ESP

MONTEREY BAY AQUARIUM RESEARCH INSTITUTE

Operating Modes: Real vs. Simulated

4/21/15 Brent Roman   brent@mbari.org

M B A R I

Mission Simulation

- Simulate missions before deployment to catch
  - Syntax errors
  - Missing, wrong, or extra parameters
  - Configuration errors
    - Trying to pull a reagent that is not configured/defined
  - Scheduling errors
    - Not leaving enough time between mission phases
    - Scheduling recovery before last phase completes
- Simulate adaptive sampling triggers
  - With recorded or generated CTD data
  - Observe when sampling occurs
    - Adjust trigger conditions as needed
- Run simulations on ESP itself, or on a Linux desktop/laptop

ESP Operating Modes

- ESPmode=real
  - Normal operation in real-time with real hardware
  - Default mode
- ESPmode=simreal
  - Real-time with simulated hardware
- ESPmode=simfast
  - Simulated time (with simulated hardware)
  - ~1000 faster than ESPmode=real
    - *But may be only 50 times faster on the ESP itself*
- ESPmode=quick
  - Like simfast mode
  - Produces less output
  - Recommended for validating mission scripts

More Operating Modes

- ESPmode=debug
  - Like real mode, but displays all I2C messages on the console
- ESPmode=quiet
  - Like real, but displays only errors on the console
- All modes are defined as ruby files in the mode subdirectory
  - One may easily create their own custom modes.
  - Mode definition files are named:
    - $ESPHOME/mode/*mode_name*.rb

Simulation Procedure

- ESPmode must be set before starting the ESP software
- Change the mode for all subsequent runs with:
  ESPmode=*newMode*
- Restore normal mode for all subsequent runs with:
  ESPmode=real
- Change mode w/o affecting subsequent runs with:
  ESPmode=*newMode* esp   *mission*
  - Omit *mission* to simulate interactively
- Most typical simulation command:

  ESPmode=quick esp *myNewMission*

Simulation Features and Limits

- Protocols are simulated in full detail
  - Every movement of the physical hardware is simulated
  - Every I2C message is simulated down to the byte level
  - Puck handling assumes that there are no stack height errors
  - Will not detect mechanical interference between axes
    - E.g. attempts to move the carousel with the Elevator up will ***succeed*** in sim
    - But, attempts to move the Elevator past its physical limits will ***fail*** in sim
  - One should test new protocols by simulating them first, before wasting reagents.
- Does simulate CTD
  - But not ISUS
- Tracks consumption of Time, but not:
  - Reagents
    - This is next on my TODO list
  - Power
- Simulation of whole missions is CPU intensive
  - Allow 90 minutes to simulate a full mission on the slow ESP processor
  - The same sim would take < 30 seconds on a fast server.
  - Figure on it taking 90 seconds for the typical laptop

Declaring Puck Stack Heights

- Puck stack height cannot be measured in simulation
  - Puck load must be prescribed in simulations
- Every new mission should define the number of pucks expected to be loaded in each tube!
  - Optional in “real” mode, but...
  - Isn't it better to “fail early” if puck load is wrong?
- Excerpt of mission with 6 pucks in tubes 2, 3 and 4:

```
pucks 2=>6, 3=>6, 4=>6    # see next slides
mission startTube:  2, until:  “9AM  4/10/15”  do
    <mission phases>
end
```

- Fails immediately if tube 2 did not start with exactly six pucks

Declaring Puck Stack Heights

- New commands to set and query the expected stack height:
  - clear! *tubeList=1..7*
    - Clears each specified tube's stack height
  - fill! *numPucks=22*, *tubeList=2..6*
    - Puts the specified number of pucks in each listed tube
  - pucks *tubeHash={}*
    - Puts the specified number of pucks in specified tubes
    - If tubeHash omitted, just displays the # of pucks in each tube

Detailed Stack Height Setting

- fill!
  - Fills all tubes except #1 (for typical fully loaded carousel mission)
- fill!; clear! 2, 4..7
  - Ends up with tube 3 containing 22 pucks, others empty
- fill! 9
  - Fills all tubes except #1 with 9 pucks
- fill! 9, 1, 3..5, 7
  - Fills tube 1, 3, 4, 5 and 7 with 9 pucks
- pucks 2=>22, 6=>18
  - Fills tube #2 with 22 pucks, tube #6 with only 18
- pucks
  - Changes nothing
  - Just returns the hash of pucks in tubes.