- Priorities and subpriorities are associated with interrupt vectors, and therefore ISRs, not IRQs. The priority of a vector is defined in an SFR IPCy, y = 0 to 15. In the definition of the associated ISR, the same priority `n` should be specified using `IPLnSOFT` or `IPLnSRS`. `SOFT` indicates that software context save and restore is performed, while `SRS` means that the shadow register set is used instead, reducing ISR entry and exit time. The PIC32 on the NU32 is configured by its device configuration registers to make the SRS available only at priority level 6, so the SRS can only be used with `IPL6SRS`.
- When an interrupt is generated, it is serviced immediately if its priority is higher than the current priority. Otherwise it waits until the current ISR is finished.
- In addition to configuring the CPU to accept interrupts, enabling specific interrupts, and setting their priority, the specific peripherals (such as counter/timers, UARTs, change notification pins, etc.) must be configured to generate interrupt requests on the appropriate events. These configurations are left for the chapters covering those peripherals.
- The seven steps to use an interrupt, after putting the CPU in multi-vector mode, are: (1) write the ISR; (2) disable interrupts; (3) configure a device or peripheral to generate interrupts; (4) set the ISR priority and subpriority; (5) clear the interrupt flag; (6) enable the IRQ; and (7) enable interrupts at the CPU.
- If a variable is shared with an ISR, it is a good idea to (1) define that variable with the type qualifier `volatile` (also use `static` unless you have good reason not to) and (2) turn off interrupts before reading or writing it if there is a danger the process could be interrupted. If interrupts are disabled, they should be disabled for as short a period as possible.

6.6 Exercises

1. Interrupts can be used to implement a fixed frequency control loop (e.g., 1 kHz). Another method for executing code at a fixed frequency is *polling*: you can keep checking the core timer, and when some number of ticks has passed, execute the control routine. Polling can also be used to check for changes on input pins and other events. Give pros and cons (if any) of using interrupts vs. polling.
2. You are watching TV. Give an analogy to an IRQ and ISR for your mental attention in this situation. Also give an analogy to polling.
3. What is the relationship between an interrupt vector and an ISR? What is the maximum number of ISRs that the PIC32 can handle?
4. (a) What happens if an IRQ is generated for an ISR at priority level 4, subpriority level 2 while the CPU is in normal execution (not executing an ISR)? (b) What happens if that IRQ is generated while the CPU is executing a priority level 2, subpriority level 3 ISR? (c) What happens if that IRQ is generated while the CPU is executing a priority level 4, subpriority level 0 ISR? (d) What happens if that IRQ is generated while the CPU is executing a priority level 6, subpriority level 0 ISR?

5. An interrupt asks the CPU to stop what it's doing, attend to something else, and then return to what it was doing. When the CPU is asked to stop what it's doing, it needs to remember "context" of what it was working on, i.e., the values currently stored in the CPU registers. (a) Assuming no shadow register set, what is the first thing the CPU must do before executing the ISR and the last thing it must do upon completing the ISR? (b) How does using the shadow register set change the situation?
6. What is the peripheral and interrupt vector number associated with IRQ 35? What are the SFRs and bit numbers controlling its interrupt enable, interrupt flag status, and priority and subpriority? Does IRQ 35 share the interrupt vector with any other IRQ?
7. What peripherals and IRQs are associated with interrupt vector 24? What are the SFRs and bit numbers controlling the priority and subpriority of the vector and the interrupt enable and flag status of the associated IRQs?
8. For the problems below, use only the SFRs IECx, IFSx, IPCy, and INTCON, and their CLR, SET, and INV registers (do not use other registers, nor the bit fields as in IFS0bits.INT0IF). Give valid C bit manipulation commands to perform the operations without changing any uninvolved bits. Also indicate, in English, what you are trying to do, in case you have the right idea but wrong C statements. Do not use any constants defined in Microchip XC32 files; just use numbers.
   a. Enable the Timer2 interrupt, set its flag status to 0, and set its vector's priority and subpriority to 5 and 2, respectively.
   b. Enable the Real-Time Clock and Calendar interrupt, set its flag status to 0, and set its vector's priority and subpriority to 6 and 1, respectively.
   c. Enable the UART4 receiver interrupt, set its flag status to 0, and set its vector's priority and subpriority to 7 and 3, respectively.
   d. Enable the INT2 external input interrupt, set its flag status to 0, set its vector's priority and subpriority to 3 and 2, and configure it to trigger on a rising edge.
9. Edit Code Sample 6.3 so that each line correctly uses the "bits" forms of the SFRs. In other words, the left-hand sides of the statements should use a form similar to that used in step 5, except using INTCONbits, IPC0bits, and IEC0bits.
10. Consulting the `p32mx795f512h.h` file, give the names of the constants, and the numerical values, associated with the following IRQs: (a) Input Capture 5. (b) SPI3 receive done. (c) USB interrupt.
11. Consulting the `p32mx795f512h.h` file, give the names of the constants, and the numerical values, associated with the following interrupt vectors: (a) Input Capture 5. (b) SPI3 receive done. (c) USB interrupt.
12. True or false? When the PIC32 is in single vector interrupt mode, only one IRQ can trigger an ISR. Explain your answer.
13. Give the numerical value of the SFR INTCON, in hexadecimal, when it is configured for single vector mode using the shadow register set; and external interrupt input INT3

triggers on a rising edge while the rest of the external inputs trigger on a falling edge. The Interrupt Proximity Timer bits are left as the default.

