MONTEREY BAY AQUARIUM RESEARCH INSTITUTE

Physical State

5/24/22 Brent Roman   brent@mbari.org

ESP Physical State

- Canister environmental sensors
- Power Switches
- All moving actuators:
  - Rotary Valves
  - Solenoid Valves
  - Syringes
  - Clamps
  - Carousel
  - Elevator
  - Elbow
  - Gripper
- Puck Heaters

Gateway Canister State

- Quietly logged once every
  - 10min for 2G, 2min for 3G

`-> Gate.queryCan`   #or simply `-> can`

```
Can@22:40:11, 24.2C, 67% humidity, 14.2psia, 13.684V, 0.256A, 187.834Ah, 3.50W
```

  - Ah is only available with newest firmware

`-> Gate.queryCan` takes an immediate reading

  - `can` is an alias for `Gate.queryCan`

`-> Gate.can` uses the most recent reading

`-> Gate.canPollInterval=30.seconds`

`-> Gate.canPollInterval=0  #no polling`

Canister State details

```
-> Can.unit          #all available can sensors
   {avgCurrent: {format: "%.3fA avg"},
    batteryUsed: {format: "%.3fAh"},
    current: {format: "%.3fA"},
    humidity: {format: "%d%% humidity"},
    pressure: {format: "%.1fpsia"},
    temperature: {format: "%.1fC"},
    voltage: {format: "%.3fV"},
    waterAlarm: {format: "%d%% Wet!",
    threshold: 1.5}}
-> Can.missionDuration   #duration power has been on
   5 weeks, 3 days, 10:19:32.40625
-> c=can; (c.voltage*c.batteryUsed).round   #~Wh used
```

Gateway Power Switches

```
-> Power  #state of all power switches
   {analytic1: false,
    analytic2: false,
    analytic3: false,
    camera: false,
    core: false,
    raw: false,
    sampler: false}
-> Power.sampler :on    #turn sampler on
-> Power.on :sampler    #turn sampler on
-> Power.off :sampler, :camera    #turn off sampler and camera
-> Power.offSince[:sampler]    #when sampler was turned off or nil
-> Power.onSince[:core]        #when core was powered on or nil
-> Power.sampler    #True if sampler is on
```

Linear Actuator Class Hierarchy

- Slide => Lowest level at which end-users interface with hardware
  - Think of a slide trombone with named positions for arbitrary “notes”
  - `Forearm, Elbow, Carousel`
- Clamp => inherits from Slide
  - Adds closed?, open?, and closeAndVerifyPuckPresence
- Scale => inherits from Slide
  - Adds a linear, numeric scale (in standard units) to Slides
  - `Elevator` #unit = puck height
- Syringe => inherits from Scale
  - adds pull, push, fill, empty volume methods
  - Collection, Processing, Sampler, Analytical syringes
- Thermal => inherits from Scale
  - replaces motor with a heater

- Errors come from microcontrollers, which are all managed by Slide class
  - This is why Linear Actuators report most errors as “Slide::Error”

Other Actuator Classes

- Shaft is a rotary actuator
  - all of which just happen to spin a rotary valves
- Gripper is a two state actuator
  - with binary position sensing
  - May control a robotic hand or a motorized valve
- Solenoid is a two or three state solenoid actuator
  - without position feedback sensing
- Valve is a Solenoid used to control fluid flow
- Valve::Manifold is an array of Valve
  - sharing a common fluid path

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

Axis Map Example

- Hardware counts <==> names and aliases

```
-> Forearm.legend
{-12793 => 3, -12782 => 5, -12726 => 7,-12711 => 4,
 -12706 => 2, -12699 => 1, -12678 => 6,
 -12501 => [CC, :collection, Collection],
 -12418 => [PC, :processing, Processing],
 -12342 => [:garage, FlushPuck, FlushPuck::Garage],
 0 => :home,
 2800 => ["retracted", :retract, :clear]}
```

