Environmental Sample Processor:
Linear Actuators

7/13/10 Brent Roman   brent@mbari.org

Linear Actuator Servo overview

- Dwarves:
  - Do all the real-time work closing servo loops
  - Send I^{2}C progress and error messages back to requester
  - Optionally output debug information to their extra RS232 port
  - Optionally interpret debug commands from same RS232 port

- Ruby:
  - Converts its objects to and from I^{2}C byte strings
  - Optimizes switches between alternative servo configurations
  - Converts to and from raw encoder counts and scales or names
  - Reinterprets dwarf error messages as Ruby Exception objects
  - Retries Error Exceptions when appropriate

Linear Actuator Positions

- LinearAxis::Position Ruby Class
- Maps to and from raw encoder counts
- A symbolic label plus an optional offset in raw encoder counts
  - slide[:position, offset]
    - Forearm[:garage, -300]   #Forearm at garage - 300 counts
    - May also be written: Forearm[:garage]-300
  - May usually be replaced by just the label when offset is zero
    - As in: Forearm.to :garage
    - Same as: Forearm.to Forearm[:garage]+0
- Positions are Ruby objects
  - myForearmGarage = Forearm[:garage, -300]
  - Forearm.to myForearmGarage
  - myForearmGarage.raw #raw position 300 counts < :garage
- Each (linear) position is defined on a specific (Slide) Axis
  - Elbow.to myForearmGarage #error because …
    - *Elbow and Forearm are different axes!*

Positions Between others
And subtracting positions

- LinearAxis::Between defines a position between two others
  - e.g. midPoint = Forearm.between :garage, :retract
    - midPoint.raw == (Forearm[:garage].raw + Forearm[:retract].raw) / 2
- Yes, you can define a position as between two Between's, etc.
  - Or between two Positions with offsets.
    - midPt2 = Forearm.between Forearm[:garage,-300], Forearm[:retract]
    - Displayed as: Forearm between garage – 300 counts and retract
- One can calculate the difference, in raw counts, between positions
  - midPoint – midPt2 = 150 #by definition
- Again, positions are bound to the axes on which they were defined
  - Elbow.between(PC, CC) – midPt2 #say, what?!
    - *Elbow and Forearm are different axes!*

Defining AxisMaps

- An AxisMap maps all raw counts to corresponding position names
  - Hash mapping raw count “detents” to a name or array of names
    - The first position name associated with a count is its “label”
    - Others are “aliases” which are acceptable substitutes
  - CC.legend => {28000=>:onguides, 0=>:home, 27000=>["closed"], 18200=>:unsealed, 20700=>:sealed, 7000=>["open", :opened]}
  - If label is a quoted String, position's article is omitted for display
    - e.g. “CC is closed” or open rather than “CC at closed” or at open

- ccMap = AxisMap.new(

```
    0=>:home, 7000=>[“open”, :opened],
    18200=>:unsealed, 20700=>:sealed, 28000=>:onguides,
    27000=>"closed"
  )
```

Linear Actuator Axis Classes

- Slide => named positions map to raw counts
  - No linear “scale”
  - Lowest level at which end-users interface with hardware
  - Think of a slide trombone with positions for arbitrary “notes”
  - e.g. Forearm, Elbow, Carousel
- Clamp => inherits from Slide
  - Adds closed?, open?, and closeAndVerifyPuckPresence
- Scale => inherits from Slide
  - Adds a linear, numeric scale to Slides
  - e.g. Elevator
- Syringe => inherits from Scale
  - adds pull, push, fill, empty volume methods
  - e.g. Collection, Processing, Sampler, Analytical syringes
- Errors come from dwarves, which are managed by Slide class
  - This is why Scale and Syringe classes report Slide errors

Basic Slide Operations

- The Slide is the “base class” for linear actuator axes
- slide.configure cfg => forces configuration object cfg to dwarf
- slide.reconfigure cfg => sends cfg only if changed from last
- slide.in(cfg) {block} => execute block in configuration cfg
- slide.position => return the slide's current position
- slide.goal => return the slide's current goal position
- slide.jog counts => move specified # of raw encoder counts
- slide.seek goal => move to specified goal position
  - Without updating servo's configuration
- slide.to goal, config => move to specified goal position
  - Updating servo's configuration if appropriate
- slide.hold => hold the current position
- slide.coast => turn off the servo
- slide.force => apply constant “force” (slide.force 0 = slide .coast)
- slide.stop => brake to a stop as fast as possible
- slide.log(decimator) {block} => log slide status while doing block
- slide.status => return current slide servo status object

