

Last Time

- ◆ **Cost of nearly full resources**
- ◆ **RAM is limited**
 - Think carefully about whether you use a heap
 - Look carefully for stack overflow
 - Especially when you have multiple threads
- ◆ **Embedded C**
 - Extensions for device access, address spaces, saturating operations, fixed point arithmetic

Today

◆ **Advanced interrupts**

- **Race conditions**
- **System design**
- **Prioritized interrupts**
- **Interrupt latency**
- **Interrupt problems:**
 - **Stack overflow**
 - **Overload**
 - **Missed interrupts**
 - **Spurious interrupts**

Typical Interrupt Subsystem

- ◆ **Each interrupt has a *pending* bit**
 - **Logic independent of the processor core sets these bits**
 - E.g. ADC ready, timer expires, edge detected, etc.
 - **A pending bit can become set at any time**
 - This logic does not need to be synchronized with the MCU
- ◆ **Each interrupt has a *disable* bit**
- ◆ **Processor has a *global disable* bit**

More Interrupt Basics

◆ Interrupt algorithm

- If global interrupt enable bit is set, processor checks for pending interrupts prior to fetching a new instruction
- If any interrupts are pending, highest priority interrupt that is pending and enabled is selected for execution
- If an interrupt can be fired, flush the pipeline and jump to the interrupt's handler

◆ Some interrupts must be *acknowledged*

- This clears the pending flag
- Failure to do this results in infinite interrupt loop
 - Symptom: System hangs

Interrupts on ColdFire

- ◆ **Interrupt controller on MCF52233 is fairly sophisticated**
 - Many MCUs have much simpler controllers
- ◆ **Processor has a 3-bit interrupt mask in SR**
- ◆ **Once per instruction, processor looks for pending interrupts with priority greater than the mask value**
 - However, level 7 interrupts are non-maskable

ColdFire Interrupt Sequence

1. **CPU enters supervisor mode**
2. **8-bit vector fetched from interrupt controller**
3. **Vector is an index into the 256-entry exception vector table**
 - ◆ **Vector entries are 32-bit addresses**
 - ◆ **Vectors 0-63 are reserved, you can use 64-255**
4. **Push SR and PC**
5. **Load vector address into PC**
6. **Set interrupt mask to level of current interrupt**
7. **First instruction of interrupt handler is guaranteed to be executed**
 - ◆ **So this would be a good place to disable interrupts, if you don't want nested interrupts**

More ColdFire

- ◆ **Within an interrupt level, there are 9 priorities**
- ◆ **Interrupt controller has registers that permit you to assign level and priority to interrupt sources**
 - **In contrast, many embedded processors fix priorities at design time**
- ◆ **Many ColdFire processors (including ours) support two stack pointers**
 - **User mode and supervisor mode**
 - **We're only using the supervisor stack pointer**

Interrupts and Race Conditions

- ◆ **Major problem with interrupts:**
 - They cause *interleaving* (threads do too)
 - Interleaving is hard to think about
- ◆ **First rule of writing correct interrupt-driven code**
 - Disable interrupts at all times when interrupt cannot be handled properly
 - Easier said than done – interrupt-driven code is notoriously hard to get right
- ◆ **When can an interrupt not be handled properly?**
 - When manipulating data that the interrupt handler touches
 - When not expecting the interrupt to fire
 - Etc.

Interleaving is Tricky

```
interrupt_3 { ... does something with x ... }
```

```
main () {  
    ...  
    x += 1;  
    ...  
}
```

- ◆ Do you want to disable interrupts while incrementing x in main()?
- ◆ How to go about deciding this in general?

◆ **Using our compiler**

```
x += 1;
```

◆ **Translates to:**

```
addq .1    #1, _x
```

◆ **Do we need to disable interrupts to execute this code?**

◆ However:

```
x += 500;
```

◆ Translates to:

```
movea.l  _x, a0  
lea     500(a0), a0  
move.l  a0, _x
```

- ◆ **The property that matters here is atomicity**
 - **An atomic action is one that cannot be interrupted**
- ◆ **Individual instructions are usually atomic**
- ◆ **Disabling interrupts is a common way to execute a block of instructions atomically**

- ◆ **Question: Do we really need atomicity?**

- ◆ **Answer: No– we need code to execute “as if” it executed atomically**
- ◆ **In practice, this means: Only exclude computations that matter**
- ◆ **Example 1: Only raise the interrupt level high enough that all interrupts that can actually interfere are disabled**
- ◆ **Example 2: Thread locks only prevent other threads from acquiring the same lock**
- ◆ **Example 3: Non-maskable interrupts cannot be masked**

