Micro-controller based systems

- mC has h/w interface built into chip
  - usually includeds RAM, flash, eeprom on chip
- mC has 1 or more parts
  - part pins provides general purpose I/O
    - input -> test on/off states
    - output -> generate on/off signals
- a mC can have any number of special units (s.f.u.)
  - timers (we are interested in, assignment 1)
  - UART (serial communications)
  - SPI & I^2C (serial communications)
  - Capture/Compare (CAPCOM)
    - On pin changes, capture timer value
    - On a timer value, change pin state
  - Analog to digital input
  - PWM (digital to analog)
  - Others
    - USB, DAC, CAN,…
- Each s.f.u. includes:
  - Config & data regs
  - Dedicated general purpose I/O (GPIO) pins
  - 1 more interrupts

Same

Interrupt programming

Guidelines:

- (1) Minimize time spent in an ISR
  - Stopping some executions
  - Inverse true : minimize time interrupts are disabled
  - General solution : ISR sets flags, record data, and exit
    - Regular code (running periodically) checks for changes and do processing
- (2) if the processing is “simple” , ISR can do all work
  - E.g. transmitting a buffer via a UART
    - Transmit interrupt to indicate byte completion
    - ISR simply loads the next byte into data reg
      - Or not, if at the end…
    - => This is DMA
- (3) callings routines from an ISR should be avoided
  - Stack limits => interrupts have a separate stack
  - Defined by priority levels
- (4) if an interrupt is the only trigger of activity, we can be more liberal

AVR interrupt programming:

- Interrupt priorities are fixed
- avr-gcc does a lot of work for us
    - sets the interrupt vector
    - provides macros & defines for our ISRs:

In avr/interrupt.h

ISR(INTO_vect)

{

In avr/io.h

…..

}

- cli() & sei() in avr/interrupt.h
    - fully disable/enable via the I bit in SREG
- do bit masking for individual interrupts
    - e.g. timer activation
        - to enable timer 1:
            - TIMSK1 |= _BV(TOIE1); (|= => or equal)
        - To disable timer 1:
            - TIMSK1 &= ~_BV(TOIE1); (&= => and equal) (~ => not)
- 2 types of interrupt handlers:
    - ISR – interrupts start enabled
    - SIGNAL – interrupts start disabled

Timers

- A timer simply increments the value in a register at a given frequency
    - Register can be any size (8 bit , 16 bit , … (more bit more accurate in timing))
    - Frequency is a divide from the cpu clock
        - $CPU_{frequency}$/2, 8 , …, /1024

          Or
        - Can be an external pin
    - A timer has (features):
        - A data register ($TCNT_x$) T Count
        - A configuration register to set frequency, enable interrupt, start/stop(set frequency) ($TCCR_x$)
        - Overflow interrupt
            - Fires when data register goes from xffff => x0000
            - (TOV_x), in ($TIFR_x$), set when register overflows; fires ($TIMER_x$_OVF_VECT) if ($TDIE_x$), in ($TIMSK_x$), is set.

General Programming Consideration

Timers:

- (1) we can have “fun” timing calculations
    - E.g. 1MHz with 8-bot timer
        - 256 ticks/overflow
        - Int fires every .256 millisecond
    - we “fix” by setting the data register on each interrupt (and at start)
        - e.g. set to 6 so wee get 250 ticks/int
- (2) how do we determine the amount of time between 2 events?
    - Record timer values (from data register) and do a diff.
    - What if the timer overflow?
        - Must track when an overflow occurs.
        - How many overflows?
    - What about reading the data register?
        - It takes time, and it is not atomic read...
        - (a) timer actively updates register
            - Depends on word size
        - (b) can overflow during the read
            - Must account for this in our overflow count value as we use it
                - Look at the timer overflow bit(TOV_x)
- (3) we use timers to determine ordering and control when things happen
    - But we have limited number of timers
        - Timers get assigned to s.f.u.
    - Can’t have 1 timer per task (1 timer/task)
        - Solution: we implement a heartbeat at a base frequency (e.g. 1 ms)
            - Everything runs offset from that

Basic scheduling:

(1) we start within infinite loop
    - Have a number of task (fcu calls) and each time through, do the next 1.

(2) Use a timer
    - Every x ms we get an interrupt
    - Then, run the next task
    - Don’t run task in the ISR
        - Trigger the running via the infinite loop

(short ver.) Basic Scheduling:

(1) Infinite loop
(2) Heartbeat timer to trigger running tasks in infinite loop

Power consumption:

- Pgms runs until power down
  - In infinite loop
- Lots of wasted cycles
  - Consumes power
    - Batteries drain quickly
- Microcontroller provides low power modes
  - Put micro controller into 1 or more levels of sleep

Standard infinite loop:

```
For (;;)
{
        sleep_mode();
}
```

Can config via a reg

- What wakes the microcontroller?
  - Interrupts
  - This gives us basic scheduling:
    - Sleep to idle
    - Timer (or…) interrupt wakes us
    - Execute next task(s)
- Modes define which s.f.u.'s are powered
  - Main core drives units by providing clock inputs
  - To shutdown(stop draining batteries), stop the clock

Levels:

Highest: only CPU steps
Lowest: only 1 s.f.u.

- External interrupt

Non-Volatile Memory

EEPROM (E^2) & flash

- NVM is used for 2 things:
  - (1) Parameter data
    - Configuration info:
      - Limits, settings, etc.
    - Can be set at manufacture or in the field

        - Usually done in E^2
    - (2) logging diagnostic / historical data
        - Commonly done with flash memory

E^2 issues:

- We can't read/write to it like we did in RAM
- Use a serial protocol or registers to access 1 byte at a time
    - Registers: addr, data, configuration
        - Status & initiate r/w
- E^2 operations are slow
    - That's mean we must wait for a operation to complete before accessing our registers
        - Busy wait before access
            - Reading config bits
    - Otherwise, we break E^2
        - That byte

Flash:

- We have 2 types of flash NOR based flash and NAND based flash
- Can read NOR flash using ptr manipulation
    - Part of our addr space
    - Part of our micro cotroller
- Special commands to write and erase
    - Slow
    - Write is slow
- NOR flash holds our code
- NAND flash will be off chip, and is used for logging, etc.
- NOR flash is expensive($$$) , NAND flash is cheap($) to make
- In our phone the storage we have are NAND flash
- All constant data is stored in NOR flash
    - All data and code that doesn't change is not in RAM
        - Saving memory
            - Just stack & heap in RAM
            - (*) don't use heap (malloc) in assignment!