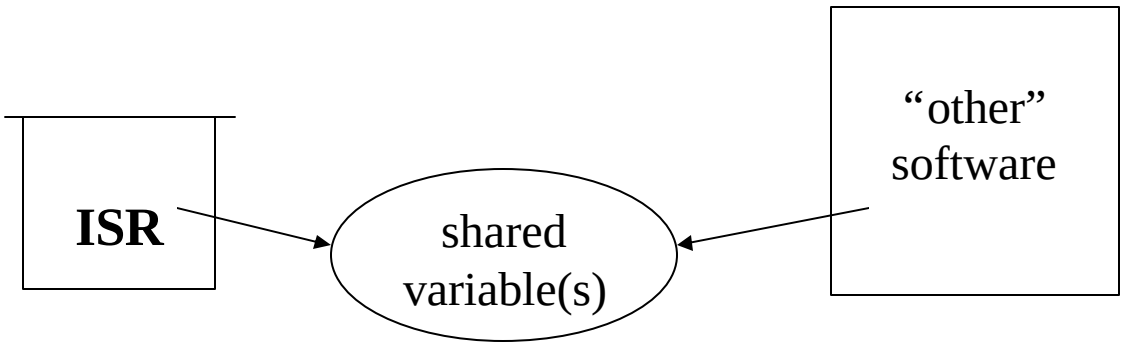
Hardware Interrupts Programming Issues

Hardware Interrupt Programming Issues

Data Flow: exchanging data with an ISR

- parameters to hardware ISR?
  - **NO !** – interrupt caused by hardware
  - not a software call !
- **shared** (persistent) state **variables** → the only way!



INTERFERENCE

- ISR **interferes** with access!

PROBLEM: when accessing variable shared with an ISR – is it possible (?) to have the:

- ISR interrupt in the middle of the access
- ISR modify the variable
- **net result**: the value of the variable is corrupted

Interference Example:

Timer scenario: shared 32-bit count variable

- main: wants to read count: reads **count_high**
- ISR interrupt main
- modifies both **count_high** and **count_low**
- ISR terminates, main continues: reads **count_low**
- **net result**: main obtains a **corrupted count value** !

Critical Region: *(terminology)*

a section of code that has the **potential** for interference

- existence of a critical region does not guarantee that interference will occur
- just provides the opportunity! (**potential** for…)
- occurrence of interference depends on implementation and requires **interruption of a critical region**, and execution of another critical region that interferes
  - **event-driven thinking!**

- **interference scenario** !

Protecting Critical Regions

- must be able to **recognize critical regions**
- must design protection into solutions

Software Protection of Critical Regions:

- prevent the interruption of critical regions

  → eliminate the occurrence of interference!
- software has limited ability to control occurrence of interrupts:
  - can prevent them by masking interrupts

- masking interrupts at **processor**: CLI / STI
- prevents all interrupts ← overkill? ☹
- keep critical regions short ! ☺

Protecting Critical Regions

- masking interrupts at **PIC**: mask register
- **remember**: PIC shared resource too ! Selectively prevent interrupts that might interfere in with particular critical region, but still let other interrupts occur

- masking interrupts at **I/O components**:
- same approach as PIC, but does not involve modifying PIC

when **interrupts enabled & unmasked**:

- no software control
- interrupts are generated by asynchronous hardware !

Protecting Critical Regions

Installing ISR ← a critical region!

- modifying vector table
- vector table is a shared resource !

ISRs & saving registers:

- ISR ***MUST*** save and restore all registers used
- interrupted program does not know the interrupt is happening
  → cannot "save" registers before interrupt

ISRs & EOI

- ISR ***MUST*** send EOI to PIC
- failure to send EOI (or sending more than one EOI) will cause undesirable behaviour

Enabling Interrupts

- **event-driven mindset**: want interrupts to be generated, and ISRs to be executed
- if IF = **0** → NO interrupts will occur
- IRET is executed by hardware ISR, IF will be reset to **1** when FLAGS are popped
- lower (or equal) priority interrupts will not be sent by PIC until EOI is received

Design Questions:

- When should an ISR set IF = 1 ?
- When should an ISR send EOI to PIC ?

Re-Entrant Code

- suppose a code fragment is interrupted, and ISR executes the same code fragment
- code fragment has been "re-entered" by execution thread of control, **BEFORE** interrupted thread of control had completed the fragment
- might happen if two threads of control (perhaps main program and an ISR) call the same function

Re-entrant code: **notorious** source of **critical regions**!

- **all persistent variables** accessed by the code are **implicitly shared**! ☹
- local variables – persist only for duration of a function invocation? ← exist in stack?

Some Common ISR Programming Errors:

1) forgetting to save/restore **ALL** of the registers altered by ISR

2) forgetting to initialize **DS** register (or ES for segment overrides) prior to accessing any variables

3) forgetting to send an **EOI** command to the PIC.

4) forgetting to enable the interrupts ☺

STI and / or @ **PIC** and / or @ **device**

5) interference

Buffered I/O

- interrupts allow devices to be serviced independent of other software activity
- want to minimize interactions between ISR and other software
- interactions must be synchronized
- potential interference
- more interactions = more frequent checking
- tighter coupling
- too slow → lost/over-written data
- e.g. shared variable in keyboard example

- interaction constraints can often be relaxed using **buffering**
- increase the capacity of shared variables

Keyboard example

- suppose keyboard ISR has single character buffer
  - other s/w must poll variable
- polling not carried out “fast enough”
- ISR over-writes before other software reads

- suppose keyboard ISR has multiple character buffer: i.e., FIFO queue
- Kybd ISR adds to end of queue as scan codes arrive
- other s/w reads from head of queue
- need to protect access to queue ← shared !
- less risk of losing data
- most operating systems have internal keyboard queue
- e.g. DOS – 8 char’s
- decouples application execution speed from speed of keystrokes arriving (relaxes interaction constraints)