Interrupts

Interrupts

Interrupts are like receiving a telephone call while you are in a face-to-face meeting:

- The phone rings (ie, an interrupt is sent)
- Tell the person you are meeting with to please wait
- Answer the phone, provide a response (eg: take a message), hang up
- Resume the meeting, picking up where you left off.

Efficiency demands that you:

- Keep the phone call short
- Don't allow a second call to interrupt the first call (no call waiting)
- Satisfy the caller so they do not immediately call you back after hanging up.

From a computer system perspective, the CPU is always busy "in a meeting" by running your main program: it must fetch the instruction, execute it, and then advance to the next instruction.

To get the attention of the CPU, a device can **interrupt** the currently executing instruction by sending a logic '1' signal to a dedicated **interrupt request** input pin on the CPU. This may happen when a KEY was just pressed, or when a time delay elapses.

The CPU usually has a handful of these interrupt request pins, often named IRQ0, IRQ1, …, IRQn. The computer system hardware designer connects each device to a different IRQ pin. In the UBC DE1 Media system, IRQ3 is connected to the COUNTER device.

Roughly speaking, whenever an IRQ pin is set to a 1, the CPU stops the current instruction before it is executed, and then jumps to a special subroutine called an **interrupt service routine** or ISR. **Each device gets its own ISR**, and this subroutine responds to events only from that device.

The ISR:

1. **cannot accept** any input parameters, or return any result
2. **must respond** to the device and handle the event; it can do this by reading the state of the device and various memory locations, then writing new values back to the device or memory locations
3. **must clear** the source of the interrupt, by telling the device to stop sending it
4. **must not pause or delay** unnecessarily; it should be quick to exit
5. **must not be interrupted** by another device.

Keep in mind that the ISR is only for hardware device interrupts. The only way the ISR should be executed is by having the device raise its IRQ pin to '1'. That is, the main program should **never directly call the ISR as a subroutine**.

After the ISR completes, it returns to the main program at exactly the instruction that was interrupted, and it again attempts to execute that instruction.

Interrupt Issues

With interrupts, it is possible to **get stuck in an infinite loop**. If you forget step (3), clearing the source of the interrupt, the device will still be sending a 1 on the IRQ pin when the ISR finishes. If this happens, the CPU will be immediately interrupted after exiting the ISR; it will not get a chance to run the main program at all. In this case, the program gets stuck in an infinite loop that repeatedly runs the ISR.

While an ISR is running, **it must not be interrupted by another device**. As a result, interrupts are automatically disabled just before the ISR starts, and they are automatically re-enabled just after the ISR finishes. This is also why **an ISR should never wait or delay** unnecessarily. If another device sends an interrupt to the CPU, it must wait for the current ISR to finish executing completely before its own ISR can be executed. Get your business done quickly, and get out!

Since an interrupt can occur at any point in your main program, the ISR **must not** modify any CPU registers. Otherwise, the main program will behave unpredictably, as register values could change at any time and alter the execution of the program. To assure that registers will not be changed, the interrupt process will save and restore register values from the stack. This process is explained next.

Interrupt Process Details

A specific sequence of events takes place after an interrupt is sent. The start of this sequence is automatically done by the CPU hardware itself, and then it quickly passes control back to software to finish the sequence.

Upon receiving the interrupt signal any IRQ pin, the CPU hardware:

- **disables interrupts**
- stops executing the current instruction; this instruction must be re-started later
- saves the current program counter into register r29, named "ea" *exception return address*
- jumps to the instruction at address 0x20.

At this point, the rest of the sequence follows by executing a small software routine called the **exception handler**. It starts at address 0x20, and is included in 259library.c. The exception handler is software, but it always does this:

- saves all CPU registers on the stack
- determines which interrupt was triggered, and calls the corresponding ISR
- executes the ISR
- restores all CPU registers from the stack
- finishes with the "eret" instruction
    - The ERET instruction **re-enables interrupts** and returns to the instruction pointed to by "ea", the point at which the main program was interrupted.