14. So far we have only seen interrupts generated by the core timer and the external interrupt inputs, because we first have to learn something about the other peripherals to complete Step 3 of the seven-step interrupt setup procedure. Let us jump ahead and see how the Change Notification peripheral could be configured in Step 3. Consulting the Reference Manual chapter on I/O Ports, name the SFR and bit number that has to be manipulated to enable Change Notification pins to generate interrupts.
15. Build `INT_timing.c` and open its disassembly file `out.dis` with a text editor. Starting at the top of the file, you see the startup code inserted by `crt0.o`. Continuing down, you see the "bootstrap exception" section `.bev_excpt`, which handles any exceptions that might occur while executing boot code; the "general exception" section `.app_excpt`, which handles any serious errors the CPU encounters (such as attempting to access an invalid memory address) (Table 6.1); and finally the interrupt vector sections, labeled `.vector_x`, where `x` can take values from 0 to 51 (12 of the possible 64 vectors are not used by the PIC32MX). Each of these exception vectors simply jumps to another address. (Note that `j`, `jal`, and `jr` are all jump statements in assembly. Jumps are not executed immediately; the next assembly statement, in the *jump delay slot*, executes before the jump completes. The jump `j` jumps to the address specified. `jal` jumps to the address specified, usually corresponding to a function, and stores in a CPU register `ra` a return address two instructions [eight bytes] later. `jr` jumps to an address stored in a register, often `ra` to return from a function.)
    a. What addresses do the `.vector_x` sections jump to? What is installed at these addresses?
    b. Find the `Ext0ISR` and `Ext1ISR` functions. How many assembly commands are before the first `_CP0_GET_COUNT()` command in each function? How many assembly commands are after the last `_CP0_GET_COUNT()` command in each function? What is the purpose of the commands that account for the majority of the difference in the number of commands? (Note that `sw`, short for "store word," copies a 32-bit CPU register to RAM, and `lw`, short for "load word," copies a 32-bit word from RAM to a CPU register.) Explain why the two functions are different even though their C code is essentially identical.
16. Modify Code Sample 6.2 so the USER button is debounced. How can you change the ISR so the LEDs do not flash if the falling edge comes at the beginning of a very brief, spurious down pulse? Verify that your solution works. (Hint: Any real button press should last much more than 10 ms, while the mechanical bouncing period of any decent switch should be much less than 10 ms. See also Chapter B.2.1 for a hardware solution to debouncing.)
17. Using your solution for debouncing the USER button (Exercise 16), write a stopwatch program using an ISR based on INT2. Connect a wire from the USER button pin to the INT2 pin so you can use the USER button as your timing button. Using the NU32

library, your program should send the following message to the user's screen: `Press the USER button to start the timer.` When the USER button has been pressed, it should send the following message: `Press the USER button again to stop the timer.` When the user presses the button again, it should send a message such as `12.505 seconds elapsed.` The ISR should either (1) start the core timer at 0 counts or (2) read the current timer count, depending on whether the program is in the "waiting to begin timing" state or the "timing state." Use priority level 6 and the shadow register set. Verify that the timing is accurate. The stopwatch only has to be accurate for periods of less than the core timer's rollover time.

You could also try using polling in your `main` function to write out the current elapsed time (when the program is in the "timing state") to the user's screen every second so the user can see the running time.

18. Write a program identical to the one in Exercise 17, but using a 16×2 LCD screen for output instead of the host computer's display.
19. Write a program that interrupts at a frequency defined interactively by the user. The `main` function is an infinite loop that uses the NU32 library to ask the user to specify the integer variable `InterruptPeriod`. If the user enters a number greater than an appropriate minimum and less than an appropriate maximum, this becomes the number of core clock ticks between core timer interrupts. The ISR simply toggles the LEDs, so the `InterruptPeriod` is visible. Set the vector priority to 3 and subpriority to 0.
20. (a) Write a program that has two ISRs, one for the core timer and one for the debounced input INT2. The core timer interrupts every 4 s, and the ISR simply turns on LED1 for 2 s, turns it off, and exits. The INT2 interrupt turns LED2 on and keeps it on until the user releases the button. Choose interrupt priority level 1 for the core timer and 5 for INT2. Run the program, experiment with button presses, and see if it agrees with what you expect. (b) Modify the program so the two priority levels are switched. Run the program, experiment with button presses, and see if it agrees with what you expect.
21. A CPU run-time error, such as attempting to access an invalid memory address, generates a general exception. As with an interrupt, program execution jumps to a new function, in this case called `_gen_exception`. In turn, this function calls the function `_general_exception_context` which calls `_general_exception_handler`. You have the option to use the Microchip default general exception handler, or you can write your own, as in Sample Code 5.2 `readVA.c` in Chapter 5, the only sample code in this book that defines a general exception handler. Looking at the disassembly file for any program that uses the Microchip default general exception handler, what does the program do after the software debug breakpoint (`sdbbp`)?

Further Reading

PIC32 family reference manual. Section 08: Interrupts. (2013). Microchip Technology Inc.