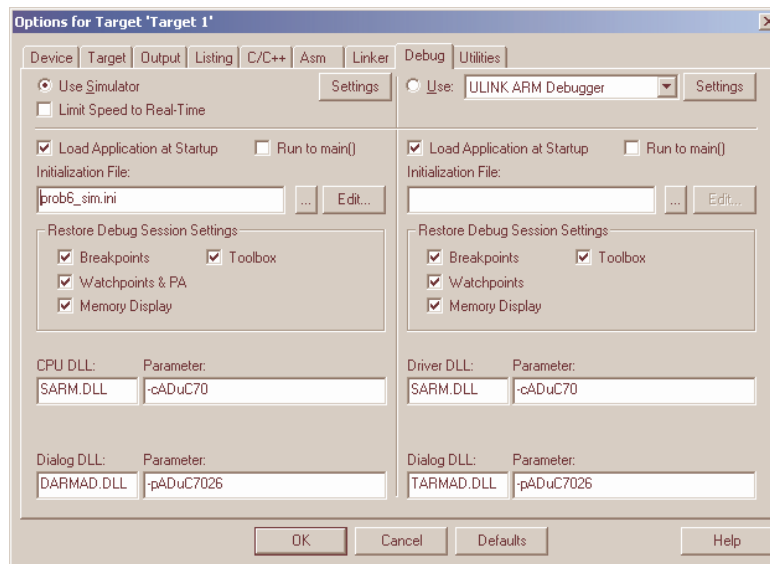


HOMEWORK ASSIGNMENT #6 - SOLUTIONDue Friday, April 24th, 2009**Notes on using simulated inputs in the Keil development environment:**

In 2 problems, you will simulate the signals that would be received from an external device using a simulation file available on the course web page. Copy this file to your Keil project directory. To use this file for simulation during your debug session, select **Project**→**Options for Target**'Target 1' from the Keil menu. Click the **Debug** tab, and enter the simulation file name in the simulator side **Initialization File:** box, as shown below (in this case, the filename is *prob6_sim.ini*). Click the **OK** button to save the change.



After you start your debug session but **before** you run your code, click the **Command** tab in the **Output Window** at the bottom of the screen, type the command *probX_sim()* (where X is the problem number) on the command prompt line, and press **Enter**. You should see the message “**probX_sim loaded...**” in the command window, indicating the simulation function has been loaded. (Remember to reload the function each time you start a debug session! (Note you also need to reload it each time you reset the simulation to time zero)) Now, when you run your problem, the simulated inputs will be applied to the ADuC7026 simulator. If you single-step through your program, the simulation will still work correctly, since the input event times are tied to ADuC7026 clock cycles. This means that you can run the simulation and debug at the same time.

1. (15 points) Multiplexed Displays

An eight (8) digit, multiplexed LED display (shown below) is to be refreshed at a 1kHz rate. The display interrupts the CPU when it requires service. Assume that it is 10 microseconds after the interrupt request before the ISR can turn on the next digit, and that in total, each ISR invocation suspends the main program’s execution for 50 microseconds. Determine:

- a) the rate at which the display must interrupt the CPU.

Each character needs to be updated at the 1kHz rate. There are 8 characters to refresh, so the interrupt rate must be 8kHz.

b) The required one-shot timer delay to ensure that the display refresh rate 1kHz. The one-shot timer sets up the interrupt frequency. We want to interrupt at 8kHz which would imply a $125\mu\text{sec}$ one-shot delay. However, it is 10msec into the ISR before we reset the one shot, so the one shot delay needs to be $115\mu\text{sec}$.

c) the duty cycle of any given digit.

