Interrupts

- Fundamental concept in computation
- Interrupt execution of a program to "handle" an event
  - Don’t have to rely on program relinquishing control
  - Can code program without worrying about others
- Issues
  - What can interrupt and when?
  - Where is the code that knows what to do?
  - How long does it take to handle interruption?
  - Can an interruption be, in turn, interrupted?
  - How does the interrupt handling code communicate its results?
  - How is data shared between interrupt handlers and programs?

What is an Interrupt?

- Reaction to something in I/O (human, comm link)
- Usually asynchronous to processor activities
- "interrupt handler" or "interrupt service routine" (ISR) invoked to take care of condition causing interrupt
  - Change value of internal variable (count)
  - Read a data value (sensor, receive)
  - Write a data value (actuator, send)

Issues

- What can interrupt and when?
- Where is the code that knows what to do?
- How long does it take to handle interruption?
- Can an interruption be, in turn, interrupted?
- How does the interrupt handling code communicate its results?
- How is data shared between interrupt handlers and programs?

Saving and Restoring Context

- Processor and compiler dependent
- Where to find ISR code?
  - Different interrupts have separate ISRs
- Who does dispatching?
  - Direct
    - Different address for each interrupt type
    - Supported directly by processor architecture
  - Indirect
    - One top-level ISR
    - Switch statement on interrupt type
  - A mix of these two extremes?

How much context to save?

- Registers, flags, program counter, etc.
- Save all or part?
- Agreement needed between ISR and program
- Where should it be saved?
  - Stack, special memory locations, shadow registers, etc.
  - How much room will be needed on the stack?
  - Nested interrupts may make stack reach its limit – what then?
- Restore context when ISR completes

Ignoring Interrupts

- Can interrupts be ignored?
  - It depends on the cause of the interrupt
  - No, for nuclear power plant temperature warning
  - Yes, for keypad on cell phone (human timescale is long)
- When servicing another interrupt
  - Ignore others until done
  - Can’t take too long – keep ISRs as short as possible
    - Just do a quick count, or read, or write – not a long computation
- Interrupt disabling
  - Will ignored interrupt "stick"?
    - Rising edge sets a flip-flop
  - Or will it be gone when you get to it?
    - Level changes again and its as if it never happened
    - Don’t forget to re-enable
Prioritizing Interrupts

- When multiple interrupts happen simultaneously
  - Which is serviced first?
  - Fixed or flexible priority?
- Priority interrupts
  - Higher priority can interrupt
  - Lower priority can’t
- Maskable interrupts
  - “don’t bother me with that right now”
  - Not all interrupts are maskable, some are non-maskable

Interrupts in the ATmega16

- External interrupts
  - From I/O pins of microcontroller
- Internal interrupts
  - Timers
    - Output compare
    - Input capture
    - Overflow
  - Communication units
    - Receiving something
    - Done sending
  - ADC
    - Completed conversion

Interrupt Jump Vector Table

- Fixed location in memory to find first instruction for each type of interrupt
- Only room for one instruction
  - JMP to location of complete ISR

Chain of Events on Interrupt

- Finish executing current instruction
- Disable all interrupts
- Push program counter on to stack
- Jump to interrupt vector table
- Jump to start of complete ISR
- Save any context that ISR may otherwise change
  - Registers and flags must be saved within ISR and restored before it returns - this is very important!
- Re-enable interrupts if nested interrupts are ok
- Complete ISR’s code
- Re-enable interrupts upon return
- Jump back to next instruction before interruption

Shared Data Problem

- When you use interrupts you create the opportunity for multiple sections of code to update a variable.
- This might cause a problems in your logic if an interrupt updates a variable between two lines of code that are directly dependent on each other (e.g. if statement)
- One solution is to create critical sections where you disable the interrupts for a short period of time while you complete your logic on the shared variable
  - cli();
  - ....critical section code goes here.....
  - sei();

External Interrupts

- Special pins: INT0, INT1, INT2
  - Can interrupt on edge or level
  - Can interrupt even if set to be output pins
  - Implements “software interrupts” by setting output

Interrupt Vector Table

- Address 0: 00h
  - PIR0: INT 0
  - PIR1: INT 1
  - PIR2: INT 2
- Address 2: 02h
  - PSW: Status Word