Defining an Axis and associating it with an AxisMap

- Axis objects are initially created with no meaningful map

  ```
  :CC.denotes Clamp.new("Collection Clamp",
    I2C::Servos[:collection], 1, CCconfig, 30)
  ```

  - CCconfig is the servo's initial or default configuration
  - 1 is the dwarf channel number (0..1)
  - 30 is the time out in seconds for movements

- Later, to associate CC with its map (from earlier slide):

  ```
  CC.with ccMap
  ```

- Normally, both operations appear in one combined expression:

  ```
  :CC.denotes Clamp.new("Collection Clamp",
    I2C::Servos[:collection], 1, CCconfig, 30).with( AxisMap.new(
      0=>:home, 7000=>["open", :opened],
      18200=>:unsealed, 20700=>:sealed, 28000=>:onguides,
      27000=>"closed"
    ))
  ```

Using AxisMaps

- An AxisMap maps all raw counts to corresponding position names
- They are typically accessed via their associated Axis or Positions:
  - axis.legend => the AxisMap as a Hash
  - axis.list => list of all names without raw positions
  - axis.labels => list of only the position labels – omitting aliases
  - axis.maxPosition => position mapped to greatest raw counts
  - axis.minPosition => position mapped to least raw counts
  - axis.advance => move to position with next higher raw counts
  - axis.retard => move to position with next lower raw counts
  - axis.at?(position) => true if axis is at (or near) specified position
  - axis.near?(position) => true if axis is at or near position
  - axis.between?(pos1,pos2) => true if axis is (nearly) between
  - axis.rawId(rawCount) => position nearest rawCount (reverse map)
  - position.advance(detents) => position with next higher raw counts
  - position.retard(detents) => position with next lower raw counts
  - position.near?(position) => true if positions very near each other

M B A R I

How Scales Differ from Slides

- Scales inherit all the operations of Slide, adding:
  - Linear mapping of logical “amounts” or “units” to raw counts
    - rawCount = scale.countsPerUnit * amount + zero
      - zero is simply the rawCount value at 0 amount
      - scale.zero => -12580 #example case
      - scale.gain => scale.countsPerUnit => 32498.0

  - AxisMap associated with a Scale:
    - Must contain at least two positions whose labels are numeric
      - Really, there should be exactly two such positions
      - These positions define the scale's linear mapping onto counts

Scale::Skew objects

- A Scale::Skew is a generic, linear mapping object
  - scale.skew => -12580.000+32498*counts
  - scale.skew.gain => 32498.0, scale.skew.bias => -12580
  - scale.skew.apply(2) => 52416.0 # == 2*32498 – 12580
  - scale.skew.reverse(scale.skew.apply(x)) => x
  - Skew.bestFit(counts, units) => skew that best fits data
  - Skew.interpolate() => interpolates among array of skews
  -
- Scale::Skews are also used to calibrate Thermal pads!

How Syringes differ from Scales

- A syringe is merely a scale with volumetric units
- volume is simply defined as an alias for amount
- Similarly for maxVolume and minVolume
- fill method moves to the syringe's maxPosition
- empty method moves to the syringe's minPosition

ESP Dwarf DC Motor Servos

- Two identical servo channels
- 64hz sampling timebase (sample rate typically 32hz)
- Each Channel's Inputs:
  - Quadrature incremental encoder
    - (A and B 90 degrees out of phase)
  - Home flag (typically a hall effect sensor)
  - Optional threshold sensor
  - Forward and Reverse limit switches
  - One General Purpose digital input bit (for gripper)
- Each Channel Outputs:
  - PWM -100% to 100% (15 kHz with 1% resolution)
  - One General Purpose digital output bit

No Floating Point

- MSP430 would not be able to compute floats fast enough
- Avoids whole issue of round off errors
- P and D gains expressed as 16-bit integers/4096
- Positions are 32-bit encoder counts relative to “home” flag
- Time expressed in “tics”
    - Each tic corresponds to one controller sample update
    - Typically 32hz or 64hz (but could be any submultiple)
- Velocity expressed in 16-bit encoder counts per tic
    - Ensure nothing ever moves faster than 32000/counts/tic!!
- Acceleration expressed as counts/tic/tic
- Electrical Current expressed in milliamps
- Pressure expressed in ADC counts (application must convert)

Configuration Object Details

- :samplePeriod = number of 64hz timebase tics per sample tic
  - Default value = 2 (Typically 1 or 2)
- :encoder, :threshold, :home sensor power / polarity
  - Default value = :off (may be :positive or :negative)
- :homeDirection = :forward or :reverse
  - Default value = :reverse
  - :reverse moves negative if home flag inactive
- :brake = short motor terminals on servo error (:false or :true)
  - Default value = true
- :debug = output servo state at sample rate while seeking goal
  - Default value = false