Duty cycle refers to the amount of time the digit is actually lit. It is given as a percentage. In this case we know the digit is lit for $115\mu\text{sec}$ out of a period of 1msec. That means 11.5%

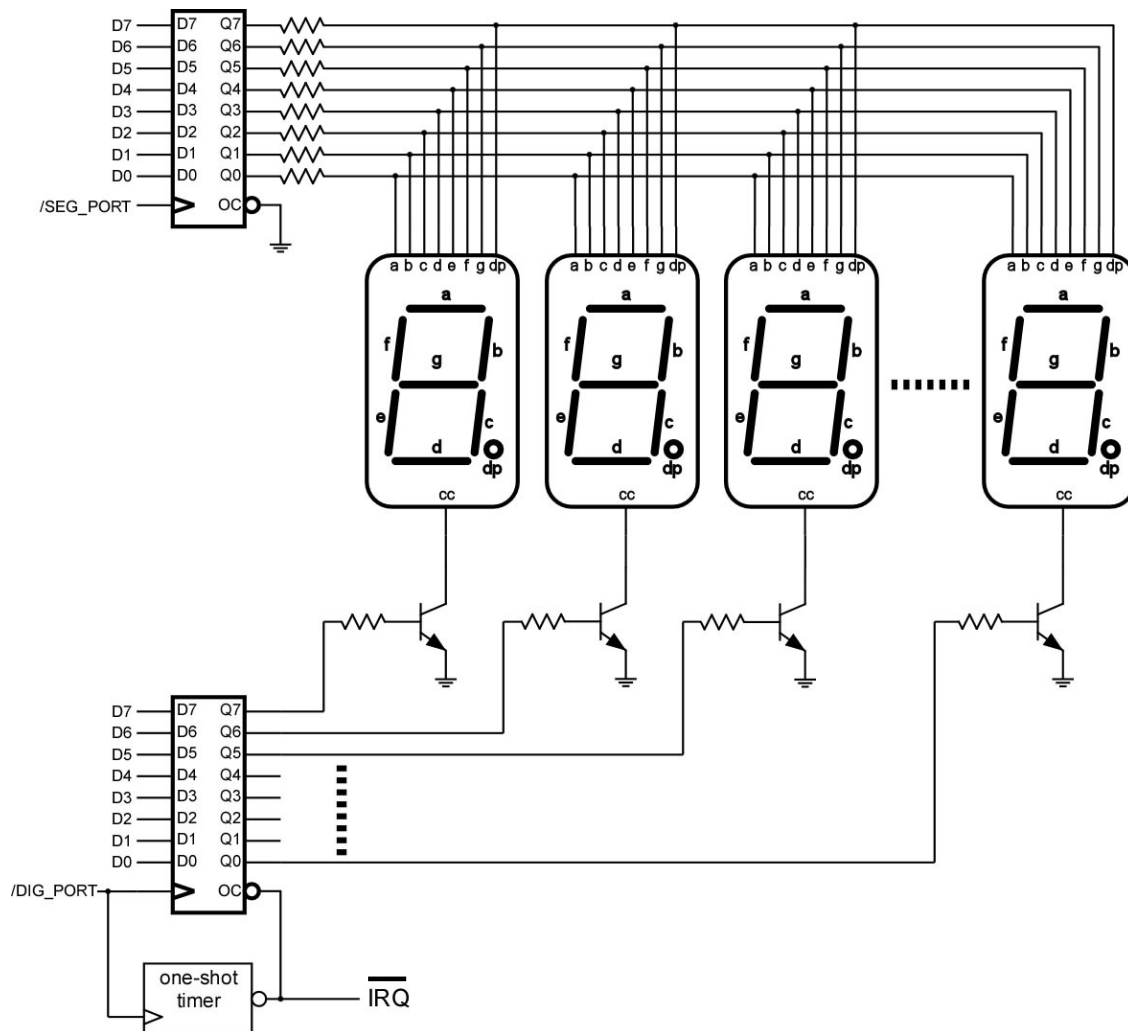
d) what percentage of total CPU time is being used to service the display.

It takes $50\mu\text{sec}$ of total time for each invocation of the ISR. It is invoked every $125\mu\text{sec}$.

