CHAPTER 6

Interrupts

Interrupts allow the PIC32 to respond to important events, even when performing other tasks. For example, perhaps the PIC32 is in the midst of a time-consuming calculation when the user presses a button. If software waited for the calculation to complete before checking the button state it could introduce a delay or even miss the button press altogether. To avoid this fate, we can have the button press generate an interrupt, or *interrupt request* (IRQ). When an IRQ occurs, the CPU pauses its current computation and jumps to a special routine called an *interrupt service routine* (ISR). Once the ISR has completed, the CPU returns to its original task. Interrupts appear frequently in real-time embedded control systems, and can arise from many different events. This chapter describes how the PIC32 handles interrupts and how you can implement your own ISRs.

6.1 Overview

Interrupts can be generated by the processor, peripherals, and external inputs. Example events include

- a digital input changing its value,
- information arriving on a communication port,
- the completion of a task that a peripheral was executing,
- the elapsing of a specified amount of time.

For example, to guarantee performance in real-time control applications, sensors must be read and new control signals calculated at a known fixed rate. For a robot arm, a common control loop frequency is 1 kHz. So we could configure one of the PIC32's timers to roll over every 80,000 ticks (or 1 ms at 12.5 ns per tick). This rollover event generates an interrupt that calls the feedback control ISR, which reads sensors and produces output. In this case, we would have to ensure that the control ISR always executes in less than 1 ms (you could measure the time using the core timer).

Pretend that the PIC32 is controlling a robot arm: a control loop running in a timer ISR holds the arm at a specified position. When the PIC32 receives data over the UART, a communication interrupt triggers another ISR, which parses the data and sets a new desired

Embedded Computing and Mechatronics with the PIC32 Microcontroller. http://dx.doi.org/10.1016/B978-0-12-420165-1.00006-8

angle for the robot arm. What happens if the PIC32 is executing the control ISR when a communication interrupt occurs? Alternatively, what happens if the PIC32 is executing the communication ISR and a control interrupt occurs? Each interrupt has a configurable *priority* that we can use to decide which ISR receives precedence. If a high priority interrupt occurs while a low priority ISR is executing, the CPU will jump to the high priority ISR, complete it, and then return to finish the low priority ISR. If a low priority interrupt occurs while a high priority ISR is executing, the low priority ISR remains pending until the high priority ISR finishes executing. When the high priority ISR finishes, the CPU jumps to the low priority ISR. Mainline code (i.e., any code that is not in an ISR) has the lowest priority and will usually be preempted by any interrupt.[1]

So, for our robot arm example, what should the relative priorities of the interrupts be? Assuming that communication is slow and lacks precise timing requirements, we should give the control loop (timer) ISR higher priority than the communication (UART) ISR. This scheme would prevent the control loop ISR from being preempted, helping to ensure the stability of the robot arm. We would then have to ensure that the control ISR executes fast enough to allow time for communication and other processes.

Every time an interrupt is generated, the CPU must save the contents of the internal CPU registers, called the "context," to the stack (data RAM). It then uses the registers while running the ISR. After the ISR completes, it copies the context from RAM back to its registers, restoring the previous CPU state and allowing it to continue where it left off before the interrupt. The copying of register data back and forth between the registers and RAM is called "context save and restore."

6.2 Details

The address of an ISR in virtual memory is determined by the *interrupt vector* associated with the IRQ. The CPU of the PIC32MX supports up to 64 unique interrupt vectors (and therefore 64 ISRs). For `timing.c` in Chapter 5.3, the virtual addresses of the interrupt vectors can be seen in this edited exception memory listing from the map file (an interrupt is also known as an "exception"):

```
.vector_0              0x9d000200            0x8            8  Interrupt Vector 0
.vector_1              0x9d000220            0x8            8  Interrupt Vector 1

  [[[ ... snipping long list of vectors ...]]]

.vector_51             0x9d000860            0x8            8  Interrupt Vector 51
```

[1] Interrupts can have the same priority as mainline code, in which case they will never execute, even when enabled.

If an ISR had been written for the core timer (interrupt vector 0), the code at 0x9D000200 would contain a jump to the location in program memory that actually holds the ISR.

Although the PIC32MX can have only 64 interrupt vectors, it has up to 96 events (or IRQs) that generate an interrupt. Therefore, some of the IRQs share the same interrupt vector and ISR.

Before interrupts can be used, the CPU has to be enabled to process them in either "single vector mode" or "multi-vector mode." In single vector mode, all interrupts jump to the same ISR. This is the default setting on reset of the PIC32. In multi-vector mode, different interrupt vectors are used for different IRQs. We use multi-vector mode, which is set by the bootloader (and `NU32_Startup()`).

With interrupts enabled, the CPU jumps to an ISR when three conditions are satisfied: (1) the specific IRQ has been enabled by setting a bit to 1 in the SFR IECx (one of three Interrupt Enable Control SFRs, with x equal to 0, 1, or 2); (2) an event causes a 1 to be written to the corresponding bit of the SFR IFSx (Interrupt Flag Status); and (3) the priority of the interrupt vector, as represented in the SFR IPCy (one of 16 Interrupt Priority Control SFRs, $y = 0$ to 15), is greater than the current priority of the CPU. If the first two conditions are satisfied, but not the third, the interrupt waits until the CPU's current priority drops.

