Understanding and Recovering from Errors

6/1/22 Brent Roman brent@mbari.org

Errors are Unhandled Exceptions

- Great. So, what is an Unhandled Exception?
  - Exceptions are:
    - Unusual conditions that obstruct the normal flow of a program
    - Handled by special code outside the usual flow

- In modern languages, when a method cannot return a valid value...
  - It “throws” (or “raises”) an exception instead!
    - `10/0` => ZeroDivisionError
    - `Math.sqrt()` => ArgumentError: wrong # of arguments(0 for 1)
    - `Math.sqrt(-1)` => Errno::EDOM -- argument out of domain

- Exceptions propagate up the call stack in search of a “handler”
  - Handler code may be very specific or generic

- If no handler is found, the exception is said to be “unhandled”

Ruby Exception Objects

- Consist of:
  - a text message describing the exception
  - A backtrace to locate the point of failure in nested methods
  - Subclasses may (and do) associate extra information
    - e.g. The servo status associated with Slide::Error

- Only subclasses of built-in Exception class may be raised/thrown
  - One cannot throw/raise a Thread or an Integer, etc.
  - Exceptions are otherwise just like any other Ruby object
  - Exceptions are always raised on a specific Thread

- Ruby's “rescue” clause encloses all exception handlers
- If no matching rescue clause found, the thread is terminated
  - ESP threads always include a top-level Exception handler
    - Logs error details just before the failing thread exits

Exception Class Hierarchy

- `ArgumentError.ancestors` =>
  [ArgumentError, StandardError, Exception, Object, …]

- `NameError.ancestors` =>
  - [NameError, ScriptError, Exception, Object, …]

- `Slide::Error.ancestors` =>
  [Slide::Error, LinearAxis::Error, Axis::Error, AxisKernel::Error,
  StandardError, Exception, Object, …]

- Example of a class that cannot be raised as an Exception:
  - `Slide.ancestors` =>
    [Slide, LinearAxis, Axis, … , AxisKernel, Object, …]
  - because it does not inherit from the Exception class

Deriving Ruby Exception Objects

- Define my own error (exception) class and raise it
  - class MyErr < StandardError; end
  - raise MyErr.new “Your honor, I respectfully object!”

- Define a Slide::Error with associated (servo status) reply and axis:

```
class Slide < LinearAxis
  class Error < LinearAxis::Error
    def initialize text, axis, reply=nil
      @reply = reply
      super text, axis
    end
    attr_reader :reply
  end
```

- So, in addition to the base Exception's backtrace and message Slide::Error exceptions store servo axis and status reply

Backtraces (1 of 2)

- Answers the question: *Where did the error occur?*
- Example:

```
ESPbruce:002:0> CC.to :spoon   #there is no spoon :-)
   Axis::Error in quick -- Unknown Collection Clamp position: spoon
ESPmack:003:0> backtrace
/home/brent/esp2/lib/axis.rb:513:in `baseRaw'
/home/brent/esp2/lib/axis.rb:297:in `raw'
/home/brent/esp2/lib/axismap.rb:175:in `fetch'
/home/brent/esp2/lib/axismap.rb:163:in `fetch'
/home/brent/esp2/lib/axis.rb:281:in `fetch'
/home/brent/esp2/lib/axis.rb:292:in `raw'
/home/brent/esp2/lib/axis.rb:553:in `raw'
/home/brent/esp2/lib/slide.rb:325:in `toRawGoal'
/home/brent/esp2/lib/slide.rb:333:in `seek'
/home/brent/esp2/lib/slide.rb:382:in `to'
(ESP):2
/opt/mbari/lib/ruby/1.8/irb/workspace.rb:52:in `irb_binding'
/opt/mbari/lib/ruby/1.8/irb/workspace.rb:52
=> #<Axis::Error: Unknown Collection Clamp position: spoon>
```

Use your text editor to seek to line numbers in each file referenced

In vi, simply enter a line number at the : prompt

In nedit, type control-L to type line number into a dialog box

Backtraces (2 or 2)

- One may also list the offending Ruby code within the ESP app
  - First, let's remember the most recent error

```
-> err = Thread.current.exception.last
-> list err  #list the error line + a few following
-> list err, 20, -10   #20 lines starting 10 before
```

- To access backtrace levels above the lowest

```
-> err[-4]   #the 4th call level from the topmost
/home/brent/esp2/lib/slide.rb:382:in `to'
```

