Introduction to Embedded Systems

CS/ECE 6780/5780

Al Davis

Today's topics:

- lab logistics
- interrupt synchronization
- reentrant code

Lab Logistics

- Lab2 Status
  - Wed: 3/11 teams have completed their labs
    - » likely due to late handout of Lab 2
      - we were 3 days late in getting it to you
      - only fair to give you 3 days extension
        - make appointment w/ William to demonstrate
        - lab reports still due at next weeks lab session
      - BUT other labs will stay on schedule
  - Pre-labs must be done before your lab session
    - » most of the Wed. non-finishers didn't do this
    - » bottom line
      - you won't get pre-lab and actual lab done in time if you don't do the pre-lab prior to your lab session
  - Don't just blow off any lab
    - » labs are additive
      - e.g. previous lab code will be useful for subsequent labs
- Schedule through first midterm is on the web
  - useful to do the reading before the lecture

Interrupts

- External wake-up stimulus
  - alternative to the software polling loop
- Efficient response to rare events
  - importance will vary
    - » low-power warning → emergency
    - » key got pushed → handle soon but not now
  - energy efficient
    - » processor doesn't consume energy in a polling loop
    - » or it can be doing something else that is useful
- Highly useful for data acquisition and control
- Predictable response time?
  - depends on how you configure the system
  - predictability is a MUST in real-time systems
- Interrupts add concurrency to your ES software
  - this can make you life very difficult

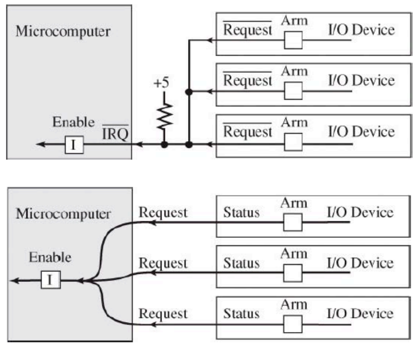
What are Interrupts

- Automatic hardware supported transfer of control
  - external hardware runs asynchronously & concurrently
    - » w.r.t. controller SW
  - interrupts transfer control out of whatever is currently running
    - » to an interrupt service routine (ISR)
      - looks like a surprise call to the ISR that you write
      - ISR is a *background* thread
    - » ISR return is done via the `rti` instruction
    - » interrupts communicate with `main` using shared memory
      - concurrency usually requires some form of mutual exclusion control
        - such as semaphores
- Software interrupts
  - these are synchronous
    - » SWI instruction (one use would be a breakpoint)
    - » automatically happens on 6812 via an unimplemented opcode
      - e.g. ISR may implement the opcode or send command to off chip implementation

ARM vs. Disable

- **ARM**
  - **each off-controller "interrupter"**
    - » should have a register that has an ARM bit
    - » allows the controller to specify who may interrupt
- **Disable**
  - **the I bit in the CC register**
    - » I=0 → all armed interrupts are enabled
    - » I=1 → disable interrupts
      - note this does not ignore interrupts
      - it just postpones them
  - **typical disable usage**
    - » first action in ISR – set I=1
    - » do whatever needs to be done that can't be interrupted
    - » set I=0
    - » NOTE: not doing this is a common source of significant confusion

Dedicated vs. Shared



Multiple open-collector requests tied together to a shared interrupt line on the controller Pull up holds IRQ' high when no service is requested

Separate interrupt inputs on the controller Each I_{in} is dedicated to a particular request

6812 Interrupt Inputs

- **IRQ'**
  - **can be disabled or enabled via the I bit in the CCR**
  - **typical use is for non-emergency interrupts**
    - » e.g. service can be handled and then resume normal control
- **XIRQ'**
  - **once enabled by setting the X bit in the CCR to 0**
    - » they can't be disabled
  - **typical use is emergency**
    - » low-power
      - ISR saves as much as it can before power is gone
        - – return to normal control in this case won't happen
- **How does the controller know what to do in the shared IRQ case?**
- **What are the fundamental advantages of shared vs. dedicated interrupts?**

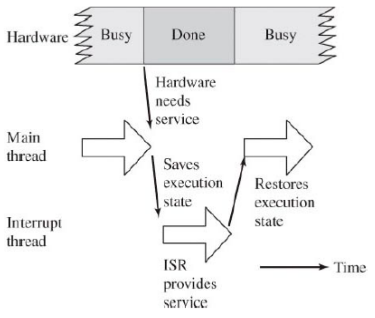
Dedicated vs. Shared

- **Shared**
  - **+'s use of wired-or style interrupts (open collector on 6812)**
    - » no limit on number of interrupt capable devices
    - » simple hardware and wiring
    - » expansion to include more I/O devices requires no redesign
  - **cons:**
    - » ISR must look at all possible device registers to see what service is required
      - there may be more than one
    - » dispatch to appropriate routine
    - » reset the request bit
- **Dedicated**
  - **+'s: simpler software due to dedicated ISR's**
    - » faster since SW doesn't need to check everybody
    - » less software coupling
    - » easier to implement priority
  - **con: you might run out of dedicated I_{in}'s**

