



COMPUTER ENGINEERING PROGRAM

California Polytechnic State University

©Copyright: 2007 by Bryan Mealy



CPE 269 Experiment 10



BASIC INTERRUPT HANDLING AND POLLING TECHNIQUES

Objectives:

- To re-emphasize the programming efficiency and productivity gain of code re-use.
- To learn two basic approaches to task oriented-programs.
- To learn the concepts behind implementing an interrupt-driven assembly language program.
- To learn the bit-manipulations required for binary-to-BCD conversion.
- To learn the basic concepts of a look-up-table (LUT)
- To learn the basic operation of the scratch RAM

Somewhat Meaningful Comments:

If you're like most humans, you occasionally need to go to the dentist. It would be a really strange world if the dentist would call you every five minutes and ask you if your teeth needed looking at. Naturally, a better (more efficient? less annoying?) approach would be that when you feel you need a trip to the dentist, you would simply call schedule an appointment. Requesting some type of service is the general approach humans take in most facets of their lives (unless you work in the sales where you're required to ask others if they want service).

Not surprisingly, a situation similar to that listed above exists for microcontroller applications. Programs you write generally do something, *i.e.*, they execute some finite number of relatively useful tasks. The two approaches to implementing tasks are analogous to the example above: you either constantly check to see if tasks need doing and perform them if they do, or you perform those tasks only when they ask you to perform them (generally, they'll only ask when they need doing). The act of constantly asking if a task needs doing (a rough definition) is referred to as *polling* in microcontroller lingo. Implementing these tasks only when they request to be done requires the use of *interrupts*.

This experiment introduces some of the theory and concepts behind writing assembly language programs that utilize interrupts. Because of the simple nature of the interrupt handling mechanism in PicoBlaze and the fact that an in-depth study of interrupt-driven systems is beyond the scope of this experiment, the explanation and approach taken in this experiment touches upon only the basics of interrupt handling.

The concept of *polling* is nothing new to you since this is the approach your program from Experiment 8 took to implement the required tasks. Remember that the program in Experiment 8 primarily consisted of an endless loop that implemented a task over and over. This relatively simple approach to implementing a task does have a potential drawback: it's inefficient to do something that does not necessarily need to be done. In the case of Experiment 8, the output values are registered so you don't need to continually read inputs and write to outputs. Although driving the 7-segment displays was a task that truly needed to be done constantly (the retinal persistence thing), the technique used to monitor, or *poll*, the input switches was inefficient. The approach taken was to read the status of the switches whether they changed or not. A more efficient approach would have been to design a system that only reads the status of the slide switches when a change in switch status was detected. Although hardware limitations constrained Experiment 8 to use polling, this experiment uses the interrupt capability of the PicoBlaze to handle a simple task. As you will see, this is a more efficient method (in terms of clock cycles) of handling tasks.

The term *interrupt* comes from the fact the normal operation of the microcontroller is briefly *interrupted* to handle some other task. Once this task is handled, the microcontroller returns to its normally scheduled task. Microcontrollers generally use three types of interrupts: internal interrupts, external interrupts, and software interrupts. The PicoBlaze is only capable of handling external interrupts and is therefore the type used in this experiment. Unfortunately, there is no single method used by all microcontrollers to handle interrupts. You will soon discover that one of the first things you must do when working with a new microcontroller is to figure out the type and number of interrupts the microcontroller handles and how exactly it handles them.

The mechanism used to handle interrupts in the PicoBlaze is referred to as a *vectored interrupt*. Once an interrupt is received, program control is transferred to a pre-determined address in instruction memory after execution of the current instruction completed. This address, known as a *vector address*, generally contains a JUMP instruction. The instruction JUMPed to generally contains the starting address of the *interrupt service routine (ISR)*. The code in the ISR implements some task to appropriately *handle* the interrupt. When the ISR is complete, program control is returned to the instruction after the instruction that was being implemented when the interrupts was detected.

The ISR is essentially a subroutine with a special instruction. When an interrupt is received by the PicoBlaze microcontroller, part of the automatic response is to prevent further interrupts from being acknowledged by the microcontroller. Preventing further interrupts from being acted upon by the microcontroller is referred to as *masking*. Remember, preventing interrupts from being acted on by the microcontroller does not prevent the interrupts from happening. You can use the ENABLE INTERRUPT instruction to at any point in your code, but you must use it sometime if your system needs to act upon future interrupts. This is another one of the things you must be familiar with before dealing with the interrupt mechanism of any microcontroller. To properly implement this experiment, you must know how to deal with these mechanisms as briefly described here but described in more detail in the PicoBlaze application note. In particular, read over the sections dealing with interrupts and how they are handled by PicoBlaze.