- ◆ **Summary: Each piece of code in a system must include protection against**
 - **Threads**
 - **Interrupts**
 - **Activities on other processors**
 - **DMA transfers**
 - **Etc.**
- ◆ **that might cause incorrect execution by preempting the code you are writing**

Reentrant Code

- ◆ A function is *reentrant* if it works when called by multiple interrupt handlers (or by main + one interrupt handler) at the same time
- ◆ What if non-reentrant code is reentered?
- ◆ Strategies for reentrancy:
 - Put all data into stack variables
 - Why does this work?
 - Disable interrupts when touching global variables
- ◆ In practice writing reentrant code is easy
 - The real problem is not realizing that a transitive call-chain reaches some non-reentrant call
 - A function is non-reentrant if it can possibly call any non-reentrant function

System-Level Interrupt Design

◆ Easy way:

- Interrupts never permitted to preempt each other
- Interrupts permitted to run for a long time
- Main loop disables interrupts liberally

◆ Hard way:

- Interrupts prioritized – high priority can always preempt lower priority
- Interrupts not permitted to run for long
- Main loop disables interrupts with fine granularity
- Pros and cons?

◆ Stupid way:

- Any interrupt can preempt any other interrupt
- ColdFire doesn't let you do this!
 - But other processors do

Interrupt Latency

- ◆ ***Interrupt latency*** is time between interrupt line being asserted and time at which first instruction of handler runs
- ◆ **Two latencies of interest:**
 - Expected latency
 - Worst-case latency
 - How to compute these?
- ◆ **Sources of latency:**
 - Slow instructions
 - Code running with interrupts disabled
 - Other interrupt handlers

Managing Interrupt Latency

- ◆ **This is hard!**
- ◆ **Some strategies**
 - **Nested interrupt handlers**
 - **Prioritized interrupts**
 - **Short critical sections**
 - **Split interrupts**
- ◆ **Basic idea: Low-priority code must not block time-critical interrupts for long**

Nested Interrupts

- ◆ Interrupts are *nested* if multiple interrupts may be handled concurrently
- ◆ Makes system more responsive but harder to develop and validate
 - Often much harder!
- ◆ Only makes sense in combination with prioritized interrupt scheduling
 - Nesting w/o prioritization increases latency without increasing responsiveness!
- ◆ Nested interrupts on ColdFire are easy
 - Just don't disable interrupts in your interrupt handler
- ◆ Some ARM processors make this really difficult

Prioritizing Interrupts

- ◆ **Really easy on some hardware**
 - **E.g. x86 and ColdFire automatically mask all interrupts of same or lower priority**
- ◆ **On other hardware not so easy**
 - **E.g. on ARM and AVR need to manually mask out lower priority interrupts before reenabling interrupts**
 - **Argh.**

Reentrant Interrupts

- ◆ **A reentrant interrupt may have multiple invocations on the stack at once**
 - **99.9% of the time this is a bug**
 - **Programmer didn't realize consequences of reenabling interrupts**
 - **Programmer recognized possibility and either ignored it or thought it was a good idea**
 - **0.1% of the time reentrant interrupts make sense**
 - **E.g. AvrX timer interrupt**

- ◆ **Does ColdFire support reentrant interrupts?**

Missed Interrupts

- ◆ **Interrupts are not queued**
 - Pending flag is a single bit
 - If interrupt is signaled when already pending, the new interrupt request is dropped
- ◆ **Consequences for developers**
 - Keep interrupts short
 - Minimizes probability of missed interrupts
 - Interrupt handlers should perform all work pending at the device
 - Compensates for missed interrupts

Splitting Interrupt Handlers

- ◆ **Two options when handling an interrupt requires a lot of work:**
 1. **Run all work in the handler**
 2. **Make the handler fast and run the rest in a deferred context**
- ◆ **Splitting interrupts is tricky**
 - **State must be passed by hand**
 - **The two parts become concurrent**
- ◆ **There are many ways to run the deferred work**
 - **Background loop polls for work**
 - **Wake a thread to do the work**
 - **Windows has deferred procedure calls, Linux has tasklets and bottom-half handlers**

Spurious Interrupts

- ◆ **Glitches can cause interrupts for nonexistent devices to fire**
 - **Processor manual talks about these**
- ◆ **Solutions:**
 - **Have a default interrupt handler that either ignores the spurious interrupt or resets the system**
 - **Ensure that all nonexistent interrupts are permanently disabled**

Interrupt Overload

- ◆ **If an external interrupt fires too frequently**
 - **Lower-priority interrupts starved**
 - **Background loop starved**
- ◆ **Why would this happen?**
 - **Loose or damaged connection**
 - **Electrical noise**
 - **Malicious or buggy node on network**
- ◆ **Apollo 11**
 - **Computer reset multiple times while attempting to land on moon**
 - **LLM guidance computer overwhelmed by phantom radar data**
 - **Ground control almost aborted the landing**