Shared Data Problem

- When you use interrupts you create the opportunity for multiple sections of code to update a variable.
- This might cause a problems in your logic if an interrupt updates a variable between two lines of code that are directly dependent on each other (e.g. if statement)
- One solution is to create critical sections where you disable the interrupts for a short period of time while you complete your logic on the shared variable
  - cli();
  - ....critical section code goes here.....
  - sei();
Closer Look at a Timer/Counter

- Timer0/Counter0
  - Clear timer on compare match (auto reload)
  - Prescaler (divide clock by up to 1024)
  - Overflow and compare match interrupts
  - Registers
    - Configuration
    - Count value
    - Output compare value

Timer/Counter Registers

- Timer/Counter Control Register TCCR0

Timer/Counter Registers (cont’d)

- Timer/Counter Control Register TCCR0

Timer/Counter Registers (cont’d)

Setting Register Values

- Defined names for each register and bit
  - Set timer to clear on match
  - Set prescaler to 1024
    \[ TCCR0 = (1<<WGM01) \mid (1<<CS02) \mid (1<<CS00); \]
  - Set count value to compare against
    \[ OCR0 = 150; \]
  - Set timer to interrupt when it reaches count
    \[ TIMSK = (1<<OCIE0); \]
Writing an Interrupt Handler in C

- Set and clear interrupt enable
  - sei();
  - clki();
- Interrupt handler
  - SIGNAL(SIG_OUTPUT_COMPARE0)
    - i++;
- Setting I/O registers
  - TCCR0 = (1<<WGM01) | (1<<CS02) | (1<<CS00);
- Enabling specific interrupts
  - TIMSK = (1<<OCIE0);

Analog to digital conversion

- Use charge-redistribution technique
  - no sample and hold circuitry needed
  - even with perfect circuits quantization error occurs
- Basic capacitors
  - sum parallel capacitance

Analog to digital conversion (cont’d)

- Two reference voltage
  - mark bottom and top end of range of analog values that can be converted (V_L and V_H)
  - voltage to convert must be within these bounds (V_X)
- Successive approximation
  - most approaches to A/D conversion are based on this
  - 8 to 16 bits of accuracy
- Approach
  - sample value
  - hold it so it doesn’t change
  - successively approximate
  - report closest match

A-to-D – sample

- During the sample time the top plate of all capacitors is switched to reference low V_L
- Bottom plate is set to unknown analog input V_X
- Q = CV
- Q_H = 16 (V_X - V_L)
- conservation of charge Q_S = Q_H
- 16 (V_X - V_L) = 16 (V_L - V_I)
- V_X - V_L = V_L - V_I

A-to-D – hold

- Hold state using logically controlled analog switches
  - Top plates disconnected from V_L
  - Bottom plates switched from V_C to V_L
- Q_H = 16 (V_L - V_X)
  - conservation of charge Q_S = Q_H
  - 16 (V_X - V_L) = 16 (V_L - V_X)
  - V_L - V_X = V_C - V_L

A-to-D – successive approximation

- Each capacitor successively switched from V_L to V_H
  - Largest capacitor corresponds to MSB
  - Output of comparator determines bottom plate voltage of cap
    - > 0 : remain connected to V_H
    - < 0 : return to V_L
Suppose \( V_x = \frac{21}{32} \) (\( V_H - V_L \)) and already sampled

- Compare after shifting half of capacitance to \( V_H \)
  - \( V_I \) goes up by \( \frac{8}{16} (V_H - V_L) \)
  - original \( V_L - V_I \) goes down and becomes
    - \( V_L - \left( V_I + \frac{5}{8} (V_H - V_L) \right) = V_L - V_I - \frac{5}{8} (V_H - V_L) \)
- Output > 0

\[ V_H \quad V_L \quad V_I (\text{next}) \]

\[ \frac{5}{8} (V_H - V_L) \]

\[ V_L \quad V_I (\text{prev}) \]

\[ V_L \quad V_I (\text{next}) \]