Low Level Details on Interrupt Handling

The low level mechanics of how interrupts are handled by PicoBlaze is similar to that of standard subroutines. These mechanics refer to the program flow associated with interrupts. When PicoBlaze senses that the signal attached to the interrupt input has been asserted, execution of the current instruction is completed before processing of the interrupt begins. The first step in the processing of the interrupt is to place the vector address in the program counter. This forces the location of the next instruction to be executed to be at the vector address of FF. At the same time, the address of the next instruction that would have been executed had there not been an interrupt is placed onto the stack and the carry and zero flags are saved. Processing the ISR begins and continues until a RETURNI instruction is encountered. This causes the address of the next instruction that is executed to be taken off the stack and placed into the program counter. In this case, that instruction is the one that would have executed had not the interrupt arrived. Program control is transferred to this instruction and normal program execution of the task code resumes. These steps are summarized below in a rough order of occurrence. Some of the bulleted items happen simultaneously since some of the listed actions are initiated by the PicoBlaze hardware.

- Asserted signal is detected on interrupt input.
- The current instruction completes execution.
- Address of next instruction (not in ISR) is stored in the stack.
- Program counter is set to FF.
- Present state of the condition flags (zero and carry flags) is saved.
- The instruction at location FF is executed (this should be a jump instruction that directs program control to the ISR).
- Execution of the ISR begins.
- Execution of the ISR completes.
- Return address is removed from stack.
- Condition flags are restored.

- Execution resumes at the instruction following the one that was being executed when the interrupt was received.

When the signal on the interrupt input is asserted, the interrupt processing listed above begins. The interrupt signal must be a pulse of a particular length in order for processing to continue. If the interrupt pulse is too short, PicoBlaze may not recognize it and it will be missed. As reported by the PicoBlaze user's manual, the length of the pulse should be approximately two clock cycles to ensure proper acknowledgement. See the note in the post-mortem section of this experiment for further details

Look-Up-Tables (LUTs) and the Scratch RAM

One of the several advantages that PicoBlaze3 has over PicoBlaze is the 64x8 scratch RAM. The scratch RAM is nothing more than a storage area for 8-bit data. The scratch RAM is accessed using the FETCH instruction (read) and the STORE instruction (write). All scratch RAM data accesses are done via the normal 16 PicoBlaze registers.

Once typical time use of scratch RAM is to set up *look-up-tables*, or LUTs. As you will see, LUTs provide quick access to *indexed* data. In terms of this experiment and the previous experiment, you were forced to manually decode the BCD values. By using a LUT, you don't need to officially "decode" the BCD values any more, you simply used the BCD value as an index into the LUT. The approach to take in any program that uses a LUT is to initialize the LUT somewhere at the beginning of the program, before the program enters the main loop. For this experiment, you'll need to input ten values into the LUT representing the 10 sets of data that tell the 7-segment displays how to represent the ten decimal numbers.

For example, the segment values for driving a "6" to the display is stored in scratch RAM location "6" (you would do this for all ten decimal values):

```
LOAD    s0, 41    ; load 7-segment value for 6 into reg s0
STORE   s0, 06
```

Then, when you have a BCD value and you need the corresponding 7-segment data value, you use the indirect FETCH instruction (assume BCD value is in register s4):

```
FETCH   s0, (s4)  ; get 7-segment value for value in s4
```

In essence, you have "looked up" the value as opposed to decoding the value as you had done before. From here, you would send this value to the 7-segment display ports. The LUT is much more efficient than decoding the values. The only slight drawback is that you must use 10 of the 64 scratch RAM locations.

Programming Assignment:

First, convert or rewrite your decode routines from Experiment 9 to use a LUT. It will massively simplify your code for this experiment. Then, write an assembly language program that will:

- Count the number of interrupts received by the system (a single interrupt is generated by pushing button-3 on the Nexys board).
 - Display the number of interrupts (in decimal) received by the system on the two right-most 7-segment displays.
 - The maximum number of interrupts is 99. After this count value is reached, the display rolls over zero and the interrupt counting continues.

- Monitor the status of slide switch-0 to indicate whether the interrupt should be counted or not.
 - If the switch is in the ON state, an interrupt should increment the current count value and update the 7-segment displays.
 - If the switch is OFF, the current count value should not change when button-3 is pressed and the currently displayed count is unaffected.