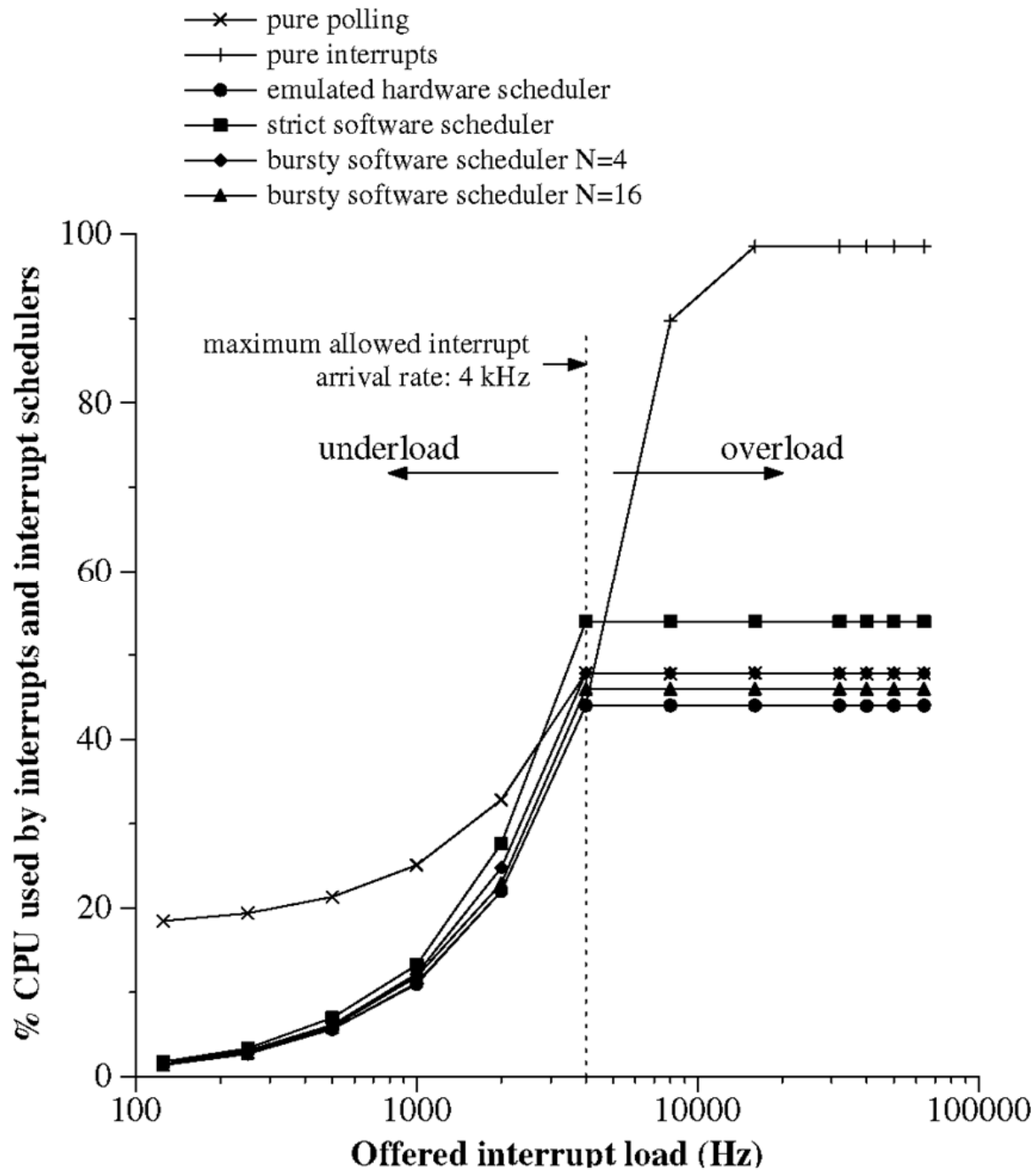
Potential Overload Sources

<i>Source</i>	<i>Max. Interrupt Freq. (Hz)</i>
knife switch bounce	333
loose wire	500
toggle switch bounce	1 000
rocker switch bounce	1 300
serial port @115 kbps	11 500
10 Mbps Ethernet	14 880
CAN bus	15 000
I2C bus	50 000
USB	90 000
100 Mbps Ethernet	148 800
Gigabit Ethernet	1 488 000