– equivalent to writing -> `err.backtrace[-4]`

```
-> list err[-4], 5, -2
def moveTo goal, tmpCfg=nil, maxDuration=@maxDuration
#like seek, but allows for a servo configuration for just this move
  return seek goal, maxDuration if tmpCfg.nil? or tmpCfg.equal? @config
  inConfiguration(tmpCfg) {seek goal, maxDuration}
end
```

Rescuing Ruby Exceptions

- Exception handlers are just blocks of code within a rescue clause

```
def safeDivide num, den
  num/den
rescue ZeroDivisionError
  prompt "Divide by zero!? "
rescue StandardError => err
  Log.recordException err  #does not raise beyond here
  :silly  #return :silly on other errors
end
```

- The exception's derived class determines how it is handled
  - ***Not its message text***
    - Error text messages just attempt to explain the error to users

ESP Top-Level Exception Handling

- Each ESP thread has an associated queue of unhandled exceptions
  - Thread[*name*].exception => list of most recent errors
  - Only the most recent 20 or so unhandled exceptions are preserved
  - The last is the most recent, the first is the oldest
  - Thread[*name*].exception displays all thread's recent errors

- The backtrace method with no arguments method displays
  Thread.current.exception.last.backtrace
- backtrace *:name* displays
  Thread[*:name*].exception.last.backtrace
- backtrace *thread* *displays*
  thread.exception.last.backtrace
  - e.g. backtrace MainThread == backtrace

- To save the 2nd to last error (prevent losing it off the queue)
  myErr = thread.exception[-2]
- Later use: backtrace myErr to display exception's backtrace

Ruby Script Errors

- NameError ==> specified method or variable is not defined
- SyntaxError ==> grammatical error
  puts "foo" If 3>2 #If should be lowercase if
- LoadError ==> cannot process specified Ruby script file
  execute “missingFile”
- The above errors will *always* require that Ruby script be edited.

Generic Runtime Ruby Errors

- ArgumentError ==> number and/or class of objects being passed into a method are incompatible with its definition
- TypeError ==> method does not handle the type of object passed in
- Interrupt ==> Unix INTerrupt signal sent to ESP Ruby process
- SignalException ==> Another Unix signal sent to ESP process
- IRB::Abort ==> Control-C pressed on ESP's server terminal
- UserAbort ==> Control-C pressed on espclient terminal
- RuntimeError ==> generic error (text message will describe it)
  - raise "something bad's happened" #raises a RuntimeError
- ZeroDivisionError ==> e.x. 10/0

Internal ESP logging errors

- Log::Locked ==> can't start 2nd ESP app in the same ESP::Mode

  Log errors below indicate serious bugs or configuration problems

- Log::CannotDump ==> attempt to log object containing files or procs
  Certain objects cannot be converted to a byte stream

- Log::Error ==> other internal error

- Log::Reader::Error ==> invalid log file format encountered by dumplog
  - May be caused by read log from different type of ESP
    - i.e. trying to dump a standard core's log from an MFB
    - Or trying to dump MFB equipped ESP's log from one lacking MFB

Scheduling Errors

- Schedule::EventInPast ==> time is in the past
  trying to schedule an operation (or delay) before current time

- Schedule::Stop ==> scheduler has been stopped by error or user
  produced as ESP app terminates (no recovery possible)

- Delay::Error ==> invalid duration syntax
  e.g. delay “1 fortnight”

- Delay::TooLate ==> phase start time in past by > Delay::MaxLate
  Did the previous phase run long?
  - Or, phase start time in future > Delay::MaxWait

- Delay::Late ==> phase start time past by < Delay::MaxLate
  Warning only. Did the previous phase run long?

Thread Errors

- Thread::Aborted ==> another thread requested this one be aborted
  t.abort #raises Thread::Abort in thread t

- Thread::ParentDied ==> the thread that spawned us had a fatal error
  Thread::ChildDied ==> a thread this one spawned had a fatal error
  Child threads may “orphan” themselves to avoid these errors

- Thread::Checkpoint::Resume ==> users should never see this...
  Exception raised in a moribund thread to resume it

Common System (Errno::) Errors

- ECONNREFUSED ==> host refused requested network service
  - may indicate that the host is still booting up
- EDOM ==> invalid {numeric} domain {e.g. sqrt(-1)}
- ENOTTY ==> data file used where an interactive terminal required
- EPIPE ==> connection between processing broken
  - a pipe error often indicates that a network connection timed out
- EPERM ==> file permission error
  - e.g. user has no permission to access or write to file in question