The "x" in the IECx and IFSx SFRs above can be 0, 1, or 2, corresponding to (3 SFRs) × (32 bits/SFR) = 96 interrupt sources. The "y" in IPCy takes values 0-15, and each of the IPCy registers contains the priority level for four different interrupt vectors, i.e., (16 SFRs) × (four vectors per register) = 64 interrupt vectors. The priority level for each of the 64 vectors is represented by five bits: three indicating the priority (taking values 0 to 7, or 0b000 to 0b111; an interrupt with priority of 0 is effectively disabled) and two bits indicating the subpriority (taking values 0 to 3). Thus each IPCy has 20 relevant bits—five for each of the four interrupt vectors—and 12 unused bits. For easier access from your code, you can access these SFRs using the structures `IECxbits`, `IFSxbits`, and `IPCybits`.

The list of interrupt sources (IRQs) and their corresponding bit locations in the IECx and IFSx SFRs, as well as the bit locations in IPCy of their corresponding interrupt vectors, are given in Table 6.1, reproduced from the Interrupts section of the Data Sheet. Consulting Table 6.1, we see that the change notification's (CN) interrupt has $x = 1$ (for the IRQ) and $y = 6$ (for the vector), so information about this interrupt is stored in IFS1, IEC1, and IPC6. Specifically, IEC1⟨0⟩ is its interrupt enable bit, IFS1⟨0⟩ is its interrupt flag status bit, IPC6⟨20:18⟩ are the three priority bits for its interrupt vector, and IPC6⟨17:16⟩ are the two subpriority bits.

As mentioned earlier, some IRQs share the same vector. For example, IRQs 26, 27, and 28, each corresponding to UART1 events, all share vector number 24. Priorities and subpriorities are associated with interrupt vectors, not IRQs.

Table 6.1: Interrupt IRQ, vector, and bit location

Interrupt Source[a]	IRQ	Vector	Interrupt Bit Location			
	Number	Number	Flag	Enable	Priority	Sub-Priority
Highest Natural Order Priority						
CT—Core Timer Interrupt	0	0	IFS0<0>	IEC0<0>	IPC0<4:2>	IPC0<1:0>
CS0—Core Software Interrupt 0	1	1	IFS0<1>	IEC0<1>	IPC0<12:10>	IPC0<9:8>
CS1—Core Software Interrupt 1	2	2	IFS0<2>	IEC0<2>	IPC0<20:18>	IPC0<17:16>
INTO—External Interrupt 0	3	3	IFS0<3>	IEC0<3>	IPC0<28:26>	IPC0<25:24>
T1—Timer1	4	4	IFS0<4>	IEC0<4>	IPC1<4:2>	IPC1<1:0>
IC1—Input Capture 1	5	5	IFS0<5>	IEC0<5>	IPC1<12:10>	IPC1<9:8>
OC1—Output Compare 1	6	6	IFS0<6>	IEC0<6>	IPC1<20:18>	IPC1<17:16>
INT1—External Interrupt 1	7	7	IFS0<7>	IEC0<7>	IPC1<28:26>	IPC1<25:24>
T2—Timer2	8	8	IFS0<8>	IEC0<8>	IPC2<4:2>	IPC2<1:0>
IC2—Input Capture 2	9	9	IFS0<9>	IEC0<9>	IPC2<12:10>	IPC2<9:8>
OC2—Output Compare 2	10	10	IFS0<10>	IEC0<10>	IPC2<20:18>	IPC2<17:16>
INT2—External Interrupt 2	11	11	IFS0<11>	IEC0<11>	IPC2<28:26>	IPC2<25:24>
T3—Timer3	12	12	IFS0<12>	IEC0<12>	IPC3<4:2>	IPC3<1:0>
IC3—Input Capture 3	13	13	IFS0<13>	IEC0<13>	IPC3<12:10>	IPC3<9:8>
OC3—Output Compare 3	14	14	IFS0<14>	IEC0<14>	IPC3<20:18>	IPC3<17:16>
INT3—External Interrupt 3	15	15	IFS0<15>	IEC0<15>	IPC3<28:26>	IPC3<25:24>
T4—Timer4	16	16	IFS0<16>	IEC0<16>	IPC4<4:2>	IPC4<1:0>
IC4—Input Capture 4	17	17	IFS0<17>	IEC0<17>	IPC4<12:10>	IPC4<9:8>
OC4—Output Compare 4	18	18	IFS0<18>	IEC0<18>	IPC4<20:18>	IPC4<17:16>
INT4—External Interrupt 4	19	19	IFS0<19>	IEC0<19>	IPC4<28:26>	IPC4<25:24>
T5—Timer5	20	20	IFS0<20>	IEC0<20>	IPC5<4:2>	IPC5<1:0>
IC5—Input Capture 5	21	21	IFS0<21>	IEC0<21>	IPC5<12:10>	IPC5<9:8>
OC5—Output Compare 5	22	22	IFS0<22>	IEC0<22>	IPC5<20:18>	IPC5<17:16>
SPI1E—SPI1 Fault	23	23	IFS0<23>	IEC0<23>	IPC5<28:26>	IPC5<25:24>
SPI1RX—SPI1 Receive Done	24	23	IFS0<24>	IEC0<24>	IPC5<28:26>	IPC5<25:24>
SPI1TX—SPI1 Transfer Done	25	23	IFS0<25>	IEC0<25>	IPC5<28:26>	IPC5<25:24>
U1E—UART1 Error SPI3E—SPI3 Fault I2C3B—I2C3 Bus Collision Event	26	24	IFS0<26>	IEC0<26>	IPC6<4:2>	IPC6<1:0>
U1RX—UART1 Receiver SPI3RX—SPI3 Receive Done I2C3S—I2C3 Slave Event	27	24	IFS0<27>	IEC0<27>	IPC6<4:2>	IPC6<1:0>