ISR's

- **ISR: software that handles interrupt requests**
- **2 polar styles**
  - **Polled – one large ISR handles all requests**
    - » **shared model**
  - **Vectored**
    - » **dedicated model**
      - **each I_{in} causes indirect jump via a specific memory location**
      - **often indexed via a separate interrupt index register**
- **If armed and enabled and a request happens**
  - **execution of main program is suspended**
  - **all registers pushed to stack**
  - **ISR executed and returns (rti)**
  - **all registers restored from stack**
  - **main program is resumed**
  - **see any problems?**

Interrupt Execution



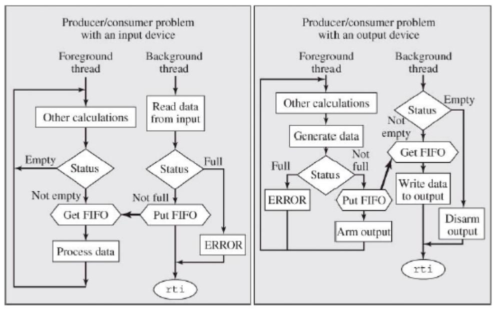
When to Use Interrupts

Gadfly	Interrupts	DMA
Predictable	Variable arrival times	Low latency
Simple I/O	Complex I/O	High bandwidth
Fixed load	Variable load	
No concurrency	Concurrent execution	
Nothing else to do	Infrequent alarms	
	Program errors	
	Overflow, illegal op	
	Illegal memory access	
	Machine/memory errors	
	Power failure	
	Real-time clocks	
	Data acquisition/control	

Communication w/ Interrupts

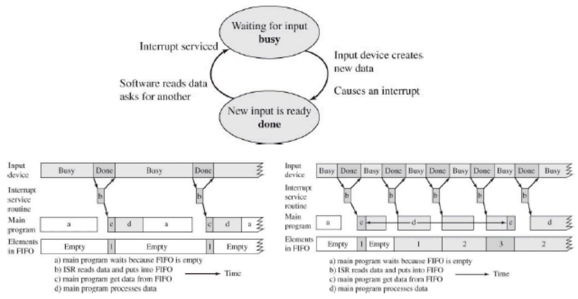
- **Any 2 threads must communicate via global memory**
  - **foreground and background threads in the interrupt case**
    - » **but still they are 2 threads**
  - **in some cases main and ISR won't communicate**
    - » **this is common in mainstream CPU's**
    - » **e.g. ISR's fix exceptional events that have nothing to do with main**
  - **ES's tend to be different**
    - » **tight coupling between I/O environment**
    - » **controllers role is often to aggregate I/O interrupt events into it's main calculation**
- **Interrupts have logically separate registers/stack**
  - **so communication must occur through global memory**
    - » **note that logically doesn't mean physically**
      - **registers saved and restored via the same physical stack**
    - » **communication can't use the stack since it's not a normal subroutine call due to the surprise factor**

2 Thread Communication Example

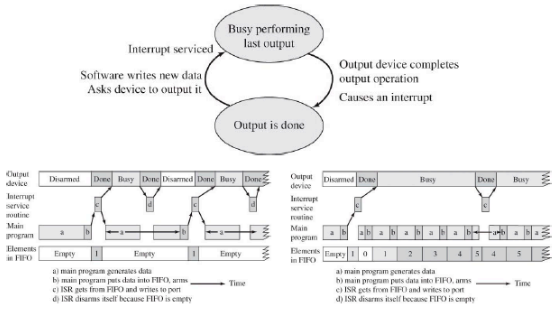


Global FIFO is a common choice - why?

Input Device Interrupts


Waiting for input busy
Interrupt serviced
Input device creates new data
Software reads data asks for another
Causes an interrupt
New input is ready done
Input device
Interrupt service routine
Main program
Elements in FIFO
Busy
Done
Empty
Time
a) main program waits because FIFO is empty
b) ISR reads data and puts into FIFO
c) main program get data from FIFO
d) main program processes data

Output Device Interrupts


Busy performing last output
Interrupt serviced
Software writes new data Asks device to output it
Output device completes output operation
Causes an interrupt
Output is done
Output device
Interrupt service routine
Main program
Elements in FIFO
Disarmed
Done
Busy
Empty
Time
a) main program generates data
b) main program puts data into FIFO, arms
c) ISR gets from FIFO and writes to port
d) ISR disarms itself because FIFO is empty