Control Gains and Factors

- PID :gain struct with members P, I, and D
  - Default values for each are 0
  - Servo will not operate until at least one is non-zero
  - Effective value of P and D is divided by 4096
  - I is effectively divided by 16384
- :friction compensation gain
  - cmdVel * friction / 4096 added to PWM output
  - cmdVel = Commanded velocity
- :stiction compensation factor
  - If negative cmdVel, subtract stiction/2 from PWM
  - If positive cmdVel, add stiction/2 to PWM

Trajectory Generator (1 of 2)

- :acceleration & :deceleration in counts/tic/tic
  - Default values for each are 0, normally positive
  - Specify negative acceleration to disable “softstart”
  - Zero :deceleration implies deceleration=abs(acceleration)
- :maxSpeed = plateau velocity in counts/tic
  - Temporarily reduced when PWM limits reached to prevent trajectory errors due to low battery voltage
- :minSpeed = slowest acceptable progress rate (counts/sec)
  - Speed error if maxSpeed reduced below minSpeed
- :maxSettling = max tics to allow to servo to settle at goal
  - Default 0, typically 2 – 3 seconds worth of tics
  - Just ensures that position error not returned too early

Trajectory Generator (2 of 2)

- :stopWindow detemines how nearly goal should be reached
  - Specified in encoder counts (16 bit limit max)
  - Temporarily increased each time goal is passed
  - Special Value false indicates no (more) reseeks allowed
  - Defaults to Special Value :deceleration = deceleration rate
  - Also accepts value :acceleration
- :hunt determines whether to adjust setpoint after goal reached
  - Defaults to false, set true to “fight” to hold exact position at goal
  - Setpoint is *never* adjusted if position within stopWindow
- :thresholdOffset determines how far from threshold to stop when reached
  - Defaults to 0 encoder counts
  - When threshold reached before goal, goal = position + thresholdOffset
  - Used to position top of puck stack with respect to ESP's top plate

Core Limits

- :maxPWM & :minPWM
  - Max must be >= min, but each may be negative or positive
  - Constrains servo output, but does not constrain “force” command
  - Effective maxSpeed is reduced when servo reaches these PWM limits

- :maxPositionErr determines absolute maximum tolerable servo error in different contexts:
  - SeekErr if stopWindow grows too large due to repeatedly missing goal
  - TrajectoryErr if position becomes too far from setpoint while transiting
  - PositionErr if position moves too far from goal after arrival

- :maxCurrent determines maximum allowable motor current
  - In milliamps
  - Should never be set > 2000mA

Pressure Limits

- :maxInPress, :maxOutPress, :minInPress, :minOutPress
  - 0 to 4095 ADC counts
  - Maximum/Minimum tolerated Intake and Outlet pressures
  - Constraint disabled if corresponding max == min
  - All default to 0

- :maxDeltaPress & :minDeltaPress -- (-4095 to 4095)ADC counts
  - Maximum/Minimum tolerated pressure difference
  - Constraint disabled if set to special value: false
  - All default to false (there is no corresponding value true)

- Generic “Pressure Error” results if any of the above are violated
  - One must check status to determine the exact problem

Pressure Servo Configuration

- :inputDeltaPress determines if pressure delta is sensed or derived
  - True to input the difference from ADC 7
  - False to derive it as (intake – outlet) pressure
  - Defaults to false
- :pressBias is subtracted from delta pressure before use
  - In servo or limit check
  - Defaults to 0
- :pressGain is the proportional gain of the pressure servo
  - Scaled like P and D, pressGain is *4096
  - Reduces acceleration from that normally determined by the trajectory generator.
  - Never causes command velocity to fall below minSpeed

Defining an I2C::Servo::Configuration

```
###  Processing Syringe, derived from default.with … ###
:PSconfig.denotes I2C::Servo::Configuration.default.with(
  :encoder=>:negative, :home=>:negative,
  :homeDirection=>false,
  :maxPositionErr => 65,  #upped from +/- 0.3ul to 2ul
  :gain => PIDgain.new(3500, 3000, 1300),
  :friction => 170,
  :maxSpeed => 100, :minSpeed => 30,
  :acceleration=>5,   #deceleration == acceleration if unspecified
  :maxCurrent =>120,      #bracket bends too much if set any higher
  :maxSettling => 3*32
)
```