Hint: Re-use as much code from the previous experiment as possible. Also, place all major functions in subroutines so that the main body of your code should primarily be comprised of a series of subroutine calls.

NOTE: The underlying hardware has attached button-3 to the interrupt input on PicoBlaze. You must use this button to generate the interrupts unless you feel a need to change the PicoBlaze hardware configuration.

NOTE: You must be careful to avoid the *ghosting* effect. The code you write in this experiment should drive the two 7-segment displays in a clear manner. If some of the segments of the display appear dim, then those segments are being driven when they should not be. To avoid the ghosting effect, make sure that your code enables the 7-segment displays only then the correct data has been written to the display segments. In other words, you must turn off the displays, write the correct data, and then re-enable the displays. Also be aware of the segment data and the enabled digit when the delay routine is called.

Deliverables:

1. Demonstrate your working circuit to the lab instructor or Teaching Assistant.
2. Provide a listing of your assembly source code with your lab write up.
3. Answers to the questions below.

Questions:

1. The PicoBlaze hardware only has one interrupt input. This was OK for the above program because there was only one device that was generating interrupts. Describe how you would alter the system so that you could have more than one device interrupting the PicoBlaze. Comment on how you would both *acknowledge* and *handle* an interrupt from one of the connected devices. Your solution should not include modifying the current PicoBlaze device; only modifications external to PicoBlaze are permitted in your solution.
2. In PicoBlaze, the interrupt is disabled by default upon power-up. What problems could arise if the default state on power-up for the interrupt was enabled? Fully explain your answer.
3. In PicoBlaze, future interrupts are automatically masked by the hardware once an interrupt is received. If this were not the case, couldn't you simply issue a DISABLE INTERRUPT as the first instruction in the interrupt service routine and achieve the same effect? Fully explain your answer.
4. If you were to design your program such that a considerable amount of processing was done in the interrupt service routine, what effect would this have on the overall functionality of your program? Answer this question in the context of the program you wrote for this experiment.
5. I decided that I did not need to call the delay subroutine that was suggested for multiplexing the 7-segment displays. The resulting output looked OK (there was no ghosting) but... the displayed numbers were really dim as compared to when I called the delay subroutines at the appropriate times. Fully explain why the numbers were dimmer without the inserted delay subroutine.

6. What are the differences between a RETURN instruction and a RETURNI ENABLE instruction? Your answer should encompass the differences between how the PicoBlaze hardware responds to these instructions. Fully explain your answer.

Procedure Post-Mortem

The interrupt used in this experiment is a hardware interrupt. This means that there is a signal in the circuit that is physically attached to the interrupt pin on the PicoBlaze. When this signal is asserted (goes high), PicoBlaze enters into a preset sequence of events that includes the disabling of further interrupts. Future interrupts will only be handled if the interrupts are re-enabled under software control (ENABLE INTERRUPT). For the current PicoBlaze hardware setup, button-3 has been arbitrarily connected to the interrupt pin on PicoBlaze. There is no magic here, just a basic manipulation of the Implementation Constraints File.

PicoBlaze is running at 50MHz which is relatively fast compared to how quickly you can press and release button-3. Chances are good that when you press the button, the button will remain pressed long enough for your PicoBlaze assembly code to properly handle the interrupt. This presents the situation that when the interrupt handler is exited and the interrupt mechanism is re-enabled, the same interrupt from your last button press will still be there because you have not lifted your finger. In this situation, the hardware would notice that the interrupt line is asserted and enter back into the interrupt service routine. In effect, the interrupt would be serviced multiple times which would be erroneous.

To avoid the situation listed above, button-3 is connected to a *mono-stable multivibrator*, which is also known as a *one-shot*. The output of the one-shot is connected to the PicoBlaze interrupt pin. The one-shot is configured so that when the input signal changes from zero to one, the output also changes from zero to one. The pulse-width of the output is preset and acts independently of the input to the one-shot. In this way, the input signal can remain high for an indefinite period of time while the output signal remains high only momentarily before it returns to zero. The high state is referred to as the unstable state while the low state is the stable state. There is only one stable state, hence, mono-stability is attained. The one-shot in the VHDL code associated with PicoBlaze is implemented using a FSM. This FSM is synchronized with the system clock and the VHDL code is written such that the pulse-width of the one-shot is long enough so that the interrupt is noticed by the PicoBlaze hardware, but short enough so that the pulse is gone before enough time passes to execute an ENABLE INTERRUPT instruction.