Environmental Sample Processor Core Library:

Threads, Events, Logs, Mutexes, Time and Date parsing

2/17/10 rev 2 Brent Roman brent@mbari.org

The ESP Log

- Log messages are generated whenever:
  - Explicitly placed in a protocol
    - Text is logged
  - A low-level core command is executed that changes the ESP's state
    - The command and parameters are logged as text
    - E.g. SC.to 4 #move the carousel to tube 4
  - A command is sent, or reply received on the Dwarf (I^2C) bus
    - The binary message is logged (not printable)
  - An Unhandled Exception propagates all the way up a thread's call stack
  - A data message for the GUI is stored (new)
- Some messages are also displayed on the users terminal
  - Which are is determined by the ESP's operating mode
    - ESPmode=debug displays all text messages
    - ESPmode=quick displays very little
- Recall that the operating mode also determines the name of the log file
  - Log files are binary data. Don't expect to be able to load them as text files.
  - Use the dumplog command to view binary log files.
- The logging subsystem is started first
  - Because all the other ESP software components use it.

Ruby Threads

- Lightweight parallelism within a single Ruby program
  - Linux “processes” run as independent programs
    - Each of which may be (separate) Ruby interpreters!
  - Threads share memory, processes do not
  - Threads are more efficient, but less safe
    - Any thread may read or over write data owned by another
    - A process may not access the memory of another
  - Ruby's threads are similar to Java's early “green threads” implementations
    - The Ruby interpreter manages them
    - The Unix kernel does not even know Ruby threads exist
      - Ruby 1.9 changes this (but we still use Ruby 1.6.8)

- Basic Ruby Threads are:
  - Unnamed → There's no way to “look one up” if one doesn't have a reference
    - Each is referred to only by its internal (non-printable) object identifier
  - Independent → No parent/child relationships maintained between them
    - Parent not notified when a child thread dies due to an error

ESP Threads

- ESP threads extend the Ruby Thread base class
  - Not a superclass of Thread
- Each created with a name
  - Typically a symbol, but may be a number or string
    - :heating, 31, 3.14, “heating”
    - Note that :heating != “heating”
- Each has a parent and child threads
  - The first parent is, by definition, the one that spawned it
    - In practice, there is always only one parent thread
    - When spawned: children.last == parents.first
  - Errors raise exceptions that propagate:
    - Up the tree of threads via parents.first
    - Down the tree via children
    - To avoid having orphaned “zombie” threads awaiting actions of other dead threads

- Each may be associated with multiple Checkpoints
  - Checkpoints record the complete state of the thread
  - So it may be resumed (or recovered) at a later time

ESP Thread Operations

- Thread**[*name*]** → look up thread by its given name
- MainThread → main ESP execution thread
- *thread*.**name** → the name of the thread
  - Thread[*aName*].name == *aName* by definition
- *thread*.**birthdate** → real-time at which thread was spawned
- *thread*.**parents** → list of thread's parents
  - childThread.parents.first == The thread that created childThread
- *thread*.**children** → list of thread's children
- *thread.***status** → threads readiness to run
  - “run”, “sleep”, false, or nil
- *thread*.**finish** → wait for thread to end, returning its result
- *thread.***exception** → list of recent unhandled exceptions
  - i.e. Why thread aborted due to an error
  - Only the last few are such exceptions are remembered
  - Output with **puts,** as in: **puts MainThread.exception**
- *thread.***lastErr** == *thread.exception.last*
- *thread*.**details** → summary of thread state
- *thread*.**progress** → summary list of recent checkpoints
  - Only the last few are retained.
  - Output with **puts**, as in: **puts MainThread.progress**
- *thread*.**checkpoint** → list of recent checkpoints
  - Each is very large, so only the most recent are retained.
- There are more, less often used, operations...

ESP Thread Resumption from Checkpoints

- A checkpoint records the complete state of a thread
  - Ruby and CompSci geeks call checkpoints “Continuations”
  - Few mainstream programming languages support Continuations
- One can resume a thread from any previously stored checkpoint
  - One cannot resurrect a dead thread!
  - Thread having defined checkpoints that experience an error are made “moribund”
    - Thread without checkpoints are allowed to die, as there's no way to resume them
- Checkpoints are stored as a side effect of writing (most) log messages
  - The message text is the checkpoint's name
  - *thread*.**progress** just outputs the name of each such stored checkpoint
- A special Checkpoint is stored when certain operations fail.
  - E.g. Commands to Dwarves, PCR commands, etc.
  - Such Checkpoints are called recovery points
    - They are not included in the list returned by *thread*.checkpoint
    - They are associated with the **Exception** the error caused
  - *thread*.**recover** → retries operation that caused the most recent recoverable error
- thread.resume → resumes thread from the most recent checkpoint
  - *thread*.resume(-***n)*** → resumes thread after the ***n***th most recent checkpoint
- Resume may require manually moving the ESP back to the appropriate state
  - If the most recent checkpoint is at all old. That's why you should try recover first!
- Common values for *thread* are MainThread, Thread[:blocking], Thread[:sh2], etc.