I2C Bus Errors

- I2C::DuplicateAddress::Error ==> two dwarves have same address
  check dwaves' dip switches very carefully
- I2C::LAN::NoGateway::Error ==> the I2C network lacks its gateway
  configuration error – not generally recoverable
- I2C::Parser::Error ==> response sent by dwarf improperly formatted
  - could be caused by very outdated firmware or electrical noise
- I2C::Request::Timeout ==> expected response not received in time
  usually indicates a motor or sensor is failing – not a network failure
- I2C::UnexpectedReply ==> received unexpected dwarf response
  May happen when rapidly logging data. Unexpected replies ignored.
- I2C::NodeOffline ==> dwarf is not responding to its address
  This **is** a network problem
- I2C::MsgErr ==> host is trying to send improperly formatted message
  Can occur in simulation of “unmodeled” operations
- Tag::Error ==> Message tags are inconsistent
  Internal ESP Error

I2C Message Processing Errors

- I2C::Solenoid::Error ==> trying to send invalid solenoid control msg
  likely a bug in lib/solenoid.rb
- I2C::Servo::Error ==> trying to send invalid servo control message
  likely a bug in lib/slide.rb or very outdated dwarf firmware
- I2C::Shaft::Error ==> trying to send invalid rotary valve control msg
  likely a bug in lib/shaft.rb
- I2C::SerialPort::Error ==> trying to send invalid dwarf serial port msg
  likely a bug in lib/serialport.rb
- I2C::SerialPort::Configuration::Error ==> invalid RS232 configuration
  unsupported port baud rate, parity, etc.
- I2C::RS232Port::Error ==> invalid dwarf RS232 serial port config
  port baud rate, parity, stop bits, etc.
- I2C::RS232Port::ReadError ==> dwarf received garbled serial data
  parity or framing errors usually indicate wrong baud rate or cabling
- I2C::Thermal::Error ==> trying to send invalid thermal control message

Contextual Sensor Errors

- Instrument::ISUS::NoACK ==> ISUS didn't acknowledge cmd receipt
  cabling problem?
- Instrument::CTDSample::Error ==> corrupt sample received
  likely trying to run a new v2 CTD with old Ruby driver
- Instrument::CTD::NotWhileLoggingError ==> can't sample if logging
  CTD should never be put into autonomous logging mode
- Instrument::CTDCore::CalFileMismatch ==> bad seabird cal file
  or a valid cal file given the wrong file name
- Instrument::CTD::Warning ==> missing cal file
  will still log data, but engineering units are suspect
- Instrument::ReadTimeout ==> instrument did not respond in time
  check cables, batteries, try CTD or ISUS.term
- Instrument::NoDataError ==> no sample available (yet)
- Instrument::Sample::Error ==> generic sample error

Email Errors

- Email::SendTimeout ==> email message could not be sent
  after numerous retries

- SystemCallError, SocketError, TimeoutError, EOFError
  various networking errors that will be retried

- Net::SMTP* ==> Email server is incompatible with ESP client
  should not happen if you are mailing with the default configuration

- Net::Proto* ==> Email server is incompatible with ESP client
  should not happen if you are mailing with the default configuration

Event Trigger Errors

- Trigger::Holdoff::Error ==> specified negative trigger holdoff value

- Trigger::Error ==> inappropriate time to enable triggers
  mission busy or not running a mission at all
- Trigger::Restart ==> trigger conditions being reevaluated
  (not an error)
- Trigger::Aborted ==> trigger conditions being disarmed
  (not an error)

Axis Errors

- AxisKernel::Missing ==> some dwarf did not respond to role call
  Check I2C and power cabling, verify configure.rb matches hardware

- AxisKernel::Error ==> trying to define the same axis object twice
  Likely a problem with the machine's configure.rb file

- Linear *or* Rotary Axis::Error ==> seeking unknown position
  Could be high level protocol bug or missing info in configure.rb

- Slide::Error ==> not yet homed or other servo error
  Likely missing ESP.ready!, mechanical problem or servo out of tune

- Scale::Error ==> invalid Scale object configuration
  Lacking 2 numeric positions or have numeric aliases for same position

Valve Errors

- Valve::Error ==> configuration error or selecting undefined position
  If during configuration, two positions likely have the same name

- Valve::Manifold::Error ==> config error or selecting undefined valve
  If during configuration, two valves likely have the same name

- Solenoid::Error ==> low-level configuration error
  Likely a low-level solenoid type is defined ambiguously
  example: two states sharing the same name

Puck, Clamp & Arm Errors

- Puck::Error ==> one of various high-level sanity checks failed
  Puck counting logic detected a misplaced puck
  Failure to specify type of puck to load or unload
  Unspecified Source or Destination tube number
  Out of pucks (emptied tube 7)

