COMP 4550
Servo Motors

Autonomous Agents Lab, University of Manitoba
jacky@cs.umanitoba.ca
http://www.cs.umanitoba.ca/~jacky
http://aalab.cs.umanitoba.ca
Servo Motors

● A servo motor consists of three components
  ○ A DC motor with an H-bridge, which allows us to turn the motor forward and backward
  ○ A position feedback sensor (potentiometers or magnetic)
  ○ A control loop (proportional and derivative) PD which control the DC motor to move to a specific position
● A servo motor can move to an arbitrary position
● Servos are very popular in remote controlled (RC) applications
Servo Motors

- Three wire input
  - Red -> Vcc 4.8V – 6V
  - Black -> GND
  - Yellow/White -> Control

- Position is controlled using pulse position modulation (PPM)
- Pulse width 0.9 – 1.8ms determines position
- Repeat rate 20ms = 50Hz
- Can be controlled using PWM outputs of the AtMega128
RC Servos: Control in main loop

- Programming of RC servo without PWM channels
  - Only 8 PWM channels (2 8-bit, 6 2-16 bit) on the ATMega 128
- CPU generates pulse on the port directly
  - This way we were able to control 40 servos on an 8 bit port
- Timing is crucial
  - Small changes in timing lead to jitter on the servo

```
Turn On Pins E4 .. 6 for servo positions 0..2
while (running != 0 )
{
    running = current_time( servo positions 0 .. 2)
    if ( running & 1 == 0 )
        Turn Off E4 – servo 0
    if ( running & 2 == 0 )
        Turn Off E5 – servo 1
    if ( running & 4 == 0 )
        Turn Off E6 – servo 2
}
delay up to 20ms
```
Attempt 1: Problems

- Timing is dependent on other interrupts or tasks in the system
- Timing is unstable and leads to large jitter
- Solution: Turn off interrupts
  - cli(); ...; sei();
  - Bad idea
    - no serial communication
    - no context switch
    - no timers
    - no sound
    - ...
  - ...
Attempt 2: Timer interrupt

- Use a timer interrupt to control the timing
- Must generate interrupts
  - to know when to turn pins on
  - to know when to turn pins off
- 4 Timer channels available on the ATMega 128
- Use a single timer for all servo motors
Attempt 2: Timer Interrupts

- Must generate interrupt each time a servo pulse needs to be turned on or off
- Turn all servos pulses on
- Calculate next deadline and set state about which servo to turn off
- May need to turn more than one servo pulse off at the same time
- What happens if
  - $\text{servo\_position}[0]=1.000000\text{ms}$
  - $\text{servo\_position}[1]=1.000001\text{ms}$
Attempt 2: Timer Interrupt

- Code becomes quite complex
- Lots of work in the interrupt handler leads to high processor utilization

Interrupt service routine (ISR)

Start: Turn all servo pulses on
- Load count[0..2] from servo positions
  state = count

Count:
- for (i=0; i<2; i++) {
  - count[i]--;
  - if (count[i] == 0 ) Turn servo pulse i off
}

- if ( count[0..2] == 0 )
  state = Delay20ms

Resolution of timer approx. 1ms to 255 positions
3.92 usec.

Discretize timing to fewer values
e.g., 10 usec

interrupt and keep track of when to turn off a servo
Attempt 3: Serialize Timer Interrupts

- Must do a lot of work to control the servo pulse
- Then wait for approximately 18ms
- Some servos work better at shorter refresh but not all
- Instead of generating the pulse on all pins at the same time, generate pulse serially
- turn PE4 on, wait, turn PE5 on wait, ...
- Side benefit: greatly improves power distribution
Use a single timer to control three or more servos
Setup a timer so that it can generate a variable delay of 0.9 ms to 1.8ms
select timer (8bit, 16 bit)
select prescalar
8 Bit value to specify the servo position

Turn off the output pin PE4-6 for the previous servo.
Read the next servo setting and convert it to a delay time.
Worry about the case when the servo is disabled/free running.
Turn the corresponding output pin PE4-6 on.
Setup the delay time on the timer.
Start the timer and wait for the next interrupt.
RC Servo Control: Serialized Timer Interrupt Version