Compare after shifting another part of cap. to \( V_H \)
- \( V_I \) goes up by \( \frac{4}{16} (V_H - V_L) \)
- original \( V_L - V_I \) goes down and becomes
  - \( V_L - (V_I + \frac{1}{4} (V_H - V_L)) = V_L - V_I - \frac{1}{4} (V_H - V_L) \)
- Output < 0 (went too far)

\[ V_H \quad V_L \quad V_I (\text{prev}) \]

\[ \frac{1}{4} (V_H - V_L) \]

\[ V_L \quad V_I (\text{prev}) \]

\[ \frac{1}{4} (V_H - V_L) \]

Compare after shifting another part of cap. to \( V_H \)
- \( V_I \) goes up by \( \frac{2}{16} (V_H - V_L) \)
- original \( V_L - V_I \) goes down and becomes
  - \( V_L - (V_I + \frac{1}{8} (V_H - V_L)) = V_L - V_I - \frac{1}{8} (V_H - V_L) \)
- Output > 0

\[ V_H \quad V_L \quad V_I (\text{prev}) \]

\[ \frac{1}{8} (V_H - V_L) \]

\[ V_L \quad V_I (\text{prev}) \]

\[ \frac{1}{8} (V_H - V_L) \]

Compare after shifting another part of cap. to \( V_H \)
- \( V_I \) goes up by \( \frac{1}{16} (V_H - V_L) \)
- original \( V_L - V_I \) goes down and becomes
  - \( V_L - (V_I + \frac{1}{16} (V_H - V_L)) = V_L - V_I - \frac{1}{16} (V_H - V_L) \)
- Output < 0 (went too far again)

\[ V_H \quad V_L \quad V_I (\text{prev}) \]

\[ \frac{1}{16} (V_H - V_L) \]

\[ V_L \quad V_I (\text{prev}) \]

\[ \frac{1}{16} (V_H - V_L) \]

Input sample of \( \frac{21}{32} \)
- Gives result of 1010 or 10/16 = 20/32
- 3% error

\[ V_H \quad V_L \]
Closer Look at A-to-D Conversion

- Needs a comparator and a D-to-A converter
- Takes time to do successive approximation
- Interrupt generated when conversion is completed

A-to-D Conversion on the ATmega16

- 10-bit resolution (adjusted to 8 bits as needed)
- 65-260 usec conversion time
- 8 multiplexed input channels
- Capability to do differential conversion
  - Difference of two pins
  - Optional gain on differential signal (amplifies difference)
- Interrupt on completion of A-to-D conversion
- 0-VCC input range
- 2^LSB accuracy (2 * 1/1024 = ~0.2%)
  - Susceptible to noise – special analog supply pin (AVCC) and capacitor connection for reference voltage (AREF)

A-to-D Conversion (cont’d)

- Single-ended or differential
  - 1 of 8 single-ended
  - ADCx – ADC1 at 1x gain
  - ADC(0,1) – ADC0 at 10x
  - ADC(2,3) – ADC2 at 200x
  - ADC(2,3) – ADC3 at 200x
  - ADC(0,1,2,3,4,5) – ADC2 at 1x

A-to-D Conversion (cont’d)

- 0-VCC input range
- 2^LSB accuracy (2 * 1/1024 = ~0.2%)
  - Susceptible to noise – special analog supply pin (AVCC) and capacitor connection for reference voltage (AREF)

A-to-D Conversion (cont’d)

- Single-ended or differential
  - 1 of 8 single-ended
  - ADCx – ADC1 at 1x gain
  - ADC(0,1) – ADC0 at 10x
  - ADC(2,3) – ADC2 at 200x
  - ADC(2,3) – ADC3 at 200x
  - ADC(0,1,2,3,4,5) – ADC2 at 1x
A-to-D Conversion (cont’d)

Writing an Interrupt Handler in C (again)

- Ensure main program sets up all registers
- Enable interrupts as needed
- Enable global interrupts (SEI)
- Write handler routine for each enabled interrupt
  - What if an interrupt occurs and a handler isn’t defined?
- Make sure routine does not disrupt others
  - Data sharing problem
  - Save any state that might be changed (done by compiler)
- Re-enable interrupts upon return
  - done by compiler with RETI

Power modes

- Processor can go to “sleep” and save power
  - Different modes put different sets of modules to sleep
    - Which one to use depends on which modules are needed to wake up processor
    - Timers, external interrupts, ADC, serial communication lines, etc.
  - set_sleep_mode (mode);
  - sleep_mode ();