- Defined in configure.rb as:

```
Forearm.detents 0=>:home, 2800=>"retracted",
     -12501=>CC, -12418=>PC, -12342=>:garage,
     -12699=>1, -12706=>2, -12793=>3, -12711=>4, -12782=>5,
     -12678=>6, -12726=>7
```

Solenoid Valves

- Solenoid valve state is either open, closed, or unknown
- Intake and Exhaust external solenoid valves
  - Unipoler, normally closed
  - Draw a lot of power while opened
- Solenoid valves in Manifolds
  - Bipoler, latching
  - Latching Valve state is initially unknown
  - Draw power in short pulses when changing state

-> `Intake.to :open` #holds Intake open

-> `Intake.open` #ditto

-> `Solenoid` #shows Intake and Exhaust state

Solenoid Valves Plumbing

- Each ESP 2G Dwarf microcontroller
  - drives 8 solenoids numbered 0..7
- To show how a solenoid is connected:

```
-> Intake.wiring
```
#how is Intake wired?

```
sampler[6]
```
#it's 2nd to last on sampler dwarf

Valve::Manifold

- Each composed of a series of Solenoid::Valve plus an endName
  - Valve::Manifold state = either the name
    - of its first opened Solenoid::Valve
    - or its endName, if no Solenoid::Valve is open
- Individual valves in the manifold can be accessed

```
-> CSR.series[0].open   #open 1st CSR valve
-> CSR.wiring   #shows wiring connections
[:lysis <=> collection[0],      #these need not be in order
 :diluent <=> collection[1],
 :RNAlater <=> collection[2],
 :mfbkill <=> collection[3],
 :kill <=> collection[4],
 :flush <=> collection[5]]
```

Valve::Manifold Configuration

- Collection Series configuration is often machine specific
- Example for ESPwaldo, defined in its configure.rb file:

```
:CSR.denotes Valve::Manifold :Collection, [
   Valve.reagent(:lysis, CollectionValves,0),
   Valve.reagent(:diluent, CollectionValves,1),
   Valve.reagent([:RNAlater, :rnal], CollectionValves,2),
   Valve.reagent(:mfbkill, CollectionValves,3),
   Valve.reagent(:kill, CollectionValves,4),
   Valve.reagent(:flush, CollectionValves,5)  ], :air
```

- :air is the manifold's “endName” representing
  - its state when all its series valves are closed
- :rnal is an alias for :RNAlater

aliases and labels

- Each named position has exactly one label
  - in addition, it may have any number of aliases
- A position is output by its label
  - but may be input by its label or any corresponding alias

```
-> CSR.aliases
{rnal: :RNAlater}
-> CSR.alias :clean=>:flush    #clean now an alias for flush
-> CSR.relabel :atmosphere, :air
-> CSR.to :air
Collection Valve::Manifold selects atmosphere
```

- rather than air!

Rotary Valves

- Each rotary valve is controlled by its Shaft position
  - Shaft is the class of all rotary valves
- Raw shaft state is an angle from 0..511
  - A number of these may be defined as named positions
- Rotation direction to the new goal position
  - may move over raw position 0
  - may be specified
    - to avoid moving over another position
- A goal position may be specified
  - as a named position
  - exactly between 2 named positions
  - as a raw offset from one of the above

Shaft Configuration

- Each 2G Dwarf may control up to 4 rotary valves numbered 0..3

```
-> PTV.wiring        #displays wiring information
processing[2]        #wired to 3rd channel of processing dwarf
-> PTV.legend        #displays position map
{64 => [PRV, 1,      #positon labeled PRV [with alias 1] is raw angle 64
 192 => [:PRVmixing, 2],
 320 => [:mixing, 3],
 448 => [:puck, 4]}
```

Rotating Shafts

`-> PTV.to :mixing` #rotates to mixing position fastest way ‘round

`-> PTV.select :mixing, avoiding: PRV`

- #rotates to mixing position
- in a direction that avoids rotating by the PRV position

`-> PTV.select :mixing, via: PRV` #rotates opposite way!