Notice that **interrupts are disabled throughout the entire sequence**. In theory, it is possible to re-enable interrupts inside the ISR. However, this is an advanced use of interrupts that can be very difficult to debug.

Programming with Interrupts

To begin using interrupts with any CPU or device, many things must be configured:

- CPU
  - The CPU must be configured to receive interrupts (in general).
  - The CPU must be configured to receive interrupts from each device.
- Device
  - The device must be configured to send an interrupt.
  - Optionally, devices usually have configuration options that specify exactly which type of event(s) should send an interrupt.

In addition, the program must abide by certain restrictions:

- One ISR must be written for each device.
- The ISR must clear the cause of the interrupt before exiting.
- The ISR must be registered (associated) with the device's IRQ pin number.

Interrupts Example – Using the COUNTER Device

Programming with interrupts involves a lot of details. We will start with the simplest possible interrupt example, shown below in Figure 1.

```
#include "259macros.h"

/* global variables */
int counter = 0;

/* This ISR will be called every 100ms */
/* Display a counter on the red LEDs */
void counterISR()
{
            /* remember: no waiting in an ISR */
            counter++;
            *pLEDR = counter;

            /* clear source of COUNTER interrupt before returning */
            *pCOUNTER_STATUS = 1; /* Device: send interrupts, clear existing interrupt */
}

/* This routine configures device-specific details about interrupts */
void enableCounterIRQ( int interval_cycles, ptr_to_function newISR )
{
            registerISR( IRQ_COUNTER, newISR ); /* specify which ISR to call with COUNTER interrupts */

            *pCOUNTER           = -interval_cycles; /* initial counter value */
            *pCOUNTER_RELOAD    = -interval_cycles; /* on overflow, start with this value */

            *pCOUNTER_STATUS    = 1;  /* Device: send interrupts, clear existing interrupt */
            enableInterrupt( IRQ_COUNTER ); /* CPU: allow it to receive COUNTER interrupts */
}


int main( int argc, char *argv[] )
{
            /* CPU: clears all interrupt enables for individual devices, then enables interrupts */
            initInterrupts();

            /* Device: set up COUNTER to interrupt and call counterISR every 100ms. */
            enableCounterIRQ( 100*ONE_MS, counterISR );

            /* Main loop is busy doing nothing, forever */
            while( 1 )
                ;

            return 0; /* never gets here */
}
```

Figure 1. Interrupts example in C using COUNTER.

The purpose of this program is to increment a global variable **counter** every 100ms, and to display this value on the red LEDs. The COUNTER hardware device is configured to generate an interrupt precisely every 100ms, which is after 5,000,000 clock cycles have elapsed (assuming 50MHz clock).

The main program sets things up, first by initializing the CPU and then by initializing the device. First, the subroutine **initInterrupts();** is used to initialize the CPU to receive interrupts in general; its source code is provided in 259library.c. Second, the subroutine **enableCounterIRQ();** is used to initialize the COUNTER device to send interrupts; it will be explained shortly. Finally, the main program enters an infinite loop, where it continuously does nothing. This loop is meant to simulate doing "real work", but we kept it as simple as possible by removing the work!

In enableCounterIRQ(), all device-specific features are configured. The parameters for enableCounterIRQ() represent two values: a time delay, 100ms, and the name of the ISR to be used with the counter device, **counterISR**. The time delay value is used to initialize the counter (described below). The second parameter is actually a pointer to a function; the original name is lost, but the new name newISR simply points to counterISR. A pointer to a function is just like any pointer – in this case, it is the address of the first instruction of the ISR subroutine. The call **registerISR(IRQ_COUNTER, newISR);** associates the pointer to counterISR with the interrupt pin defined for the COUNTER device (IRQ3). The value of IRQ_COUNTER is a constant (3) that is defined in 259macros.h – its value depends upon the hardware design, so it must not be changed! The registerISR() subroutine is provided in 259library.c.

The operation of this program is heavily dependent upon details of the COUNTER device.