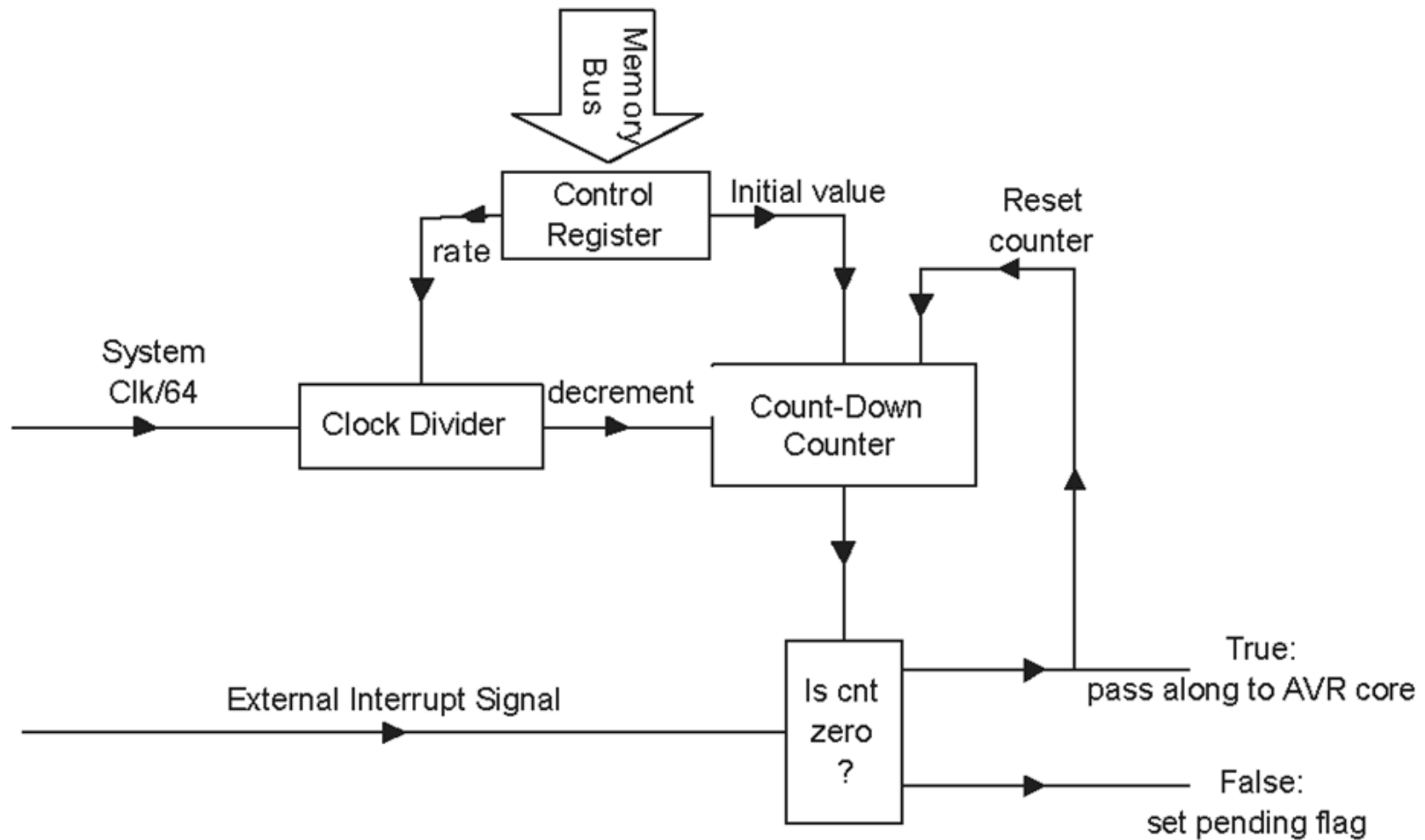
Preventing Interrupt Overload

◆ Strategies:

- Trust the hardware not to overload
- Don't use interrupts – poll
- Design the software to prevent interrupt overload
- Design the hardware to prevent interrupt overload



Hardware Interrupt Scheduler



Interrupt Pros

◆ Support very efficient systems

- No polling – CPU only spends cycles processing work when there is work to do
- Interrupts rapidly wake up a sleeping processor

◆ Support very responsive systems

- Well-designed and well-implemented software can respond to interrupts within microseconds

Interrupt Cons

- ◆ **Introduce hard problems:**
 - **Concurrency and reentrance**
 - **Missed interrupts**
 - **Spurious interrupts**
 - **Interrupt overload**
- ◆ **Make stack overflow harder to deal with**
- ◆ **Interrupt-driven codes hard to test adequately**
- ◆ **Achieving fast worst-case response times is difficult**
- ◆ **Overall:**
 - **Few safety-critical systems are interrupt-driven!**

Summary

- ◆ **Interrupts are very convenient**
 - **But a huge can of worms**
- ◆ **Interrupt handling is a whole-system design issue**
 - **Can't just handle each interrupt individually**
- ◆ **Never write code that allows reentrant interrupts**