U1TX—UART1 Transmitter SPI3TX—SPI3 Transfer Done I2C3M—I2C3 Master Event	28	24	IFS0<28>	IEC0<28>	IPC6<4:2>	IPC6<1:0>
I2C1B—I2C1 Bus Collision Event	29	25	IFS0<29>	IEC0<29>	IPC6<12:10>	IPC6<9:8>
I2C1S—I2C1 Slave Event	30	25	IFS0<30>	IEC0<30>	IPC6<12:10>	IPC6<9:8>
I2C1M—I2C1 Master Event	31	25	IFS0<31>	IEC0<31>	IPC6<12:10>	IPC6<9:8>
CN—Input Change Interrupt	32	26	IFS1<0>	IEC1<0>	IPC6<20:18>	IPC6<17:16>
AD1—ADC1 Convert Done	33	27	IFS1<1>	IEC1<1>	IPC6<28:26>	IPC6<25:24>
PMP—Parallel Master Port	34	28	IFS1<2>	IEC1<2>	IPC7<4:2>	IPC7<1:0>
CMP1—Comparator Interrupt	35	29	IFS1<3>	IEC1<3>	IPC7<12:10>	IPC7<9:8>
CMP2—Comparator Interrupt	36	30	IFS1<4>	IEC1<4>	IPC7<20:18>	IPC7<17:16>
U3E—UART2A Error SPI2E—SPI2 Fault I2C4B—I2C4 Bus Collision Event	37	31	IFS1<5>	IEC1<5>	IPC7<28:26>	IPC7<25:24>
U3RX—UART2A Receiver SPI2RX—SPI2 Receive Done I2C4S—I2C4 Slave Event	38	31	IFS1<6>	IEC1<6>	IPC7<28:26>	IPC7<25:24>
U3TX—UART2A Transmitter SPI2TX—SPI2 Transfer Done IC4M—I2C4 Master Event	39	31	IFS1<7>	IEC1<7>	IPC7<28:26>	IPC7<25:24>
U2E—UART3A Error SPI4E—SPI4 Fault I2C5B—I2C5 Bus Collision Event	40	32	IFS1<8>	IEC1<8>	IPC8<4:2>	IPC8<1:0>
U2RX—UART3A Receiver SPI4RX—SPI4 Receive Done I2C5S—I2C5 Slave Event	41	32	IFS1<9>	IEC1<9>	IPC8<4:2>	IPC8<1:0>
U2TX—UART3A Transmitter SPI4TX—SPI4 Transfer Done IC5M—I2C5 Master Event	42	32	IFS1<10>	IEC1<10>	IPC8<4:2>	IPC8<1:0>
I2C2B—I2C2 Bus Collision Event	43	33	IFS1<11>	IEC1<11>	IPC8<12:10>	IPC8<9:8>
I2C2S—I2C2 Slave Event	44	33	IFS1<12>	IEC1<12>	IPC8<12:10>	IPC8<9:8>
I2C2M—I2C2 Master Event	45	33	IFS1<13>	IEC1<13>	IPC8<12:10>	IPC8<9:8>
FSCM—Fail-Safe Clock Monitor	46	34	IFS1<14>	IEC1<14>	IPC8<20:18>	IPC8<17:16>
RTCC—Real-Time Clock and Calendar	47	35	IFS1<15>	IEC1<15>	IPC8<28:26>	IPC8<25:24>
DMA0—DMA Channel 0	48	36	IFS1<16>	IEC1<16>	IPC9<4:2>	IPC9<1:0>
DMA1—DMA Channel 1	49	37	IFS1<17>	IEC1<17>	IPC9<12:10>	IPC9<9:8>

Continued

Table 6.1: (b) Interrupt IRQ, vector, and bit location – Cont'd

Interrupt Source[a]	IRQ	Vector	Interrupt Bit Location			
	Number	Number	Flag	Enable	Priority	Sub-Priority
Highest Natural Order Priority						
DMA2—DMA Channel 2	50	38	IFS1<18>	IEC1<18>	IPC9<20:18>	IPC9<17:16>
DMA3—DMA Channel 3	51	39	IFS1<19>	IEC1<19>	IPC9<28:26>	IPC9<25:24>
DMA4—DMA Channel 4	52	40	IFS1<20>	IEC1<20>	IPC10<4:2>	IPC10<1:0>
DMA5—DMA Channel 5	53	41	IFS1<21>	IEC1<21>	IPC10<12:10>	IPC10<9:8>
DMA6—DMA Channel 6	54	42	IFS1<22>	IEC1<22>	IPC10<20:18>	IPC10<17:16>
DMA7—DMA Channel 7	55	43	IFS1<23>	IEC1<23>	IPC10<28:26>	IPC10<25:24>
FCE—Flash Control Event	56	44	IFS1<24>	IEC1<24>	IPC11<4:2>	IPC11<1:0>
USB—USB Interrupt	57	45	IFS1<25>	IEC1<25>	IPC11<12:10>	IPC11<9:8>
CAN1—Control Area Network 1	58	46	IFS1<26>	IEC1<26>	IPC11<20:18>	IPC11<17:16>
CAN2—Control Area Network 2	59	47	IFS1<27>	IEC1<27>	IPC11<28:26>	IPC11<25:24>
ETH—Ethernet Interrupt	60	48	IFS1<28>	IEC1<28>	IPC12<4:2>	IPC12<1:0>
IC1E—Input Capture 1 Error	61	5	IFS1<29>	IEC1<29>	IPC1<12:10>	IPC1<9:8>
IC2E—Input Capture 2 Error	62	9	IFS1<30>	IEC1<30>	IPC2<12:10>	IPC2<9:8>
IC3E—Input Capture 3 Error	63	13	IFS1<31>	IEC1<31>	IPC3<12:10>	IPC3<9:8>
IC4E—Input Capture 4 Error	64	17	IFS2<0>	IEC2<0>	IPC4<12:10>	IPC4<9:8>
IC4E—Input Capture 5 Error	65	21	IFS2<1>	IEC2<1>	IPC5<12:10>	IPC5<9:8>
PMPE—Parallel Master Port Error	66	28	IFS2<2>	IEC2<2>	IPC7<4:2>	IPC7<1:0>
U4E—UART4 Error	67	49	IFS2<3>	IEC2<3>	IPC12<12:10>	IPC12<9:8>
U4RX—UART4 Receiver	68	49	IFS2<4>	IEC2<4>	IPC12<12:10>	IPC12<9:8>
U4TX—UART4 Transmitter	69	49	IFS2<5>	IEC2<5>	IPC12<12:10>	IPC12<9:8>
U6E—UART6 Error	70	50	IFS2<6>	IEC2<6>	IPC12<20:18>	IPC12<17:16>
U6RX—UART6 Receiver	71	50	IFS2<7>	IEC2<7>	IPC12<20:18>	IPC12<17:16>
U6TX—UART6 Transmitter	72	50	IFS2<8>	IEC2<8>	IPC12<20:18>	IPC12<17:16>
U5E—UART5 Error	73	51	IFS2<9>	IEC2<9>	IPC12<28:26>	IPC12<25:24>
U5RX—UART5 Receiver	74	51	IFS2<10>	IEC2<10>	IPC12<28:26>	IPC12<25:24>
U5TX—UART5 Transmitter	75	51	IFS2<11>	IEC2<11>	IPC12<28:26>	IPC12<25:24>
(Reserved)	—	—	—	—	—	—
Lowest Natural Order Priority						

