Lab 3 Winter 2017 EECS 363 Digital Filtering – Due Jan. 27.

Interrupt Service Routines

Examine the basic loopback program called `main` that was used in Lab 2. The statement `while(1)` just sets up an infinite loop, since 1 is always true. Everything else happens in iterations of this outer `while` loop.

The program does its I/O by the method of "polling". The single statement `while(!(I2S2_IR & RcvR) );` , having a semi-colon at the end, is a complete `while` loop. It says that execution just gets hung up on that statement while it is not true that both left and right input values have been read.

Likewise `while( !(I2S2_IR & XmitR) );` means execution gets hung up there until the codec is ready to transmit left and right output values. After transmitting a number of output values equal to the sample rate, an LED is toggled.

Polling can be considered an inefficient way to do the I/O, since execution, apart from those inner do-nothing `while` loops, is suspended while waiting for input values to arrive. The alternative is the "interrupt service routine" (ISR), in which the subprograms handling the I/O are only executed when I/O is called for. At other times, `main` can be doing something else.

Start this lab as if you were doing Lab 2 all over again. After you get it running you should convert your loopback program into one using an ISR to do I/O. Before `main` add the prototype `interrupt void codec_read_isr(void);` , and before `while(1)` insert

```
IRQ_plug(RCV2_EVENT,&codec_read_isr);
IRQ_enable(RCV2_EVENT);
IRQ_globalEnable();
```

Next, in the file `main`, append the ISR, i.e.

```
/* ------------------------------------------------------------------------ *
 *                                                                          *
 *  End of main.c                                                           *
 *                                                                          *
 * ------------------------------------------------------------------------ */

interrupt void codec_read_isr(void)
//
{
// variable declarations, left_input etc, including "dummy".
// Remove or comment out those variable declarations from
// Lab 2.
//
  if (I2S2_IR & RcvR)
     {
```

```
      // the read/write statements from main go here
      // get rid of the "mono_input"
      // you don't need while( !(I2S2_IR & XmitR) );
      }
  return;
}
```

Execution passes to the ISR when an input appears on the left channel only, but immediately exits the ISR if both inputs are not there. Thus for any given right/left input pair, the ISR is entered twice. On the second entry both inputs are there, so execution of whatever should be done proceeds. There is probably a more elegant way to do this but this strategy works.

As for the transmit (output) part of this, it appears no interrupt is necessary because my experience is that, after an input pair has been received, the codec is ready to transmit.*

In `main` you must of course delete the read/write statements and the various variables associated with them. In addition, put

```
#include "csl_i2s.h"
#include "csl_intc.h"
```

after the statement `#include "stdio.h"`.

It is possible that, if an interrupt occurs while you are blinking the diodes, a register can be corrupted. Fix that by briefly turning off interrupts as follows. After `toggle = 1-toggle;` put `asm(" SSBX INTM");` the space after `"` is important. That is an assembly language statement, made within the C program that shuts off interrupts. After the statement that turns the LED off, put `asm(" RSBX INTM");` to turn interrupts back on.

If an interrupt occurs during the blinking, you will not lose the input sample; you will just process it with a small time delay.

Now you must modify the part of `main` that blinks the diode. That blinking will be considered the "background task" in this exercise, and its importance consists entirely in showing that in principle there can be something else happening while the audio I/O is going on. If your ISR is working, the LED will be blinking so fast that it will appear to be not blinking at all, because the blinking of the LED now has nothing to do with the sample rate. It relates to the chip's processing speed and is so rapid that the LED appears to be always on. Modify your program so that the blinking is at about the same rate as before. Note that the `unsigned short` variable `j` can go no higher that 2^{16}-1, so replace it with `an unsigned long` type. You could use the `unsigned long i` that is already there, and is a useless survivor from the original TI tutorial program.

As a rule, we don't like to use the long types with this processor. TI means it when it says it is a 16-bit processor. Use of the long type requires invoking, behind the scenes, some awkward

library functions. If you want to be a purist and not use the long type you could do the equivalent with nested loops using two `unsigned short` control variables.

For this Lab, give me a demo and also hand in a printout of your C code for `main` and `interrupt void codec_read_isr .`

*It could make one feel better to, within the ISR, poll to transmit. Another way would be an ISR within the ISR to transmit. These two possibilities should be noted but seem unnecessary.

1/16/2017