Therefore $50/125 \Rightarrow 40\%$ of the processor's time is spent servicing the display.

e) what the actual current would have to be for an illuminated LED segment, assuming that the average LED segment current desired was 3mA.

One chooses the instantaneous current such that the instantaneous current time the duty cycle equals the desired average current. That means the desired average current divided by the duty cycle would have to be the instantaneous. $3\text{mA}/11.5\% \Rightarrow 26\text{mA}$.



2. (20 points) ISR Basics...with dispatcher See solution (zip file) posted on web.

You will be submitting 2 files for this problem. **teamX_exceptions_p2.s** and **teamX_prob2.s**.

Timer 1 and Timer2 can both operate in a real-time mode (hours:minutes:seconds:hundreths) mode. The data sheet is rather unclear about this mode, but the timing is encoded as follows:

TxVAL[6:0] = 1/128 of a second, in range of 0-127. TxVAL[7] is not used.

TxVAL[13:8] = seconds, in range of 0-59. TxVAL[15:14] are not used.

TxVAL[21:16] = minutes, in range of 0-59. TxVAL[23:22] are not used.

TxVAL[28:24] = hours, in range of 0-23. TxVAL[31:29] are not used (*unless in 0-255 hour mode*)

You are to setup each of these timers in this real-time mode. Both should be setup to count down, and reload. Timer1 should be setup to count down from 1.00 seconds to zero, and Timer2 should be setup to count from 1.50 seconds to zero.

Both timers should be configured to cause an interrupt when they underflow.

Now in the **teamX_exceptions_p2.s** file there is a dummy stub for handling interrupts. You will place a ISR dispatcher here. This dispatcher will read the IRQSTA register to determine who interrupted (Timer1 or Timer2).

If Timer1 was the source of the interrupt it should call an ISR called `TIMER1_ISR`. If Timer2 was the source of the interrupt it should call an ISR called `TIMER2_ISR`. These ISR's should **not** be defined in **teamX_exceptions_p2.s**, but should rather be defined in your main source code file (**teamX_prob2.s**). Therefore you will have to use `IMPORT/EXPORT` directives properly to get stuff compiling.

The main code should have setup both Port1 and Port2 to be outputs. `TIMER1_ISR` will simply increment the 8-bit value of port1. `TIMER2_ISR` will simply increment the 8-bit value of port2.

Where does the value currently in port1 and port2 get stored? Do not read the pin state and increment it. Instead you should have `SPACE` allocated in the `SRAM DATA AREA` for 2 bytes that store the current value of port1 and port2.

None of the routines (Dispatcher, `TIMER1_ISR`, or `TIMER2_ISR`) should disturb the contents of the users registers. This means pushes/pops will have to be taking place. **HINT:** You are calling a routine from the dispatcher (probably using a branch and link...so what does that mean?)

Important: Submit Both your **teamX_exceptions_p2.s** and **teamX_prob2.s** files to the dropbox at `Learn@UW`. Also, submit a **paper copy** of both files.

3. (20 points) Switch Debouncing in Software **See Krishna's solution (ZIP file) posted on Web.**

In this problem, you will perform debouncing of a simulated switch input on pin P4.0. Configure P4.0 as an input pin, and P3.0-1 as output pins. Configure Timer2 to use `HCLK` and obtain a 10.0 kHz interrupt rate (or as close as possible). The Timer 2 interrupt should be handled in `IRQ` mode. In the `IRQ` exception handler, you are to perform software debouncing of the switch

input. Remember to clear the timer interrupt. Assume that the switch is connected with a pull-up resistor to 3.3V so that P4.0 will be 0 when the switch is pressed, and its bounce time will be less than 8ms. You are to only interpret the switch as being pressed or released once it is seen that way continuously for 8ms. Perform debouncing of both switch press and switch release events. The ISR cannot corrupt any registers.

Whenever you determine a switch event should be issued, simply set P3.0 high for one Timer 2 period if it is a press event and set P3.1 high for one Timer 2 period if it is a release event. If no event is issued, both pins should be low.

