The right one?

Q4. What is the variable "Counter" doing? How many samples must occur before we start the FFT?

G1. Explain to your GSI what this program is doing. Be sure to address

a. Your answers to Q3 and Q4 above.
b. Which frequency in the FFT is being examined and where it is being output.
c. How the ISR (interrupt service routine) and the main are interacting.

Ideally we want to check the magnitude of that frequency bin using AIC output, however the AIC is AC coupled so if we output a relatively DC signal, what we see on the scope won't be what we want. So this code generates a 1 KHz sine wave in the ISR and use the output from the FFT as its amplitude. Then the Vpp measured by the scope will represent the magnitude of that frequency bin from FFT. In other words, the output is an amplitude modulated 1 KHz signal.

Q5. At what input frequency would you expect to see a maximum Vpp for the output signal? Show your work.

Import starting_point project into Code Composer Studio and add fft.c to it. Make sure you have 452_Support project imported to your workspace as well. Now run the code. Put the input frequency at the frequency you computed for Q5. Move it around a little bit. Move it to exactly twice the frequency you computed in Q5.

G2. Explain to your GSI what it is you are seeing. Specifically address why the three cases (at the computed frequency, near the computed frequency and twice the computed frequency) look the way they do.

Q6. This code is making extensive use of interrupts. Use a reasonable technique to figure out how many data samples arrive before the FFT is done being computed. How many samples

was it? Explain the technique you used.

G3. Change the FFT to be 1024 points and apply the 100dB Chebyshev window you computed for the pre-lab. Show your GSI what you've done and test to see if your output has changed in any interesting ways.

Part2: Fun with graphics

In lab thus far, we have worked with both the FPGA and the processor. But we have not worked with both at the same time. In this section you will play with some code written to allow the processor to use the FPGA to do graphics work. The version we are running is pretty basic: it only allows one color to be written and it only allows the drawing of lines. But that's actually enough to do nearly anything.

The method we have to describe what we want will be simple line drawing. Think of a device which has a single pen that is either in the "up" (not writing) or "down" (writing) position. At any given point in time the pen is in a certain position. We will tell the pen to go to some new position while down (Draw) or up (GoTo). If down a line will be drawn from where we were to where we went.

Download the codes associated with this part as well as the .sof file. Load the .sof file on the FPGA then add the C codes to your Code Composer Studio project and build it. Have your GSI check that you got the correct connection between the C5515 and the FPGA: SPI 1 should be connected to J5 on the extension board. A simple outline of a square should show up on the VGA output.

Modify the program to draw a **20 by 20 hollow square**, a **30 by 30 solid square** and a **hollow triangle** of any size. None should touch each other or be very close to the edge of the screen. Once you have that working, add a **box** which is as large as possible, effectively framing the entire screen. You might need to do an auto-adjust for your screen before you do this.

G4. Show your GSI your art work.

Q7. What are the dimensions of the screen? Where is the location (0, 0) on the screen?

Part 3: Graphing the FFT

Now that we can draw images, the next obvious step is to graph our FFT. This is a bit tricky for a number of reasons. First of all, our graphics routines are using another serial bus (SPI) to talk to the FPGA and this serial bus isn't massively fast. In fact we can't graph the whole FFT output before it is time to start on the next FFT. That solution to that is fairly simple—we'll just ignore the inputs for a while. For some applications (such as just displaying the FFT), that can be fine. For others it's a lot more problematic (when doing speech recognition you can't just ignore parts of the words and hope to get a reasonable output).

The second issue is that we need to now compute the magnitude of the FFT for all the points. It's not hard, but it is a change. The final issue is that we've got two bits of code to combine that aren't really designed to be combined. Getting all the initialization right can be a pain. We're providing a bit of the initialization code to get you past that hurdle.

So what you are to do is compute the magnitude for each FFT bin and graph it. Go back down to 512 frequency bins with just a rectangular window. Don't worry about displaying an axis or anything like that. Disable interrupts while you are computing and plotting the FFT by calling the macro `_disable_interrupts()`.

```
InitSystem();
ConfigPort();
USBSTK5515_init();
AIC_init();
I2S_interrupt_setup();
InitSpi();                          // initialize SPI
_enable_interrupts();
```

We very much suggest that you try some simple things first—you might want to see if you can get the FFT code running at the same time you are drawing a few lines. Or maybe just try to graph a single line. Also, think carefully about scaling the output. You should know how large the screen is at this point; think about the range of values "Output" can take (think carefully about negative values in particular—can they occur?). Take good advantage of the debugger.

Once you have the FFT being displayed, again play with different frequencies. In particular look at a sine wave of 940 Hz and a square wave that has the same frequency. Does the output seem reasonable? Compare it to a ramp of that same frequency.

G5. Demonstrate your FFT display to your GSI. Explain why the 940 Hz square wave and ramp have the outputs they do. In particular explain the magnitude *and* spacing of the various peaks.

Once you've done that, we'd like to more formally write explain this in terms of the Fourier series.

Q8. What did you see with the ramp and square wave? In terms of the Fourier series, what would you expect to see? Does the output meet expectations?

G6. Find the 100dB Chebyshev window for 512 points and apply it to your FFT. Make sure you are using the entire display so. Show your GSI your output and explain any difference with the rectangular window.

3. Post-lab

Q9. When observing the FFT output you should have noticed that certain output bins vary with time. That is, they jump around a bit even with a constant input. Explain why that is. You might find it useful to discuss the possible range of values for X(0) if the input is half the frequency spacing. What impact did your window have on this if any?

Q10. Do some reading about the SPI protocol (the Wikipedia article is pretty good). What are the signals involved? What is their role? What is a "slave" vs. a "master" device? In our set up is the C5515 acting as slave or master? Is the FPGA acting as a slave or master?

---

* To quote from a TI slide show:

"The IVPD and IVPH are 16-bit registers and represent the upper 16-bits of the 24-bit vector table address. The advantage with having two options for the upper bits is that the IVPD register can place the selected interrupt vectors (IV0-15, IV24-31) (where IV0 is reset, IV1 is NMI and so on) in DSP internal memory where the host may

not have access (if you have a host like the ARM9 e.g.). And, the other vectors (IV16-23) can be placed in shared memory where the host can access them and change the target vector address for the ISR real-time. If a host does not exist, the default way of programming this is to make IVPD=IVPH."