[a]Not all interrupt sources are available on all PIC32s.

If the CPU is currently processing an ISR at a particular priority level, and it receives an interrupt request for a vector (and therefore ISR) at the same priority, it will complete its current ISR before servicing the other IRQ, regardless of the subpriority. When the CPU has multiple IRQs pending at a higher priority than its current operating level, the CPU first processes the IRQ with the highest priority level. If multiple IRQs sharing the highest priority are pending, the CPU processes them based on their subpriority. If interrupts have the same priority and subpriority, then their priority is resolved using the "natural order priority" given in Table 6.1, where vectors earlier in Table 6.1 have higher priority.

If the priority of an interrupt vector is zero, then the interrupt is effectively disabled.[2] There are seven enabled priority levels.

Every ISR should clear the interrupt flag (clear the appropriate bit of IFSx to zero), indicating that the interrupt has been serviced. By doing so, after the ISR completes, the CPU is free to return to the program state when the ISR was called. If the interrupt flag is not cleared, then the interrupt will be triggered immediately upon exiting the ISR.

When configuring an interrupt, you set a bit in IECx to 1 indicating the interrupt is enabled (all bits are set to zero upon reset) and assign values to the associated IPCy priority bits. (These priority bits default to zero upon reset, which will keep the interrupt disabled.) You generally never write code setting an IFSx bit to 1.[3] Instead, when you set up the device that generates the interrupt (e.g., a UART or timer), you configure it to set the interrupt flag IFSx upon the appropriate event.

The shadow register set

The PIC32MX's CPU provides an internal *shadow register set* (SRS), which is a full extra set of registers. You can use these extra registers to eliminate the time needed for context save and restore. When processing an ISR using the SRS, the CPU switches to this extra set of internal registers. When it finishes the ISR, it switches back to its original register set, without needing to save and restore them. We see examples using the shadow register set in Section 6.4.

The Device Configuration Register DEVCFG3 determines which priority level is assigned to the shadow register set. The preprocessor command

```
#pragma config FSRSSEL = PRIORITY_6
```

implemented in `NU32.c` and the NU32 bootloader allows only priority level 6 to use the shadow register set. To change this setting you need to either use a standalone program or modify and re-install the bootloader.

[2] One reason for giving an interrupt priority zero is to prevent the ISR from triggering while still having the interrupt source set the IRQ flag, allowing you to monitor the interrupt without it preempting other code.

[3] Setting an IFSx bit to one would cause the interrupt to be pending, just as if hardware had set the flag.

External interrupt inputs

The PIC32 has five digital inputs (INT0 to INT4) that can generate interrupts on rising or falling signal edges. The flag and enable status bits are IFS0 and IEC0, respectively, at bits 3, 7, 11, 15, and 19 for INT0, INT1, INT2, INT3, and INT4, respectively. The priority and subpriority bits are in IPCy⟨28:26⟩ and IPCy⟨25:24⟩ for the input INTy. The SFR INTCON bits 0-4 determine whether the associated interrupt is triggered on a falling edge (bit cleared to 0) or rising edge (bit set to 1). From C, you can access these bits as `IFS0bits.INTxIF`, `IEC0bits.INTxIE`, `IPCxbits.INTxIP`, and `IPCxbits.INTxIS`, where `x` is 0 to 4. The interrupt vector number for each external interrupt is stored as the constant `_EXTERNAL_x_VECTOR`.

Special Function Registers

The SFRs associated with interrupts are summarized below. We omit some fields: for full details, consult the Interrupt Controller section of the Data Sheet, the Interrupt IRQ, Vector, and Bit Location table reproduced earlier (Table 6.1), or the Interrupt section of the Reference Manual.

INTCON The interrupt control SFR determines the interrupt controller mode and the behavior of the five external interrupt pins INT0 to INT4.