Other Interrupt Issues

- **Periodic interrupts**
  - **essential for implementing data acquisition and control systems**
- **ISR goals**
  - **occur only when needed**
  - **come in clean, perform function, return right away**
    - » **clearly Gadfly loops should be avoided**
  - **minimize time spent in ISR's**
- **Latency**
  - **interface latency**
    - » **time between new data interrupt and when ISR or main gets the data**
      - **beware the difference**
    - » **device latency – response time of external I/O device**
  - **real-time?**
    - » **requires tight bounds on latency guarantees**

Reentrant Programming

- **Program reentrant if**
  - can be XEQ's by >1 threads of control
- **Program must**
  - place local variables on the stack or use some form of mutual exclusion to prevent conflict on global storage accesses
- **Non-reentrant subroutine**
  - has a vulnerable window
  - error occurs if
    - » main or an ISR calls while running in the vulnerable window
  - fix:
    - » make sure only one active call can be in the vulnerable window
    - » mutex mechanisms
      - semaphore variable
      - disable interrupts

Reentrant or Not?

- **Must be able to recognize potential sources of bugs**
  - due to non-reentrant code in high-level languages
    - » just for yucks we'll define C to be high level
- **Another example of why you'll sometimes need to examine assembly code**
  - Is `time++` atomic?
    - » Yes if compiler generates
      - `inc time`
    - » No if compiler generates
      - `ldd time`
      - `add #1`
      - `std time`

What is the essential difference?

Atomic Operations

- **Atomic operation**
  - an operation that once started is guaranteed to finish
- **In most machines**
  - a single assembly instruction is atomic
    - » if an interrupt happens it will be taken between instructions
    - » not during
- **Hence**

The following is atomic:

```
inc counter     where counter is global variable
```

The following is *nonatomic*:

```
ldaa counter    where counter is global variable
inca
staa counter
```

Read-Modify-Write Example

- **Typical case where atomicity is desired**
  - but not guaranteed ===➔ bug time BIG TIME

Software reads global variable, producing a copy of the data.
Software modifies the copy.
Software writes modification back into global variable.

```
unsigned int Money;  /* bank balance (global) */
/* add 100 dollars */
void more(void){
     Money += 100;}

Money rmb  2      bank balance implemented as a global
* add 100 dollars to the account
more  ldd  Money  where Money is a global variable
      addd #100
      std  Money  Money=Money+100
      rts
```

Write followed by Read Example

- **RAW hazard**

Software writes to a global variable.
Software reads from global variable expecting original data.

```
int temp;  /* global temporary */
/* calculate x+2*d */
int mac(int x, int d){
       temp = x+2*d;     /* write to a global variable */
       return (temp);}  /* read from global */

temp  rmb  2      global temporary result
* calculate RegX=RegX+2*RegD
mac   stx  temp   Save X so that it can be added
      lsld        RegD=2*RegD
      addd temp   RegD=RegX+2*RegD
      xgdx        RegX=RegX+2*RegD
      rts
```

Nonatomic Multistep Write

Software write part of new value to a global variable.
Software write rest of new value to a global variable.

```
int info[2];  /* 32-bit global */
void set(int x, int y){
    info[0]=x;
    info[1]=y;}

Info  rmb  4      32-bit data implemented as a global
* set the variable using RegX and RegY
set   stx  Info   Info is a 32 bit global variable
      sty  Info+2
      rts
```

Disabling Interrupts in C

```
int Empty;   /* -1 means empty, 0 means it contains somethi
int Message; /*  data to be communicated */
int SEND(int data){ int OK;
   char SaveSP;
   asm tpa
   asm staa SaveSP
   asm sei     /* make atomic, entering critical */
   OK=0;         /* Assume it is not OK */
   if(Empty){
       Message=data;
       Empty=0;  /*  signify it is now contains a message*/
       OK=-1;}   /* Successfull */
   asm ldaa SaveSP
   asm tap     /* end critical section */
   return(OK);}
```

A Binary Semaphore

```
* Global parameter: Semi4 is the mem loc to test and set
* If the location is zero, it will set it (make it -1)
*     and return Reg CC (Z bit) is 1 for OK
* If location is nonzero, return Reg CC (Z bit) = 0
Semi4 fcb  0        Semaphore is initially free
Tas   tst  Semi4    check if already set
      bne  Out      busy, operation failed, return Z=0
      dec  Semi4    signify it is now busy
      bita #0       operation successful, return Z=1
Out   rts
```

Interrupt Synchronization Summary

- Device synchronization requires some protocol
  - to permit HW and SW FSM's to interact
  - HW protocols found in the device specifications
    - » often cast in concrete
    - » SW protocol must be compatible
- Interrupts are essentially a state machine
  - interaction between main and the ISR follows a protocol
    - » supported by hardware
  - However
    - » since you're writing both main and the ISR
      - you'll be tempted to not think it through carefully
    - » this is why interrupts are hard
      - most of us make this mistake on the initial try or two
      - fortunately mistake enhanced learning works quite well