- Assume that timer is setup correctly and setting TCNTX to servo position will yield the correct delay
- Double/triple/quadruple check your timing so that you will not destroy the servo
- Use an oscilloscope when in doubt
- Use an array `uint8_t servoSettings[3];`
- `uint8_t currentServo` to keep track of which servo to control
- Increment `currentServo` to point to the next servo. Wrap around when `currentServo` is NUM_SERVOS
ISR

volatile uint8_t servoSettings[3];

ISR(TIMERX_OVF_vector) {
    static uint8_t currentServo;

    currentServo++;
    if ( currentServo >= 3 ) {
        currentServo = 0;
    }
    PORTE = ( PORTE & 0x8F ) | mask;       // Turn all bits off and one on
    TCNTX = servoSettings[ currentServo ];
}
RC Servo ISR

- Free servo is also useful
  - Foot placement for humanoid robots
- Deal with servos are disabled: servo setting is 0
  - Do not generate a pulse
- Modify the code to add special case if servo setting is 0
ISR(TIMERX_OVF_vector) {
    static uint8_t currentServo = 0;

    currentServo++;
    if ( currentServo >= 3 ) {
        currentServo = 0;
    }
    if ( servoSettings[ currentServo ] != 0 ) {
        mask = ( 1 << (4 + currentServo) );
    } else {
        mask = 0;
    }
    PORTE = ( PORTE & 0x8F ) | mask;    // Turn off all other servos
    TCNTX = servoSettings[ currentServo ];
}
RC Servo ISR

- This code is still buggy. Why?
- Timing should be as accurately as possible
- Difference in timing between servo 0 and servo 2
- When does the pulse generation start/end?
- Read timing information on page 12 and page 343 of the AtMega 128 datasheet
- Interrupt latency is a minimum of four clock cycles
- Look at file timer.lst
/* prologue end (size=12) */

static uint8_t currentServo = 0;

uint8_t mask;

currentServo++;

if ( currentServo >= 3 )
{
currentServo = 0;
}

mask = 0x10 << currentServo;

PORTE = ( PORTE & 0xF8 ) | mask;
AtMega 128 Instruction Set

- ISR Prologue
- Where is the currentServo++ instruction?
- Use of __zero_reg__

```
78 0000 1F92       push __zero_reg__
79 0002 0F92       push __tmp_reg__
80 0004 0FB6       in __tmp_reg__,__SREG__
81 0006 0F92       push __tmp_reg__
82 0008 1124       clr __zero_reg__
83 000a 2F93       push r18
84 000c 8F93       push r24
85 000e 9F93       push r25
86 0010 EF93       push r30
87 0012 FF93       push r31
```
Serialized Timer Interrupt

/* prologue end (size=12) */

static uint8_t currentServo = 0;
uint8_t mask;
currentServo++;

lds r24,currentServo.12
subi r24,lo8(-(1))
sts currentServo.12,r24 2

if ( currentServo >= 3 )
cpi r24,lo8(3) 1
brlo .L2 1/2

currentServo = 0;

sts currentServo.12,__zero_reg__ 2

mask = 0x10 << currentServo;
lds r30,currentServo.12
clr r31

PORTE = ( PORTE & 0xF8 ) | mask;
currentServo = 0,1
Tick + 4
Prologue
push registers on the
stack
ignored
6+2=8

currentServo=2
Tick + 4
Prologue
ignored
6+1+2=9

How do we balance the code?
Ballast Code

- Add nop into processor stream
- asm syntax for gcc
- used more in Assignment 3
- asm("nop":);
- Now must add an else branch

```c
if ( currentServo >= 3 ) {
    currentServo = 0;
} else {
    asm("nop":);
}
```
Ballast Code