INTCON⟨12⟩, or INTCONbits.MVEC: Set to enable multi-vector mode. The bootloader (and `NU32_Startup()`) sets this bit, and you will probably always use this mode.

INTCON⟨x⟩, for x = 0 to 4, or INTCONbits.INTxEP: Determines whether the given external interrupt (INTx) triggers on a rising or falling edge.

1	External interrupt pin x triggers on a rising edge.
0	External interrupt pin x triggers on a falling edge.

INTSTAT The interrupt status SFR is read-only and contains information about the latest IRQ given to the CPU when in single vector mode. We will not need it.

IPTMR The interrupt proximity timer SFR can be used to implement a delay to queue up interrupt requests before presenting them to the CPU. For example, upon receiving an interrupt request, the timer starts counting clock cycles, queuing up any subsequent interrupt requests, until IPTMR cycles have passed. By default, this timer is turned off by INTCON, and we will leave it that way.

IECx, x = 0, 1, or 2 Three 32-bit interrupt enable control SFRs for up to 96 interrupt sources. Setting to 1 enables the given interrupt, clearing to 0 disables it.

IFSx, x = 0, 1, or 2 The three 32-bit interrupt flag status SFRs represent the status of up to 96 interrupt sources. Setting to one requests a given interrupt, and clearing to 0 indicates that no interrupt is requested. Typically, peripherals set the appropriate IFSx bit in response to an event, and user software clears the IFSx bit from within the ISR. The CPU services pending interrupts (those whose IFSx and IECx bits are set) in priority order.

IPCy, y = 0 to 15 Each of the 16 interrupt priority control SFRs contains 5-bit priority and subpriority values for 4 different interrupt vectors (64 vectors total). Interrupts will not be serviced unless the CPU's current priority is less than the interrupt's priority. When the CPU services an interrupt, it sets its current priority to that of the interrupt, and when the ISR completes, its previous priority is restored.

6.3 Steps to Configure and Use an Interrupt

The bootloader (and `NU32_Startup()`) enables the CPU to receive interrupts, setting multi-vector mode by setting INTCONbits.MVEC to 1. After being in the correct mode, there are seven steps to configure and use an interrupt. We recommend your program execute steps 2-7 in the order given below. The details of the syntax are left to the examples in Section 6.4.

1. Write an ISR with a priority level 1-7 using the syntax

   ```
   void __ISR(vector_number, IPLnXXX) interrupt_name(void) { ... }
   ```

   where `vector_number` is the interrupt vector number, `n`=1 to 7 is the priority, `XXX` is either `SOFT` or `SRS`, and `interrupt_name` can be anything but should describe the ISR. `SOFT` uses software context save and restore, and `SRS` uses the shadow register set. (The bootloader on the NU32 allows only priority level 6 to use the SRS, so if you use the SRS, you should use the syntax `IPL6SRS`.) No subpriority is specified in the ISR function. The ISR should clear the appropriate interrupt flag IFSxbit.
2. Disable interrupts at the CPU to prevent spurious generation of interrupts while you are configuring. Although interrupts are disabled by default on reset, `NU32_Startup()` enables them. To disable all interrupts you can use the special compiler instruction `__builtin_disable_interrupts()`.
3. Configure the device (e.g., peripheral) to generate interrupts on the appropriate event. This procedure involves configuring the SFRs of the particular peripheral.
4. Configure the interrupt priority and subpriority in IPCy. **The IPCy priority should match the priority of the ISR defined in Step 1.**
5. Clear the interrupt flag status bit to 0 in IFSx.
6. Set the interrupt enable bit to 1 in IECx.
7. Re-enable interrupts at the CPU. You can use the compiler instruction `__builtin_enable_interrupts()`.

6.4 Sample Code

6.4.1 Core Timer Interrupt

Let us toggle a digital output once per second based on an interrupt from the CPU's core timer. First, we store a value in the CPU's CP0_COMPARE register. Whenever the core timer counter equals this value (CP0_COMPARE), an interrupt is generated. In the interrupt service routine, we reset the core timer counter to 0. Since the core timer runs at half the frequency of the system clock, setting CP0_COMPARE to 40,000,000 toggles the digital output once per second.

To view the ISR in action we toggle LED2 on the NU32 board. We shall use priority level 6, subpriority 0, and the shadow register set.

Code Sample 6.1 `INT_core_timer.c`**. A Core Timer Interrupt Using the Shadow Register Set.**

```
#include "NU32.h"           // constants, funcs for startup and UART
#define CORE_TICKS 40000000 // 40 M ticks (one second)

void __ISR(_CORE_TIMER_VECTOR, IPL6SRS) CoreTimerISR(void) {  // step 1: the ISR
  IFS0bits.CTIF = 0;                // clear CT int flag IFS0<0>, same as IFS0CLR=0x0001
  LATFINV = 0x2;                    // invert pin RF1 only
  _CP0_SET_COUNT(0);                // set core timer counter to 0
  _CP0_SET_COMPARE(CORE_TICKS);     // must set CP0_COMPARE again after interrupt
}

int main(void) {
  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init

  __builtin_disable_interrupts();   // step 2: disable interrupts at CPU
  _CP0_SET_COMPARE(CORE_TICKS);     // step 3: CP0_COMPARE register set to 40 M
  IPC0bits.CTIP = 6;                // step 4: interrupt priority
  IPC0bits.CTIS = 0;                // step 4: subp is 0, which is the default
  IFS0bits.CTIF = 0;                // step 5: clear CT interrupt flag
  IEC0bits.CTIE = 1;                // step 6: enable core timer interrupt
  __builtin_enable_interrupts();    // step 7: CPU interrupts enabled

  _CP0_SET_COUNT(0);                // set core timer counter to 0

  while(1) { ; }
  return 0;
}
```