- #rotates to mixing position via the PRV position

`-> PTV.dialBetween :mixing, :puck`

- #rotates to between mixing & puck position fastest way ‘round

`-> PTV.dialBetween :mixing, :puck, avoiding: PRV`

- #rotates to between mixing & puck position, avoiding PRV

`-> PTV.rawAngle` #return the raw angular position of the shaft

`320`

`-> PTV.at? :mixing` #true if PTV is at, or very near, mixing

`true`

Gripper Characteristics

- Two position actuator with minimal position feedback
- In one of two states
  - or transitioning between those states
- The two states are named when configured
  - they may not have aliases or labels
- Examples are:
  - The ESP 2G Hand was original Gripper actuator
  - Some ESP 2G external Sample valves are controlled as Grippers
  - 3G External rotary valves are all Grippers

Gripper Use

```
-> Hand.close       #closes the Hand Gripper
-> Hand.open        #opens it
-> Hand.state       #:open, :closed, or :unknown
-> Hand.open?       #true if Hand is open
true
```

- Only on NOAA GLERL 2G ESPs...

```
-> Sample.deep        #moves Sample valve to its :deep position
-> Sample.shallow     #moves Sample valve to its :shallow position
```

- `-> Sample.state        #:shallow, :deep, or :unknown`
- `-> Sample.deep?        #true if Sample valve is in its :deep state`

```
false
```

Slide Characteristics

- Linear Actuator having precise position feedback
- Raw positions are in hardware counts
  - Each Slide's count units may be different
  - Most Slides require physical 'homing'
    - to calibrate their counts position sensor
    - A Slide that is not yet homed is 'lost'
  - are mapped to names with associated AxisMap
- Each Slide includes a set of named configurations
  - that set motor limits and velocity profile
  - only one such configuration is active at a time

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

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
      - [there should be only two numerically labeled positions]
      - These positions project the scale's linear mapping onto counts

Scale::Skew objects

- A Scale::Skew is a generic, linear mapping object
  - represents
    y=mx * b

    - scale.skew => -12580.000+32498*counts
    - scale.skew.gain => 32498.0, scale.skew.bias => -12580

    - scale.skew.apply(2) => 52416.0 # == 2*32498 – 12580
      - solves for y (engineering units) given x (counts)

    - scale.skew.reverse(scale.skew.apply(x)) => x
      - solves for x (counts) given y (engineering units)

    - 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 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

2G 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

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

Default Processing Syringe I2C::Servo::Configuration

```
:PS.denotes Syringe "Processing Syringe",
                    :processing, 0,
  :encoder=>:negative, :home=>:negative,
homeDirection:false,
  maxPositionErr:65,
  gain:PIDgain(3500, 3000, 1300),
  friction:170,
  maxSpeed:100, minSpeed:30,
  acceleration:5,
  maxCurrent:120,
  maxSettling:3*32

PS.maxDuration=160   #how long can a move take?
```

- *excerpted from shallow/preconfig.rb*

Alternative Processing Syringe Servo Configurations

- Based on default configuration shown on previous slide:

  PS.defCfg :fast, maxSpeed:300

  PS.defCfg :slow, maxSpeed:50

  PS.defCfg :slow1, maxSpeed:10, minSpeed:2, acceleration:2

- Based on :slow1 configuration (defined above):

  PS.defCfg :slow2, :like=>:slow1, minSpeed:5

- *excerpted from shallow/postconfig.rb*

Switching Servo Configurations

The *easy (and correct way)*

```
PS.to  PS.maxVolume/2, :slow1  #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 :slow1 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

Plotting Slide Servo Trajectories

- 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 possibly the display number)

  -> require 'plot' #only once per session
- To produce each new plot window:
  -> remotePlot *slide*.log {blockOfCodeExercisingSlide}
- e.g. plotting default status fields of position, velocity and current:

  -> plot CC.log {CC.to :closed}
- e.g. plotting :current,:voltage, :pwm, and :err

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