For this problem, add the simulation file *prob3_sim.ini* to the debug configuration, and execute the *prob3_sim()* command in the debugger command window before running the program. You need to run the simulation long enough to see all of the simulated inputs (~2s of CPU time). You may find it very useful for debugging to turn on the logic analyzer function (**View→Logic Analyzer Window**), click **Setup**, and insert the signals **PORT3.0**, **PORT3.1** and **PORT4.0** to see the pin waveforms in a time domain display.

Important: Submit ONLY your program source code files **teamX_exception_p3.s** and **teamX_prob3.s** using your homework team's dropbox in Learn@UW. Also, submit a **paper copy** of both files.

4. (15 points) Dimmable Light using an LED.

- A. We are going to drive an LED with our ADuC. Our LED of choice is a **OVS5WBCR4** (as I have stated previously DigiKey is always a good place to look up data sheets). Our ADuC is going to be operating at 3.3V. However we do have a strong 5V supply available on this system. When operating at maximum brightness we are going to put 150mA through this LED. Design the interface circuitry so that a ADuC pin can control this LED. (Draw the schematic, show any extra parts (including part numbers) of extra devices used. Show calculations of why these parts and component values are appropriate).

First off take a look at what our GPIO pins on the ADuC can sink. They can only drive 1.6mA of current. So there is no way we can just use the GPIO as part of the direct current path of the LED. We will need to use an external MOSFET. I choose to use an NMOS on the low side.

Now to size the resistor. The LED in question has a V_f of around 3.2V. So using using the 3.3V supply is pretty much out of the question. We will use the 5V supply that is available in this system.

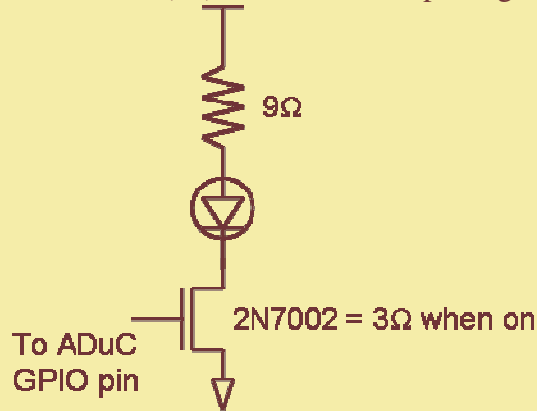
$V=IR$, so $R = V/I$. $R = (5.0-3.2)/150\text{mA}$. So $R = 12 \Omega$.

Now lets look for our MOSFET on DigiKey. We are looking for 5 criteria.

- 1.) Cheap
- 2.) Small footprint
- 3.) Capable of sinking 150mA
- 4.) Low enough $R_{ds(on)}$

5.) Cheap

A 2N7002 in a SOT-23 package is found to meet all these criteria (\$.022) in volume of 10,000. 260mA, $R_{ds(on)} = 3\Omega$, SOT-23 package.



Use PWM to dim with a
Refresh rate > 100Hz

B. How are you going to control the brightness of this dimmable light? It is driven at 150mA for maximum desired brightness, but how will you dim it?

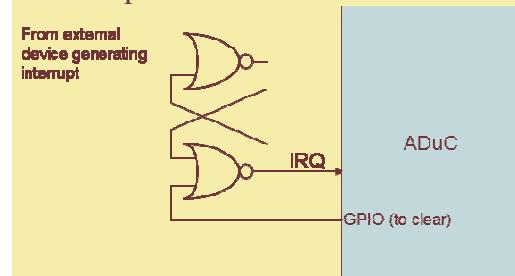
Use PWM. Lower the duty cycle from 100% to dim. Keep refresh rate > 100Hz.