Following our seven steps to use an interrupt, we have:

Step 1. The ISR.

```
void __ISR(_CORE_TIMER_VECTOR, IPL6SRS) CoreTimerISR(void) {  // step 1: the ISR
  IFS0bits.CTIF = 0;            // clear CT int flag IFS0<0>, same as IFS0CLR=0x0001
  ...
}
```

We can name the ISR anything, so we name it `CoreTimerISR`. The `__ISR` syntax is XC32-specific (not a C standard) and tells the compiler and linker that this function should be treated as an interrupt handler.[4] The two arguments are the interrupt vector number for the core timer, called `_CORE_TIMER_VECTOR` (defined as 0 in `p32mx795f512h.h`, which agrees with Table 6.1), and the interrupt priority level. The interrupt priority level is specified using the syntax `IPLnSRS` or `IPLnSOFT`, where `n` is 1 to 7, `SRS` indicates that the shadow register set should be used, and `SOFT` indicates that software context save and restore should be used. Use `IPL6SRS` if you'd like to use the shadow register set, as in this example, since the device configuration registers on the NU32's PIC32 specify priority level 6 for the shadow register set. You do not specify subpriority in the ISR.

The ISR should clear the interrupt flag in IFS0⟨0⟩ (`IFS0bits.CTIF`), because, according to the table of interrupts (Table 6.1), this bit corresponds to the core timer interrupt. We also need to write to CP0_COMPARE to clear the interrupt, an action specific to the core timer (many interrupts have peripheral-specific actions that are required to clear the interrupt).

Step 2. Disabling interrupts. Since `NU32_Startup()` enables interrupts, we disable them before configuring the core timer interrupt.

```
__builtin_disable_interrupts();   // step 2: disable interrupts at CPU
```

Disabling interrupts before configuring the device that generates interrupts is good general practice, to avoid unwanted interrupts during configuration. In many cases it is not strictly necessary, however.

Step 3. Configuring the core timer to interrupt.

```
_CP0_SET_COMPARE(CORE_TICKS);     // step 3: CP0_COMPARE register set to 40 M
```

This line sets the core timer's CP0_COMPARE value so that an interrupt is generated when the core timer counter reaches `CORE_TICKS`. If the interrupt were to be generated by a peripheral, we would consult the appropriate book chapter or the Reference Manual, to set the SFRs to generate an IRQ on the appropriate event.

Step 4. Configuring interrupt priority.

```
IPC0bits.CTIP = 6;                // step 4: interrupt priority
IPC0bits.CTIS = 0;                // step 4: subp is 0, which is the default
```

These two commands set the appropriate bits in IPCy ($y = 0$, according to Table 6.1). Consulting the file `p32mx795f512h.h` or the Memory Organization section of the Data Sheet

4 `__ISR` is a macro defined in `<sys/attribs.h>`, which `NU32.h` includes.

shows us that the core timer's priority and subpriority bits of IPC0 are called IPC0bits.CTIP and IPC0bits.CTIS, respectively. Alternatively, we could have used any other means to manipulate the bits IPC0⟨4:2⟩ and IPC0⟨1:0⟩, as indicated in Table 6.1, while leaving all other bits unchanged. We prefer using the bit-field `structs` because it is the most readable method. The priority must agree with the ISR priority. It is unnecessary to set the subpriority, which defaults to zero.

Step 5. Clearing the interrupt flag status bit.

```
IFS0bits.CTIF = 0;              // step 5: clear CT interrupt flag
```

This command clears the appropriate bit in IFSx (x = 0 here). A less readable but possibly more efficient alternative would be `IFS0CLR = 1`, to clear the zeroth bit of IFS0.

Step 6. Enabling the core timer interrupt.

```
IEC0bits.CTIE = 1;              // step 6: enable core timer interrupt
```

This command sets the appropriate bit in IECx (x = 0 here). An alternative would be `IEC0SET = 1` to set the zeroth bit of IEC0.

Step 7. Re-enable interrupts at the CPU.

```
__builtin_enable_interrupts();    // step 7: CPU interrupts enabled
```

This compiler built-in function generates an assembly instruction that allows the CPU to process interrupts.

6.4.2 External Interrupt

Code Sample 6.2 causes the NU32's LEDs to illuminate briefly, on a falling edge of external interrupt input pin INT0. You can find the IRQ associated with INT0, and the flag, enable, priority, and subpriority bits in Table 6.1. In this example we use interrupt priority level 2, with software context save and restore.

You can test this program with the NU32 by connecting a wire from the D7/USER pin to the D0/INT0 pin. Pressing the USER button creates a falling edge on digital input RD7 (see the wiring diagram in Figure 2.3) and therefore INT0, which causes the LEDs to flash. You might also notice the issue of switch *bounce*: when you release the button, nominally creating a rising edge, you might see the LEDs flash again. The extra flash occurs because of chattering when mechanical contact between two conductors is established or broken, causing spurious rising and falling edges. Reading reliably from mechanical switches requires a *debouncing* circuit or software; see Exercise 16.

