## Homework

- Reading
  - Intel 8254 Programmable Interval Timer (PIT) Data Sheet
- Machine Projects
  - Continue on MP3
- Labs
  - Continue in labs with your assigned section

# Restrictions on ISR Code

- Software that was executing never got a chance to save any registers it was using!
- ISR must save context (not use ANY registers without pushing them on stack and popping them off before returning from the interrupt)
- ISR must finish its execution promptly
- Two additional considerations:
  - Interrupt windows / critical regions
  - C keyword “volatile”

## ISR and Background Code

- ISR and background code design must be careful interacting via shared memory to avoid “interrupt windows”
- With a multithreaded OS, this issue is called:
  - “Thread Safety” or
  - “Synchronized Access to Critical Regions”
- Must be handled by design because problems are very hard to detect – never mind fix - in testing
- Causes problems that can not be reproduced

# ISR and Background Code

- Note following sequence in background code:

```
inb         (%dx), %al
orb         $0x01, %al     Interrupt Occurs Here!
outb        %al, (%dx)     ISR returns Here!
```

- With this sequence in ISR code:

```
pushl       %eax
pushl       %edx
inb         (%dx), %al
orb         $0x10, %al
outb        %al, (%dx)
popl        %edx
popl        %eax
iret
```

## ISR and Background Code

- If a sequence of instructions in background must not be interrupted, that software must:
  - "inhibit" interrupts before starting (cli instruction)
  - "enable" interrupts after finishing (sti instruction)
  - (sti and cli instructions set or clear IF in %eflags)
- Must not disable interrupts for very long!!
- This is commonly done in software that initializes an I/O device to operate under interrupt control – preventing an interrupt from occurring prematurely

# ISR and Background Code

- Corrected sequence in background code:

```
cli                         # disable interrupts
inb      (%dx), %al
orb      $0x01, %al
outb     %al, (%dx)
sti                         # reenable interrupts
```

ISR can not execute within this section of code

- Now it does not conflict with this sequence in ISR:

```
…
inb      (%dx), %al
orb      $0x10, %al
outb     %al, (%dx)
…
iret
```

# C Keyword “volatile”

- A similar issue that can arise in C coding for embedded systems is that the compiler may optimize code incorrectly if it is not warned that a variable can change its value unexpectedly
- A shared memory location or memory mapped I/O register may change its value without any compiler generated code causing the change
- Compiled code may read a value into a register and fail to reread it later because it “thinks” that it already has “cached” the value in the register

# C Keyword “volatile”

- To prevent this, the programmer must warn the compiler that this can happen using the keyword “volatile” in the variable declaration
  - Example for ISR/BG shared memory location:

    ```
    volatile int foobar;
    ```

  - Example for pointer to memory mapped I/O register:

    ```
    volatile unsigned char *port;
    ```

- Compiler generated code will always read current value for a “volatile” variable from memory

# Programmable Interval Timer

- This is just an overview – Read data sheet
- 8254 VLSI chip with three 16 bit counters
- Each counter:
  - Is decremented based on its own input clock
  - Is only decremented while its gate is active
  - Generates its own output clock =
    input clock / count length
  - Generates an interrupt when count value reaches zero
  - Automatically reloads initial value when it reaches zero

## PIT Device (Timer 0)

- Simplest device: always is interrupting, every time it down counts to zero
- Can’t disable interrupts in this device!
- Can mask them off in the PIC
- We can control how often it interrupts
- Timer doesn’t keep track of interrupts in progress—just keeps sending them in
- We don’t need to interact with it in the ISR (but we do need to send an EOI to the PIC)

## Use of PIT in MP3

- We use PIT counter 0 with 18.2 Hz output to generate an interrupt every 55 millisecs
- MP3 gives you the boilerplate for the required PIT driver code in tickpack.c.
- You finish the hardware related lines of code:
  - Init must set up and enable PIT interrupts
  - ISR must invoke provided callback function
  - Stop must disable PIT interrupts
- Test with PC-Tutor and use as basis for MP5

## Timer Interrupt Software

- `Initialization`
  - Disallow interrupts in CPU (`cli`)
    - Unmask IRQ0 in the PIC by ensuring bit 0 is 0 in the Interrupt Mask Register accessible via port 0x21
    - Set up interrupt gate descriptor in IDT, using irq0inthand
    - Set up timer downcount to determine tick interval
  - Allow interrupts (`sti`)
- `Shutdown`
  - Disallow interrupts (`cli`)
    - Disallow timer interrupts by masking IRQ0 in the PIC by making bit 0 be 1 in the Mask Register (port 0x21)
  - Allow interrupts (`sti`)

