Interrupts (Part 2)
Interrupt Sequence

• When the CPU receives an interrupt:
  – Finishes current instruction
  – Pushes CPU registers on stack
  – Disables further interrupts by setting the I bit
  – Identify source of the interrupt – if more than one is pending, figure out which is highest priority
  – Fetch corresponding ISR address from the vector table
  – Start executing the ISR

• When the ISR is done
  – Execute the RTI instruction, which is the last instruction in the ISR
  – This causes CPU registers to be restored from the stack
  – This automatically re-enables interrupts and transfers control back to the main program
Interrupt programming in C

- The C compiler needs to
  - Put the ISR address in the right place in the table
  - End the ISR with the RTI assembly instruction

- You tell the compiler that the function you are declaring is an ISR using the syntax:

  ```c
  void interrupt N yourISRname() {
      (your C code for the ISR goes here)
  }
  ```

- The compiler will
  - Put the address of the ISR into the Nth place in the table
  - End the routine with an “RTI” instruction

ISR functions don’t have any input parameters and they don’t return anything (must have return type “void”)
Example – heartbeat with interrupts

• Program the RTI system to flash an LED at a rate of about 1 per second
  – I.e., turn on for one second, then turn off for 1 sec
  – This is kind of a “heartbeat” indicator by which you can tell your program is running

• The slowest RTI timeout rate is 8/second
  – So we only want to take an action every 8th timeout

• We will have to keep a count of timeouts
  – Every time we sense a timeout, we increment the counter
  – When the counter reaches 8, we reset the counter to zero and toggle the LED
Approach

- Main program
  - Set up PT0 for output (bit 0 in register DDRT)
  - Set up RTI rate (bits RTR6:RTR0 in register RTICTL)
  - Enable RTI interrupts (set RTIE bit in register CRGINT)
  - Turn on interrupt system
  - Go into an infinite loop

```c
void main(void) {
    DDRT = 0x01;    // Set up PT0 for output
    RTICTL =  0x7f; // Set RTI period for the slowest rate (about 1/8 sec)
    CRGINT = 0x80;  // RTIE=1: enable rti interrupts

    EnableInterrupts;  // Turn on interrupt system (clears I bit)

    for(;;) {
        _FEED_COP(); /* feeds the dog */
    } /* loop forever */
}
```
Approach (continued)

- **Interrupt service routine**
  - Clear the flag (write a 1 to the RTIF bit in CRGFLG)
  - Increment count
  - If count is equal to 8, toggle PT0 and clear count

```c
void interrupt VectorNumber_Vrti rti_isr() {
    // "static" causes a local variable to retain its value when the
    // function exits, and not disappear like a normal local variable.
    static int count = 0;

    CRGFLG = 0x80; // clear RTIF flag
    if (++count == 8) {
        PTT = PTT ^ 0x01; // toggle PT0
        count = 0; // reset count back to zero
    }
}
```

This constant is defined as 7 in the include file “mc9s12c32.h”
General structure

• General structure of programs that use interrupts
  
  – Main program
    • Initialize the subsystem(s)
    • Enable interrupts for each of the subsystems
    • Turn on interrupt system (e.g., the CLI instruction)
    • Go into an infinite loop
  
  – Each interrupt service routine (ISR)
    • Clear the flag that caused the interrupt
    • Service the interrupt
    • Execute the RTI instruction

And of course you have to make sure that the ISR addresses get loaded into the vector table in the right places
Summary / Questions

• In C, you define an interrupt service routine (ISR) with the syntax

```c
void interrupt N yourISRname() {
    (your C code for the ISR goes here)
}
```

• What would happen (i.e., what would be the behavior of the program) if you forgot to enable interrupts?

• Why should an ISR be as quick as possible?