Code Sample 6.2 `INT_ext_int.c`**. Using an External Interrupt to Flash LEDs on the NU32.**

```
#include "NU32.h"          // constants, funcs for startup and UART

void __ISR(_EXTERNAL_0_VECTOR, IPL2SOFT) Ext0ISR(void) { // step 1: the ISR
  NU32_LED1 = 0;                  // LED1 and LED2 on
  NU32_LED2 = 0;
  _CP0_SET_COUNT(0);

  while(_CP0_GET_COUNT() < 10000000) { ; } // delay for 10 M core ticks, 0.25 s

  NU32_LED1 = 1;                  // LED1 and LED2 off
  NU32_LED2 = 1;
  IFS0bits.INT0IF = 0;            // clear interrupt flag IFS0<3>
}

int main(void) {
  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init
  __builtin_disable_interrupts(); // step 2: disable interrupts
  INTCONbits.INT0EP = 0;          // step 3: INT0 triggers on falling edge
  IPC0bits.INT0IP = 2;            // step 4: interrupt priority 2
  IPC0bits.INT0IS = 1;            // step 4: interrupt priority 1
  IFS0bits.INT0IF = 0;            // step 5: clear the int flag
  IEC0bits.INT0IE = 1;            // step 6: enable INT0 by setting IEC0<3>
  __builtin_enable_interrupts();  // step 7: enable interrupts
                                  // Connect RD7 (USER button) to INT0 (D0)
  while(1) {
      ; // do nothing, loop forever
  }

  return 0;
}
```

6.4.3 Speedup Due to the Shadow Register Set

This example measures the amount of time it takes to enter and exit an ISR using context save and restore vs. the SRS. We write two identical ISRs; the only difference is that one uses `IPL6SOFT` and the other uses `IPL6SRS`. The two ISRs are based on the external interrupts INT0 and INT1, respectively. To get precise timing, however, we trigger interrupts in software by setting the appropriate bit of IFS0.

After setting up the interrupts, the program `INT_timing.c` enters an infinite loop. First the core timer is reset to zero, then the interrupt flag is set for INT0. The `main` function then waits until the ISR clears the flag. First the ISR for INT0 records the core timer counter. Next it toggles LED2 and clears the interrupt flag. Finally it logs the time again. After the interrupt exits the `main` function logs the time when control is returned. The timing results are written back to the host computer over the UART. The ISR for INT1 is timed in a similar manner.

These are the results (which are repeated over and over):

```
IPL6SOFT in  19 out  26 total  40 time 1000 ns
 IPL6SRS in  17 out  24 total  37 time  925 ns
```

For context save and restore, it takes 19 core clock ticks (about 38 SYSCLK ticks) to begin executing statements in the ISR; the last ISR statement completes about 7 (14) ticks later; and finally control is returned to `main` approximately 40 (80) total ticks, or 1000 ns, after the interrupt flag is set. For the SRS, the first ISR statement is executed after about 17 (34) ticks; the ISR runs in an identical 7 (14) ticks; and a total of approximately 37 (74) ticks, or 925 ns, elapse between the time the interrupt flag is set and control is returned to `main`.

Although the exact timing may be different for other ISRs and `main` functions, depending on the register context that must be saved and restored (which depends on what the code does), we can make some general observations:

- The ISR is not entered immediately after the flag is set. It takes time to respond to the interrupt request, and instructions in `main` may be executed after the flag is set.
- The SRS reduces the time needed to enter and exit the ISR, approximately 75 ns total in this case.
- Simple ISRs can be completed less than a microsecond after the interrupt event occurs.

The sample code is below. We note that this example also serves to demonstrate different methods for setting bit fields in ISRs. We hope that after viewing this code you will agree that using the bit-field `structs` makes the code easier to read and understand.

Code Sample 6.3 `INT_timing.c`**. Timing the Shadow Register Set vs. Typical Context Save and Restore.**

```
#include "NU32.h"          // constants, funcs for startup and UART
#define DELAYTIME 40000000 // 40 million core clock ticks, or 1 second

void delay(void);

static volatile unsigned int Entered = 0, Exited = 0;  // note the qualifier "volatile"

void __ISR(_EXTERNAL_0_VECTOR, IPL6SOFT) Ext0ISR(void) {
  Entered = _CP0_GET_COUNT();       // record time ISR begins
  IFS0CLR = 1 << 3;                 // clear the interrupt flag
  NU32_LED2 = !NU32_LED2;           // turn off LED2
  Exited = _CP0_GET_COUNT();        // record time ISR ends
}

void __ISR(_EXTERNAL_1_VECTOR, IPL6SRS) Ext1ISR(void) {
  Entered = _CP0_GET_COUNT();       // record time ISR begins
  IFS0CLR = 1 << 7;                 // clear the interrupt flag
  NU32_LED2 = !NU32_LED2;           // turn on LED2
  Exited = _CP0_GET_COUNT();        // record time ISR ends
}

int main(void) {
  unsigned int dt = 0;
  unsigned int encopy, excopy;      // local copies of globals Entered, Exited
  char msg[128] = {};

  NU32_Startup(); // cache on, min flash wait, interrupts on, LED/button init, UART init
  __builtin_disable_interrupts();   // step 2: disable interrupts at CPU
  INTCONSET = 0x3;                  // step 3: INT0 and INT1 trigger on rising edge
  IPC0CLR = 31 << 24;               // step 4: clear 5 priority and subp bits for INT0
  IPC0 |= 24 << 24;                 // step 4: set INT0 to priority 6 subpriority 0
  IPC1CLR = 0x1F << 24;             // step 4: clear 5 priority and subp bits for INT1
  IPC1 |= 0x18 << 24;               // step 4: set INT1 to priority 6 subpriority 0
  IFS0bits.INT0IF = 0;              // step 5: clear INT0 flag status
  IFS0bits.INT1IF = 0;              // step 5: clear INT1 flag status
  IEC0SET = 0x88;                   // step 6: enable INT0 and INT1 interrupts
  __builtin_enable_interrupts();    // step 7: enable interrupts
  while(1) {
    delay();                        // delay, so results sent back at reasonable rate
    _CP0_SET_COUNT(0);              // start timing
    IFS0bits.INT0IF = 1;            // artificially set the INT0 interrupt flag
    while(IFS0bits.INT0IF) {
        ;                           // wait until the ISR clears the flag
    }
    dt = _CP0_GET_COUNT();          // get elapsed time
    __builtin_disable_interrupts(); // good practice before using vars shared w/ISR
    encopy = Entered;               // copy the shared variables to local copies ...
    excopy = Exited;                // ... so the time interrupts are off is short
    __builtin_enable_interrupts();  // turn interrupts back on quickly!
    sprintf(msg,"IPL6SOFT in %3d out %3d total %3d time %4d ns\r\n"
        ,encopy,excopy,dt,dt*25);
    NU32_WriteUART3(msg);           // send times to the host

    delay();                        // same as above, except for INT1
    _CP0_SET_COUNT(0);
    IFS0bits.INT1IF = 1;            // trigger INT1 interrupt
    while(IFS0bits.INT1IF) {
        ;                           // wait until the ISR clears the flag
    }
    dt = _CP0_GET_COUNT();
    __builtin_disable_interrupts();
    encopy = Entered;
    excopy = Exited;
    __builtin_enable_interrupts();
    sprintf(msg," IPL6SRS in %3d out %3d total %3d time %4d ns\r\n"
        ,encopy,excopy,dt,dt*25);
    NU32_WriteUART3(msg);
  }
  return 0;
}

void delay() {
  _CP0_SET_COUNT(0);
  while(_CP0_GET_COUNT() < DELAYTIME) {
      ;
  }
}
```