16:timer.c  ****  if ( currentServo >= 3 )
95  .LM2:
96 001e 8330  cpi r24,lo8(3)
97 0020 18F0  brlo .L2
17:timer.c  ****  {
18:timer.c  ****  currentServo = 0;
99  .LM3:
100 0022 1092 0000  sts currentServo.1294,___zero_reg___
101 0026 01C0  rjmp .L4
102  .L2:
19:timer.c  ****  }
20:timer.c  ****  else
21:timer.c  ****  {
22:timer.c  ****  asm("nop":);
104  .LM4:
105  /* #APP */
106 0028 0000  nop
107  /* #NOAPP */
108  .L4:
23:timer.c  ****  }
16:timer.c  **** if ( currentServo >= 3 )
95 .LM2:
96 001e 8330       cpi r24,lo8(3)     1
97 0020 18F0       brlo .L2          1/2
17:timer.c  **** {
18:timer.c  **** currentServo = 0;
99 .LM3:
100 0022 1092 0000  sts currentServo.1294,__zero_reg__  2
101 0026 01C0      rjmp .L4          2
102 .L2:
19:timer.c  **** }
20:timer.c  **** else
21:timer.c  **** {
22:timer.c  ****   asm("nop"::);
104 .LM4:
105 /* #APP */
106 0028 0000      nop 1
107 /* #NOAPP */
108 .L4:
23:timer.c  **** }
Correctly Balanced Code

```c
if ( currentServo >= 3 ) {
    currentServo = 0;
} else {
    asm("nop");
    asm("nop");
    asm("nop");
}  
```
Other solutions to balance the timing are also possible

Remove conditional or variable length instructions with constant speed instructions

- `currentServo & 0x03` – if modulo power of 2
- `(currentServo % 3)` – if constant speed div instruction
Ballast Code

- Timing is still incorrect
- Branches are always unbalanced
  - if clause takes one extra instruction
  - else clause is faster
- Must balance for that additional delay as well
Instruction Timing

- The same problem occurs when checking whether the servoSetting is 0 or not.
- Most micro-controller architectures require varying time for shifts/rotates depending on the number of shifts.
- What about the AtMega 128?
Instruction Timing

While loops are always converted into do .. while

How to access ports?

How to execute variable shifts

Local labels

1b,2f
Instruction Timing

- Accessing ports
- in/out using address of register – 32: 2 clock cycle
- lds/sts using address of register: 1 clock cycle
- Only shift/rotate left/right once are implemented as instructions on the AtMega169
- Implement a loop to execute the shift/rotate
- Local labels: 1f – 1 forward, 2b – 2 backwards
- Replace with a constant expression
Instruction Timing

```c
uint8_t const masks[3] = { (1 << PINE4), (1 << PINE5), (1 << PINE6) };
ISR(TIMER0_OVF_vect) {
    static uint8_t currentServo = 0;
    uint8_t mask;
    currentServo++;
    if ( currentServo >= 3 )
    {
        currentServo = 0;
    }
    else
    {
        asm("nop"::);
        asm("nop"::);
        asm("nop"::);
    }
    mask = masks[ currentServo ];
    PORTE = ( PORTE & 0xF8 ) | mask;
    TCNT0 = servoSettings[currentServo];
}
```
Instruction Timing

28:timer.c       ****  mask = masks[ currentServo ];
117                    .LM7:
118 0030 A091 0000  lds r26,currentServo.1295
119 0034 BB27          clr r27
29:timer.c       ****
30:timer.c       ****  PORTE = ( PORTE & 0xF8 ) | mask;
121                    .LM8:
122 0036 8EB1         in r24,46-0x20
123 0038 887F         andi r24,lo8(-8)
124 003a FD01         movw r30,r26
125 003c E050         subi r30,lo8(-(masks))
126 003e F040         sbci r31,hi8(-(masks))
127 0040 9081         ld r25,Z
128 0042 892B         or r24,r25
129 0044 8EB9         out 46-0x20,r24
Instruction Timing

- Z register is indirect addressing mode for
  - R30, R31
- Y register is indirect addressing mode for
  - R28, R29
- X register is indirect addressing mode for
  - R26, R27
- Mov instructions moves arg2 into arg1
Instruction Timing

- Restructure code
- No need to increment currentServo before the access
- Move after write to PORTE

ISR(TIMER0_OVF_vect) {
    static uint8_t currentServo = 0;
    uint8_t mask;

    mask = masks[ currentServo ];
    PORTE = ( PORTE & 0xF8 ) | mask;
    TCNT0 = servoSettings[currentServo];

    currentServo++;
    if ( currentServo >= 3 )
    {
        currentServo = 0;
    }
}