- The counter always counts up by 1 every clock cycle, at a rate of 50MHz (20ns).
- When the counter wraps around (ie, counts past 0xFFFFFFFF), it will be automatically re-initialized with the value stored at the COUNTER_RELOAD address. Normally, the value stored at COUNTER_RELOAD is 0, allowing the counter to wrap around to 0.
- If 1 is stored at the COUNTER_STATUS address, the counter sends an interrupt to the CPU by setting pin IRQ3 to '1' when the counter wraps around (increments past 0xFFFFFFFF).
- The process of writing ***any value*** to COUNTER_STATUS clears the current interrupt by resetting the IRQ3 pin back to '0'. Thus, writing a 0 to COUNTER_STATUS clears the current interrupt *and* disables any further counter interrupts, whereas writing a 1 clears the current interrupt *and* leaves counter interrupts enabled for next time.

The enableCounterIRQ() routine configures the counter in three ways:

- The counter is initialized with a value of -5,000,000. This allows it to count up towards 0. It will ultimately wrap around after 100ms.
- The reload value is also set to -5,000,000. After the current counter wraps around, this will set the time duration for the next wrap around to be 100ms.
- It enables interrupts by writing 1 to COUNTER_STATUS.

Note that the process of merely writing ***any value*** to COUNTER_STATUS will stop the counter from sending its ***current*** interrupt. Thus, writing a 0 will clear the current interrupt *and* disable new interrupts from being sent; writing a 1 will clear the current interrupt *and* enable an interrupt to be sent the next time the counter wraps around.

NAME: _____ STUDENT #: _____

EECE 259: Introduction to Microcomputers **Lecture Quiz** **Mar 24, 2011**

An **ABS brake controller** does two things at the same time. First, it monitors an input signal to indicate that the wheel is spinning. Second, if the wheel ever stops spinning, it must quickly release and engage the brakes (pulsing them) several times per second. By pulsing the brakes, the wheel is allowed to spin again and this restores tire traction. For this problem, **write a main program and interrupt service routine** to:

- **Detect wheel spin.** First, your program should continuously poll KEY3 and count the number of 0-to-1 transitions. For this, KEY3 represents an encoder attached to the wheel: if the wheel is spinning rapidly, you will get many 0-to-1 transitions per 100ms. Due to speed limits, at least 1ms will pass before the next 0-to-1 transition occurs. However, you don't know exactly when it will occur. Hence, the polling of KEY3 must be done frequently enough to not miss any transitions!
- **If no spin, pulse the brakes.** Second, your program should apply the brakes by sending a '1' to LEDG0. Every 100ms, you should check that at least 5 encoder transitions have been detected, suggesting the wheel is still spinning. If fewer than 5 transitions have occurred, alternately apply & release the brakes every 100ms. If at least 5 transitions have been detected, the wheel is still spinning so you should apply the brakes again for the next 100ms.

```
#include "259macros.h"

/* global variables */
int counter = 0;
int brake_flag = 0;

int main(...)
{
    initInterrupts();
    enableCounterIRQ(100*ONE_MS,cntrISR);

      /* write your code below */
      while(1) {

      }
}
```

```
/* this ISR will be called every 100ms */
void cntrISR()
{
    /* remember: no waiting in here */
}
```

NAME: _____ STUDENT #: _____

EECE 259: Introduction to Microcomputers **Lecture Quiz** **Mar 24, 2011**

An **ABS brake controller** does two things at the same time. First, it monitors an input signal to indicate that the wheel is spinning. Second, if the wheel ever stops spinning, it must quickly release and engage the brakes (pulsing them) several times per second. By pulsing the brakes, the wheel is allowed to spin again and this restores tire traction. For this problem, **write a main program and interrupt service routine** to:

- **Detect wheel spin.** First, your program should continuously poll KEY3 and count the number of 0-to-1 transitions. For this, KEY3 represents an encoder attached to the wheel: if the wheel is spinning rapidly, you will get many 0-to-1 transitions per 100ms. Due to speed limits, at least 1ms will pass before the next 0-to-1 transition occurs. However, you don't know exactly when it will occur. Hence, the polling of KEY3 must be done frequently enough to not miss any transitions!
- **If no spin, pulse the brakes.** Second, your program should apply the brakes by sending a '1' to LEDG0. Every 100ms, you should check that at least 5 encoder transitions have been detected, suggesting the wheel is still spinning. . If fewer than 5 transitions have occurred, alternately apply & release the brakes every 100ms. If at least 5 transitions have been detected, the wheel is still spinning so you should apply the brakes again for the next 100ms.

```
#include "259macros.h"

/* global variables */
int counter = 0;
int brake_flag = 0;

int main(...)
{
    initInterrupts();
    enableCounterIRQ(100*ONE_MS,cntrISR);

    /* write your code below */
    while(1) {

        /* wait while KEY3 == 1 */
        while( (*pKEY & 8) )
            *pLEDR = counter;

        /* wait while KEY3 == 0 */
        while( !(*pKEY & 8) )
            *pLEDR = counter;

        /* count 0-to-1 transition */
        counter++;


    }
}

/* The main() routine above displays
 * the value of counter on LEDR to aid
 * debugging. Since the counter value can
 * be reset at any time by cntrISR(),
 * both while() loops must update LEDR
 * continuously.
 */
```

```
/* this ISR will be called every 100ms */
void cntrISR()
{
    /* remember: no waiting in here */


    /* clear source of interrupt */
    *pCOUNTER_STATUS = 1;

    /* wheel spinning, apply brakes */
    if( counter >= 5 )
            brake_flag = 1;

    /* not spinning, pulse brakes 100ms*/
    else
            brake_flag = !brake_flag;

    /* show brake flag, reset counter */
    *pLEDG = brake_flag;
    counter = 0;

}
```

Communication between ISR and Main Program

One of the most difficult things to get correct is the communication between your ISR and the main part of your program. Getting the right behavior can be tricky because of very subtle timing bugs. You may not even notice the bugs right away because it works 99.9999% of the time. However, you must be very careful to get it working 100% of the time – especially in mission-critical applications where human life is at stake (airplane controllers, nuclear power plant controls, missile guidance, medical instruments, etc).

In C, communication between an ISR and main program occurs through shared (global) variables that are stored in memory. For example, consider the following main program fragment which increments a global variable:

```
Main-C:         counter++;              /* increment a counter (global variable) */

Main-assembly: ldw     r8, 0(r16)      /* read value from memory*/
               addi    r8, r8, 1       /* modify value */
               stw     r8, 0(r16)      /* write value back to memory */
```

…and the ISR contains code to decrement the same variable:

```
ISR-C:          counter--;

ISR-assembly:  ldw     r8, 0(r16)
               subi    r8, r8, 1
               stw     r8, 0(r16)
```

This code has a problem: depending on exactly which instruction in Main-assembly is interrupted, the results can be catastrophically different.

- First, consider what happens if the *ldw* in the main program is interrupted. The ISR decrements the counter first, the main program resumes, executes the ldw, reads the decremented value from memory, increments it, writes it back to memory. **This produces the correct behavior**: the net counter value is unchanged.
- Second, consider what happens if *addi* or *stw* is interrupted instead. The ISR decrements the counter value, the main program resumes after the ldw and r8 contains the *old* counter value, so it increments this old value and it writes back to memory. **This produces incorrect behavior:** the net counter value is +1, and the decrement by the ISR is lost!