ESP Event Scheduler

- Defers execution of a block of Ruby code to some exact, future time.
  - This code executes in the scheduler thread, but often affects others.
- Time may be real or simulated
  - ESP code never accesses Linux time directly for this reason
- Time may not advance until all ScheduleThreads are ready
  - Each event is processed completely before time can advance
  - A single thread that “hangs” will stop time from advancing
    - Unless it “unsyncs” itself from the rest of the ScheduleThreads first
  - This rule is necessary to ensure deterministic behavior
  - A ScheduleThread is “ready” when it is waiting for input from an external device
- ScheduleThread is a superclass of Thread
  - All child threads of ScheduleThreads are normally ScheduleThreads

- Outstanding events are maintained on a list sorted by the time at which they are to run
- It is common for events to be removed from this list
  - Error “time-outs” are implemented by deferring the error processing event
    - The error processing event is removed in the **normal** case

- *delay 3* → defer code to wake up the current thread at Thread.time+3
- Same as *delayUntil Thread.time+3*
- *Delay.sleep 3* → delay thread 3 seconds without outputting anything in the log

Recursive Mutexes

- Mutexes prevent interleaved access to an abstract or concrete resource
  - That would otherwise lead to data corruption or inconsistent operation
- They ensure exclusive serial access by a single thread
  - The thread, after having locked the resource, is said to own it.
- Mutexes are hard to manage and error prone
  - The ESP Ruby code uses very few for this reason
    - Arm, FlushPuck, the I^2C bus, and two or three others.
- ESP Mutexes are “recursive”
  - Recursive mutexes may be redundantly locked and unlocked
  - They contain a count of how many times the owner has locked the resource
  - 30.times{Arm.lock}; 29.times{Arm.unlock} → Arm.lock
- Resources must be released in exact reverse order in which they were allocated
  - The Dining Philosophers have a Mexican Standoff and starve otherwise
  - Threads blocked waiting for each other's resources are said to be “**deadlocked**”
    - Resulting in starvation for the resource
    - http://en.wikipedia.org/wiki/Dining_philosophers_problem
- This is why the FlushPuck is always claimed after the Arm
  - And the FlushPuck is always released before releasing the Arm

Date Parsing

- Unix time → dates must be >=1970 and <= 2038
- Dates and Times must be quoted in “strings”
- *month* / *day* / *year* or *day – month – year* or *monthName day, year*
  - *month* may be numeric or an English month name or abbreviation
  - *year* may be 4-digit or 2-digit (xx>=70 is assumed 19xx, else 20xx)
- *year* % *dayOfYear* → julian date
- Above may include dayOfWeek specification
  - Days of the Week must be written as English names or abbreviations
  - Beware of overspecified dates, ie. “Sat 2/15/09”
    - ArgumentError ... -- 02/15/09 falls on a Sunday -- not Saturday
- Last date/time entered is remembered as a reference
  - First date entered must specify a year
  - When fields are omitted, next date meeting remaining criteria can be chosen
- Examples: Time ...
  - “2/17/10” or '10-2-17' or '10%48' or 'February 17, 2010' → *today*
  - 'Sat' → *the next Saturday i.e. 2/20/10*
  - “4/5” → *April 5th, 2010*
  - “%300” → *October 27, 2010*

Time Parsing

- hh:mm:ss.fraction
  - All the above are optional
  - May be followed by AM or PM
    - If omitted, 24 hour format is assumed (military time)
  - May be proceeded or followed by three letter time zone code
    - UTC and GMT are equivalent
    - The only other option is the local time zone
      - You may not specify EST unless host's local time is Eastern Standard !!
- Last time entered is remembered as a reference
  - When fields are omitted, next time meeting remaining criteria can be chosen
  - This may be in the next day
  - Time may be proceeded by a plus sign (+) to explicitly add to the last time entered

- Examples: Time ...
  - “2/17/10 1PM” or '10-2-17 1PM' or '10%48 1PM' or '1PM February 17, 2010' → *Wed Feb 17 13:00:00 PST 2010*
  - “9AM” → *9AM Thursday*
  - “23:59:59.100” → *nine tenths of a second before midnight Thursday*
  - “2::.” → nine tenths of a second before 2AM Friday
  - “14:20” → exactly twenty minutes after 2PM Friday
  - “12:10 Feb 17, 2010 → ten minutes after noon on February 17th, 2010
  - “+:5” → five minutes later (fifteen after noon)