6.4.4 Sharing Variables with ISRs

Code Sample 6.3 was the first to share variables between an ISR and other functions. Namely, `Entered` and `Exited` are used in both ISRs as well as `main`. Sharing data between ISRs and mainline code is one area where using global variables is required, even though they should be generally avoided. We use the `static` keyword so at least, in a larger project, the variables are limited in scope to the file in which they are declared.

This code demonstrates two good practices when sharing variables with ISRs:

(1) Using the type qualifier `volatile`

By putting the qualifier `volatile` in front of the type in the global variable definition

```
static volatile unsigned int Entered = 0, Exited = 0;
```

we tell the compiler that external processes (namely, an ISR that may be triggered at an unknown time) may read or write the variables at any time. Therefore, any optimizations the compiler performs should not take shortcuts in generating assembly code associated with a `volatile` variable. For example, if you had code of the form

```
static int i = 0;    // global variable shared by functions and an ISR

void myFunc(void) {
  i = 1;
  // some other code that doesn't use or affect i
  i = 2;
  // some code that uses i
}
```

a compiler running optimizations might not generate any code for `i = 1` at all, believing that the value 1 for `i` is never used. If an external interrupt triggered during execution of the code between `i = 1` and `i = 2`, however, and the ISR used the value of `i`, it would use the wrong value (perhaps the originally initialized value `i = 0`).

To correct this problem, the declaration of the global variable `i` should be

```
static volatile int i = 0;
```

The `volatile` qualifier ensures that the compiler will emit full assembly code for any reads or writes of `i`. The compiler does not assume anything about the value of `i` or whether it is changed or used by processes that it does not know about. This is why all SFRs are declared as `volatile` in `p32mx795f512h.h`; their values can be changed by processes external to the CPU.

(2) Enabling and disabling interrupts

Consider a scenario where the mainline code and an ISR share a 64-bit `long double` variable. To load the variable into two of the CPU's 32-bit registers, one assembly instruction first loads the most significant 32 bits into one register. Then the process is interrupted, the ISR modifies the variable in RAM, and control returns to the main code. At that point, the next assembly instruction loads the lower 32 bits of the new value of the `long double` in RAM into the other CPU register. Now the CPU registers have neither the correct variable value from before nor after the ISR.

To prevent data corruption like this, interrupts can be disabled before reading or writing the shared variables, then re-enabled afterward. If an IRQ is generated during the time that the CPU is ignoring interrupts, the IRQ will simply wait until the CPU is accepting interrupts again.

Interrupts should not be disabled for long, as this defeats the purpose of interrupts. In the sample code, the time that interrupts are disabled is minimized by simply copying the shared variables to local copies during this period, rather than performing time-consuming operations with them. This avoids having the interrupts disabled during the `sprintf` command, which can take many CPU cycles.

In many cases it is not necessary to disable interrupts before using shared variables. (For example, it was not necessary in the sample code above.) Determining whether such precautions are necessary sometimes depends not only on the algorithms you employ but how those algorithms translate into assembly instructions. Disabling interrupts is usually the simpler and safer option.[5] As long as you limit the code that executes while interrupts are disabled, you will delay any interrupts that would have occurred by at most a few hundred nanoseconds; most applications can tolerate such delays.

6.5 Chapter Summary

- The CPU of the PIC32MX supports 96 interrupt requests (IRQs) and 64 interrupt vectors, and therefore up to 64 interrupt service routines (ISRs). Therefore, some IRQs share the same ISR. For example, all IRQs related to UART1 (data received, data transmitted, error) have the same interrupt vector.
- The PIC32 can be configured to operate in single vector mode (all IRQs result in a jump to the same ISR) or in multi-vector mode. The bootloader (and `NU32_Startup()`) puts the NU32 in multi-vector mode.

[5] Errors caused by incorrect sharing of data between interrupts and mainline code are called *race conditions*. Race conditions are notoriously difficult to discover and fix; they may only appear intermittently because they closely depend on timing.

- 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.