This error occurs because both the main program and ISR are modifying shared variables. Any instruction sequences in the main program that use a series of statements to "read-modify-write" the variable (eg, increment) must be protected from interruption. We call this sequence a **critical section**. Critical sections must execute **atomically**, meaning the sequence of operations is indivisible: once started, it should finish without interruption. The solution is to momentarily disable interrupts in the main program:

```
Main-C:         disableInterrupts();   /* start critical section */
                counter++;
                enableInterrupts();    /* end critical section */

Main-assembly: wrctl   status, r0     /* disable interrupts, start critical section */
               ldw     r8, 0(r16)
               addi    r8, r8, 1
               stw     r8, 0(r16)
               movi    r8, 1
               wrctl   status, r8     /* enable interrupts, end critical section */
```

If an interrupt arrives during the critical section, it will not be lost. It will be executed as soon as interrupts are re-enabled. Thus, you should re-enable them as quickly as you can (no waiting or time delays).

NAME: _____ STUDENT #: _____

EECE 259: Introduction to Microcomputers **Lecture Quiz** **Mar 24, 2011**

An **ABS brake controller** does two things at the same time. First, it monitors an input signal to indicate that the wheel is spinning. Second, if the wheel ever stops spinning, it must quickly release and engage the brakes (pulsing them) several times per second. By pulsing the brakes, the wheel is allowed to spin again and this restores tire traction. For this problem, **write a main program and interrupt service routine** to:

- **Detect wheel spin.** First, your program should continuously poll KEY3 and count the number of 0-to-1 transitions. For this, KEY3 represents an encoder attached to the wheel: if the wheel is spinning rapidly, you will get many 0-to-1 transitions per 100ms. Due to speed limits, at least 1ms will pass before the next 0-to-1 transition occurs. However, you don't know exactly when it will occur. Hence, the polling of KEY3 must be done frequently enough to not miss any transitions!
- **If no spin, pulse the brakes.** Second, your program should apply the brakes by sending a '1' to LEDG0. Every 100ms, you should check that at least 5 encoder transitions have been detected, suggesting the wheel is still spinning. . If fewer than 5 transitions have occurred, alternately apply & release the brakes every 100ms. If at least 5 transitions have been detected, the wheel is still spinning so you should apply the brakes again for the next 100ms.

```
#include "259macros.h"

/* global variables */
int counter = 0;
int brake_flag = 0;

int main(...)
{
    initInterrupts();
    enableCounterIRQ(100*ONE_MS,cntrISR);

    /* write your code below */
    while(1) {

        /* wait while KEY3 == 1 */
        while( (*pKEY & 8) )
            *pLEDR = counter;

        /* wait while KEY3 == 0 */
        while( !(*pKEY & 8) )
            *pLEDR = counter;

        /* count 0-to-1 transition */
        disableInterrupts();
        counter++;
        enableInterrupts();
    }
}

/* The main() routine above displays
 * the value of counter on LEDR to aid
 * debugging. Since the counter value can
 * be reset at any time by cntrISR(),
 * both while() loops must update LEDR
 * continuously.
 *
 * The disableInterrupts() and
 * enableInterrupts() routines protect
 * a critical section containing a
 * read-modify-write sequence of the
 * shared counter variable.
 */
```

```
/* this ISR will be called every 100ms */
void cntrISR()
{
    /* remember: no waiting in here */

    /* clear source of interrupt */
    *pCOUNTER_STATUS = 1;

    /* wheel spinning, apply brakes */
    if( counter >= 5 )
            brake_flag = 1;

    /* not spinning, pulse brakes 100ms*/
    else
            brake_flag = !brake_flag;

    /* show brake flag, reset counter */
    *pLEDG = brake_flag;
    counter = 0;

}
```

UBC DE1 Media Computer – Devices that can Interrupt

We have already presented the COUNTER device which uses pin IRQ3. The interrupt pins used for all devices are shown in Figure 2.

- This is correct for the (v3) version of the UBC DE1 Media computer system.
- The default Altera DE1 Media computer system assignments are slightly different.