# Timer Interrupts: Interrupt Handler (Two Parts)

- `irq0inthand` – the outer assembly language interrupt handler
  - Save registers
  - Calls C function irq0inthandc
  - Restore registers
  - Iret
- `irq0inthandc` - the C interrupt handler
  - Issues EOI
  - Calls the callback function, or whatever is wanted

# PIT Characteristics

- PIT chip has four I/O ports assigned to it:

  $A_1$ $A_0$

  - Timer 0 assigned port 40 = 0100 0000
  - Timer 1 assigned port 41 = 0100 0001
  - Timer 2 assigned port 42 = 0100 0010
  - Control assigned port 43 = 0100 0011
  - Chip selected by “chip select” and $A_1$-$A_0$
  - Other signals include read, write, and data

# Control Word Format

- Actually only a byte:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **SC1** | **SC0** | **RW1** | **RW0** | **M2** | **M1** | **M0** | **BCD** |

- SC1-SC0 select which counter to write/read
- RW1-RW0 to latch value or select which byte of count value
- M2-M0 determines which operating mode
- BCD specifies whether binary or BCD count
- Command formats found in datasheet

# Custom C Library Symbolic Constants

- Refer to timer.h

```
#define TIMER0_COUNT_PORT 0X40
#define TIMER_CNTRL_PORT 0X43
/* bits 6-7: */
  #define TIMER0 (O<<6)
  #define TIMER1 (1<<6)
/* Bits 4-5 */
  #define TIMER_LATCH (0<<4)
  #define TIMER_SET_ALL (3<<4)
/* Bits 1-3 */
  #define TIMER_MODE_RATEGEN (2<<1)
/* Bit 0 */
  #define TIMER_BINARY_COUNTER 0
```

# Custom C Library Symbolic Constants

- Bits to initialize

```
TIMER0 | TIMER_SET_ALL | TIMER_RATEGEN
  |TIMER_BINARY_COUNTER
```

- Output to the timer I/O port

```
outpt(TIMER_CNTRL_PORT, …);
```

- Then load the downcount

```
outpt(TIMER0_COUNT_PORT, count & 0xFF);
    // LSByte
outpt(TIMER0_COUNT_PORT, count >> 8);
    // MSByte
```

## Custom C Library Functions

- The cpu.h library functions to enable/disable all interrupts in the processor

```
/* do CLI instruction, clear I bit in EFLAGS,
   to disable interrupts in CPU */
void cli(void);
/* do STI instruction, set I bit in EFLAGS,
   to enable interrupts in CPU */
void sti(void);
```

- Samples for Usage

```
cli();      /* disable interrupts */
sti()       /* enable interrupts  */
```

# Custom C Library Functions

- The pic.h library functions to enable/disable PIC

```
/* Command PIC to let signals for a specified IRQ get through to CPU.
   Works for irqs 0-15, except 2, which is reserved for cascading to
   the slave chip. */
void pic_enable_irq(int irq);

/* Command PIC to stop signals on line irq from reaching CPU. */
void pic_disable_irq(int irq);
```

- Examples of Usage for IRQ0 (PIT):

```
#define TIMER0_IRQ 0                      /* defined in timer.h */
pic_enable_irq(TIMER0_IRQ);
pic_disable_irq(TIMER0_IRQ);
```

# Custom C Library Functions

- The cpu.h library function to set idt gate:

```
/* write the nth idt descriptor as an interrupt gate to inthand_addr
  We use an argument of type pointer to IntHandler here so we can
  reestablish a saved interrupt-handler address (such a variable
  would need type pointer-to-function, and would not match a
  parameter type  of IntHandler here--an obscure C gotcha.  */

void set_intr_gate(int n, IntHandler *inthand_addr);
```

- Example of usage for IRQ0 (PIT) interrupt:

```
/* irq 0 maps to slot n = 0x20 in IDT for linux setup */
#define IRQ_TO_INT_N_SHIFT 0x20                /* defined in pic.h */
set_intr_gate(TIMER0_IRQ+IRQ_TO_INT_N_SHIFT, &irq0inthand);
```