5. (5 points) ADuC7026 External Interrupts

The ADuC7026 has four external interrupt request inputs. These are level-sensitive, active-high interrupt request inputs. Assume that there is an external device that asynchronously signals an interrupt by making its IRQ output signal to change from a 0 to a 1 and then returns the signal to a 0 level 50ns later. Design a circuit to capture (latch) this interrupt even and hold it as a high level on the ADuC external interrupt pin. Keep in mind this captured high level interrupt will have to be cleared by the ISR. Assume you have a GPIO pin configured as an output that will be used to clear this interrupt signal.

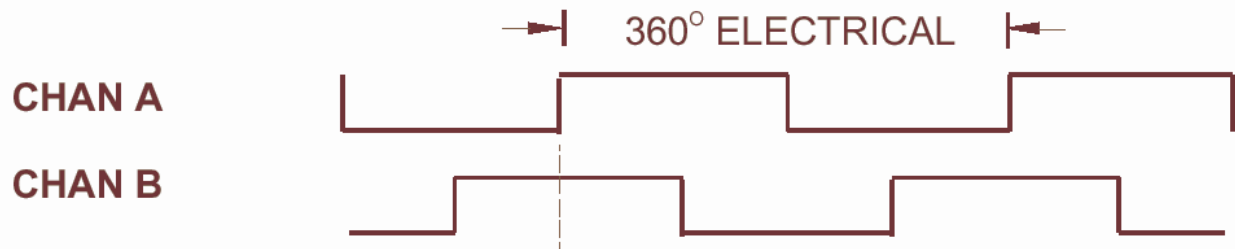
Draw a sketch of the logic you would use. Stick to available 74HCxx logic cells.

Several ways to skin this cat. Some did it with flops with the interrupt source hitting the clock input with VDD on the D input and a clear mechanism from the GPIO pin. This is good. Another possible solution is shown below.



6. (15 points) Rotary Encoders – See Solution (zip file) posted on Web

Assume that you have a system that uses a quadrature rotary encoder for input. The encoder is connected so that the A output is connected to pin P4.4, and the B output is connected to P4.5. The encoder signals behave as shown below. We will use the encoder signals to control the value of a **signed byte** variable *EncoderPos* so that the variable reflects the change in encoder position since program start, so initialize *EncoderPos* to zero. If you detect a CW rotation of the encoder, increment the *EncoderPos* value. If you detect a CCW rotation of the encoder, decrement the *EncoderPos* value.



Chan B leads Chan A for CW shaft rotation, viewed from shaft end.

You will set-up timer 2 to generate 5.0 kHz interrupt rate. The interrupts are to be handled in FIQ mode. In the FIQ exception handler, you will read the state of the encoder signals, and determine if the encoder has moved. Note that each time you check the encoder; you need to compare the current A/B signals to the previous A/B signals. That means that there are 16 possible value sets that could occur. (In some cases, you cannot tell which way the encoder was moved; if this happens, you should not change the *EncoderPos* value.) You should recognize that there are quick, efficient ways to consider the 16 possibilities, and there are also very cumbersome, inefficient ways to do it as well – you should be trying for the former, not the latter. You should start your solution with the Keil sample project that is posted on the course website. All of your code should be placed in **exception.s** and **main.s**.

For this problem, add the simulation file *prob6_sim.ini* to the debug configuration, and execute the *prob6_sim()* command in the debugger command window before running the program. Now, run your program for approximately 250ms of ADuC7026 clock time. When you halt it, if your program is correct you should find that *EncoderPos* is now equal to +3. (While running, it should hit extremes of -4 and +5.) If you write the value out to a port and add that port to the logic analyzer, you can see what it did over time.

Important: Submit ONLY your program source code files **teamX_p6_exception.s** and **teamX_p6_main.s** using the homework #6 dropbox in Learn@UW. Also, submit a **paper copy** of **teamX_p6_exception.s** and **teamX_p6_main.s** with the rest of the assignment.

7. (10 points) Quiz Question

Design one original quiz question for any material covered in Module 6. This must test one of the module objectives in a specific problem. Explicitly state which particular objective you are attempting to test. Provide a complete, detailed solution to your question.