Using interrupts with the KEY device will be described below. For the remaining devices, you can read about how to configure and control their behavior with interrupts in the Altera DE1 Media Computer manual. This is located on the course web site in the (homework files) section as file **DE1_Media_Computer.pdf**. You can also find the file installed on your PC at a location like:
C:\Altera\91sp2\University_Program\NiosII_Computer_Systems\DE1\DE1_Media_Computer\doc\DE1_Media_Computer.pdf

Interrupt	**Device Description**
irq0	Countdown timer device
irq1	KEY device
irq2	GPIO1 port
irq3	COUNTER device
irq6	Audio device
irq7	PS2 (mouse/keyboard) port
irq8	JTAG terminal device
irq10	RS232 port
irq11	GPIO0 port

Figure 2. Interrupt assignments in the UBC DE1 Media computer system.

Interrupts Example – Using the KEY Device

As a final example, we illustrate how to call interrupts when KEY3 or KEY2 are pressed in Figure 3. This program is just a small modification to the original example in Figure 1. The main changes are:

- The subroutine call **enableKeyIRQ(keys_to_watch, keyISR);** is used to initialize the KEY device to send interrupts. The first parameter, keys_to_watch, is a mask value which indicates which keys are allowed to send interrupts. In this example, only KEY3 and KEY2 are allowed to send interrupts. By writing the mask value to KEY_IRQENABLE, interrupts are enabled for those keys. The second parameter, keyISR, is the name of the ISR to be called.
- keyISR() is executed via interrupt. Since the interrupt may be slightly delayed (eg, by a critical section or another ISR which has momentarily disabled interrupts), it is possible the key that was originally pressed has already been released. Hence, the value of *pKEY, which only contains live real-time status of the keys, is useless. Instead, the special location KEY_EDGECAPTURE is used. When a key is pressed, the rising edge sets the corresponding bit in KEY_EDGECAPTURE. The interrupt is cleared by writing 0 to KEY_EDGECAPTURE.

```c
#include "259macros.h"

/* global variables */
int counter = 0;
int incr = 1;

/* This ISR will be called every 100ms */
/* Display a counter on the red LEDs */
void counterISR()
{
		/* remember: no waiting in an ISR */
		counter += incr;
		*pLEDR = counter;

		/* clear source of COUNTER interrupt before returning */
		*pCOUNTER_STATUS = 1; /* Device: send interrupts, clear existing interrupt */
}

/* This routine configures device-specific details about interrupts */
void enableCounterIRQ( int interval_cycles, ptr_to_function newISR )
{
		registerISR( IRQ_COUNTER, newISR ); /* specify which ISR to call with COUNTER interrupts */

		*pCOUNTER		= -interval_cycles; /* initial counter value */
		*pCOUNTER_RELOAD	= -interval_cycles; /* on overflow, start with this value */

		*pCOUNTER_STATUS	= 1; /* Device: send interrupts, clear existing interrupt */
		enableInterrupt( IRQ_COUNTER ); /* CPU: allow it to receive COUNTER interrupts */
}

/* This ISR will be called every time KEY3 or KEY2 is pressed */
/* On KEY3, change the increment direction */
/* On KEY2, load a new increment amount from switches */
void keyISR()
{
		/* remember: no waiting in an ISR */
		int keypress = *pKEY_EDGECAPTURE;

		if( keypress & 8 )	incr = -incr;
		if( keypress & 4 )	incr = *pSWITCH;

		/* clear source of KEY interrupt before returning */
		*(pKEY_EDGECAPTURE) = 0;
}

/* This routine configures device-specific details about interrupts */
void enableKeyIRQ( int keys_to_watch, ptr_to_function newISR )
{
		registerISR( IRQ_KEY, newISR );	/* specify which ISR to call with KEY interrupts */
		*pKEY_IRQENABLE = keys_to_watch; /* Device: to send interrupts for KEY3,KEY2 */
		enableInterrupt( IRQ_KEY );	/* CPU: allow it to receive KEY interrupts */
}


int main( int argc, char *argv[] )
{
		int keys_to_watch;

		/* CPU: clears all interrupt enables for individual devices, then enables interrupts */
		initInterrupts();

		/* Device: set up COUNTER to interrupt and call counterISR every 100ms. */
		enableCounterIRQ( 100*ONE_MS, counterISR );


		/* Device: set up KEY to interrupt every time KEY3 or KEY2 is pressed */
		keys_to_watch = 0x8 | 0x4 ;
		enableKeyIRQ( keys_to_watch, keyISR );

		/* Main loop is busy doing nothing, forever */
		while( 1 )
		    ;

		return 0; /* never gets here */
}
```

Figure 3. Interrupts example in C using COUNTER and KEY.