- From betty's configure.rb

Servo Configuration Example

```
###  Processing Syringe, derived from PSconfig … ###
:PSslow1.denotes PSconfig.dup.with (
     :maxSpeed => 10,  :minSpeed => 2,
     :acceleration => 2
 )
```

- From betty's configure.rb
- Alternative configuration to standard PSconfig on previous page

Switching Among Configurations

```
###  The “Hard”, wrong way ###
PS.configure PSslow1
PS.seek  PS.maxVolume/2  #half full (or is it empty?)
PS.configure Psconfig
```

- But, what if PS was not “in” the PSconfig configuration?
- What if PS was already in PSslow1?

```
###  The harder, correct way  ###
 oldPSconfig = PS.config
 begin
     PS.reconfigure Psslow1
     PS.seek PS.maxVolume/2
 ensure  #in case an error occurs between here and previous 'begin'
     PS.reconfigure oldPSconfg
 end
```

Switching Servo Configurations

```
### The easy (and correct way) ###
PS.to  PS.maxVolume/2, PSslow1  #half full (or is it empty?)
```

- Only changes the configuration if necessary
- Don't use .seek unless sure the config already loaded on dwarf.

```
### The hard (and also correct way)  ###
PS.in PSslow1 do
    PS.to  PS.maxVolume/2
    PS.empty   #this is still in PSslow1
end
PS.fill   #old configuration restored (likely PSconfig)
```

- slide.in {block} constructs may be nested arbitrarily deep

I2C::Servo::Status Objects

- :enabled = true if servo control is active
- :pastRLS, :pastFLS, :pastThreshold, :home
  - True if corresponding switch is closed
- :position = 32-bit signed offset from home position
- :velocity = 16-bit signed in encoder counts/tics
- :current = signed milliamps
  - Always agrees with sign of PWM status below
- :PWM = signed percent PWM duty cycle
- :err = 16-bit signed (setpoint – position)
- :voltage = raw motor voltage (in volts)
  - This is the *only* floating point value

Servo Pressure Status

- Recall that pressure may be a proxy for any arbitrary volage input
- :inPress = intake pressure in raw ADC counts (0-4095)
- :outPress = outlet pressure in ADC counts
- :deltaPress = delta pressure in ADC counts
  - This is always ADC channel 7
  - It is *not* affected by the :inputDeltaPress configuration flag

Capturing Slide Servo Status Logs

- To record a real-time trace of a Slide servo's behavior:

  ```
  slide.log(decimator) { block }
  ```

  - Where block is (usually) code that exercises the actuator
  - Returns a large array of I2C::Servo::Status objects
    - Size of result array depends on how log {block} runs !!
    - Records one sample for every decimator servo updates
      - I2C bus traffic can overload slow ARM host board if decimator==1
        - » Only recovery possible may be to reset dwarf
        - » Attach faster Linux host to ESP's gateway to avoid this
    - If there is an error, partial result is stored in $errLog global variable
  - slide.log method calls may not be nested
    - Beware that Clamp.close uses slide.log, use Clamp.to :closed instead
- Example:

  ```
  ccLog = CC.log(2)  {CC.to :closed}; nil  #to prevent display of large array
  ccLog.each {|stat| puts [stat.current, stat.velocity]}; nil
  ```

Plotting Slide Servo Status Logs

- Connect an ESP Linux workstation to the ESP's gateway
  - This may require use of a USB<->RS232 serial adapter
  - /dev/I2Cgate must by symlinked to the that adapter
- The package used for plotting is called quickplot
- Load quickplot interface code

```
require 'plot'    #only once per session
```

- To produce each new plot window:

```
plot  slide.log(2) {blockOfCodeExercisingSlide}
```

- e.g. plotting default status fields of position, velocity and current:

```
  plot  CC.log(2)  {CC.to :closed}
```

- e.g. plotting :current,:voltage, :pwm, and :err

```
plot(CC.log(2) {CC.to :closed},  :current, :voltage, :pwm, :err)
```

Remotely Plotting Servo Logs

- Difficult to configure
  - but well worth it for tuning Slide servos' PID gains.
  - Does not require opening the ESP enclosure to change any connections
- Add ssh key to workstation's authorized_keys file

– So that the ESP host can run commands without password prompts

  - Test from Linux shell prompt, on ESP host, by invoking:

```
$ ssh  workstation  ls
```

  - This is a security breach. Remove key when done if it worries you.
- Edit remotePlot method utils/plot.rb as necessary
  - To change the workstation name and the display number

– As before:

```
require 'plot'    #only once per session
```

- To produce each new plot window:

```
remotePlot  slide.log(2) {blockOfCodeExercisingSlide}
```