NAME: _____ STUDENT #: _____ L34-1

EECE 259: Introduction to Microcomputers **Lecture Quiz** **Mar 25, 2011**

An **ABS brake controller** does two things at the same time. First, it monitors an input signal to indicate that the wheel is spinning. Second, if the wheel ever stops spinning, it must quickly release and engage the brakes (pulsing them) several times per second. By pulsing the brakes, the wheel is allowed to spin again and this restores tire traction. For this problem, **write a main program and 2 interrupt service routines** to:

- **Detect wheel spin.** First, your program should use interrupts with KEY3 and count the number of 0-to-1 transitions. For this, KEY3 represents an encoder attached to the wheel: if the wheel is spinning rapidly, you will get many 0-to-1 transitions per 100ms. Due to speed limits, at least 1ms will pass before the next 0-to-1 transition occurs. However, you don't know exactly when it will occur. Hence, use interrupts!
- **If no spin, pulse the brakes.** Second, your program should apply the brakes by sending a '1' to LEDG0. Every 100ms, you should check that at least 5 encoder transitions have been detected, suggesting the wheel is still spinning. . If fewer than 5 transitions have occurred, alternately apply & release the brakes every 100ms. If at least 5 transitions have been detected, the wheel is still spinning so you should apply the brakes again for the next 100ms.

```
#include "259macros.h"

/* global variables */
int counter = 0;
int brake_flag = 0;

int main(...)
{
    initInterrupts();
    enableCounterIRQ(100*ONE_MS,cntrISR);
    enableKeyIRQ( 0x8, keyISR );
    /* write your code below */
    while(1) {









      }
}
```

```
/* this ISR will be called every 100ms */
void cntrISR()
{
    /* remember: no waiting in here */









}

void keyISR()
{
    /* remember: no waiting in here */









}
```

A NOTE TO 259 STUDENTS:

Interrupts involve a lot of details.

The details presented after this page provide further background on exactly what happens at the CPU logic and assembly code levels. This may better help you understand the previous pages, as they define exactly how interrupts work. It may also help you understand how use interrupts in assembly code for your project. However, I will not test you on this material specifically – it is too detailed, and it is best left as a reference manual that you consult during a project (not during an examination).

I expect you to understand everything discussed prior to this page. In particular:

From 259library.c:

initInterrupts(); // clears history of all registered ISRs with IRQs, disables each specific device interrupt, enables CPU to receive interrupts

enableInterrupts(); // interrupts can be received by CPU from any specific device interrupt that is enabled
disableInterrupts(); // CPU ignores all interrupts

enableInterrupt(IRQ_NUM); // enables specific device interrupt to be received by CPU
disableInterrupt(IRQ_NUM); // CPU ignores specific device interrupt

registerISR(IRQ_NUM, ISR_name); // registers ISR_name with IRQ_NUM; interrupts received from IRQ_NUM will cause ISR_name() to be called for service

You need to know the purpose of these functions and how to use them.

You do not need to know the source code for these functions (although it is provided in 259library.c, and reading it may help you understand things better).

From example code (irq-example.c, irq-example2.c):

enableCounterIRQ(delay_amount, counterISR);
enableKeyIRQ(keymask, keyISR);

You need to know the purpose of these functions and how to use them.

You need to be familiar with the internal details of these functions, but you do not need to memorize the internal details. If needed, I will provide (most of the) internal details on a test, but may leave blank sections. You may be asked to explain, alter, or fill in the internal details.

I cannot guarantee that quiz/exam question(s) on interrupts will be exactly like the Lecture Quiz.