- Puck::Warning ==> specified puck type does not match that in clamp
  you explicitly specify unload an :sh2, but you'd loaded an :sh1 puck
  Not fatal, just a warning written to the log

- Clamp::Error ==> clamp open/closed inappropriately or missing puck
  Likely someone left a puck in a clamp or forget to put one there
  Clamp::VelocityError ==> puck detection algorithm failed
  e.g. Clamp never reached plateau velocity

- Arm::StretchError ==> failure in Arm.stretch!
  Arm may be mechanically jammed, unable to reach stops

Thermal::

- AmbientChanged ==> ambient temperature changed
  Ambient temperature changed by > 2C while heating puck

Power Errors

- Busoff ==> power was off when it needed to have been on
  e.g. trying to access a microcontroller that is powered off

ShallowSampler::

- IntakeClogged ==> Intake pressure will not equalize with exhaust's
- Clogged ==> Puck filter is clogged
  (sampling ends, puck is evacuated, processing begins)
- Error ==> fatal sampling error
  - Pressure sensor failure
  - Puck leak detected
  - Syringe jammed
  - Filter clogged while priming
  - Clamp not closed on puck
  - Failure to specify a filter bubble point

Thread::Checkpoint

- Each Checkpoint contains a specific thread's complete call stack
  - ESP's Checkpoints are built upon Ruby's standard “Continuations”
    - Adds a timestamp and an Exception (with its backtrace)
  - Threads can be “resumed” from when the ckpt was stored
  - Global $variables are not stored, nor any other thread's variables
    - Nor is the physical state of the ESP somehow “stored” !!!
  - Log.record “text” creates a checkpoint called “text” as a side effect
  - Many errors create checkpoints just before stopping the thread
    - Such stopped threads are said to be suspended or “moribund”

- Without a checkpoint, the only recourse is to restart the thread
  - With a mission custom coded to pick up from the current state
- With a checkpoint, one must only restore the ESP's state to one consistent with conditions as of the time the checkpoint was created.
  - Often, valves must be correctly set – but it can also be more subtle!
- Checkpoints cannot be used to resurrect a terminated thread
  - Other threads to be resumed must be suspended/moribund

Managing Checkpoints

- Checkpoint objects are large. Old ones are not usually relevant
  - So, for each thread, only the last 20 or so are retained in a queue
    - If you want to save one “forever”, just assign it to a variable

```
-> mainCkPt = MainThread.checkpoint.last
```

  - *thread*.progress #displays that thread's last few checkpoints

```
-> MainThread.progress
-> Thread[:sh2].progress
```

    - The most recent is the last line output

  - *thread*.checkpoint returns an array of checkpoints
    - *thread*.checkpoint.last (or .[-1]) is the most recent
    - *thread*.checkpoint[-2] is the 2nd most recent
    - *thread*.checkpoint.first (or .[0]) is the oldest recorded
    - *thread*.checkpoint[1] is the 2nd oldest

  - These [*index*] operations are common to all Ruby arrays

Resuming from Checkpoints

- *thread*.resume is equivalent to *thread*.checkpoint.last.resume
- *thread*.resume(-2) == *thread*.checkpoint[-2].resume
- How would you resume from the oldest recorded checkpoint?

- *thread*.recover is equivalent to *thread*.resume

```
-> trouble    #lists all suspended threads and their checkpoints
```

- Take care to clean up operations that might have altered ESP state
  - e.g. Turning off heaters, closing outer valves, etc.

- Not all errors have associated checkpoints
  - eg. NameError, SyntaxError, LoadError, etc.
  - Such errors are not “recoverable”
    - *thread*.recover will fail if *thread* has recorded no checkpoints

Resuming from Checkpoints (cont'd)

- One can change global variables while threads are suspended
  - To reset parameters that caused the error, etc.
- One cannot change local variables.
  - The stack embedded in the checkpoint is immutable until resumed

- However, one can patch code!
  - But, not for any methods that are on the checkpoint's stack.
    - One must resume or recover from a checkpoint before the method(s) being patched were called.
      - Modify the file(s) containing those methods
      - Reload the methods with the “define” or “reload” commands:
        - define “filename”
        - reload method :methodName

Resuming after a Slide::Error

- An I2C::Servo::Status snapshot is associated with each Slide::Error
  - containing information to help diagnose the cause of the failure
    - *Was the motor being driven hard at the instant of failure?*
    - *How fast and in what direction was it moving?*
    - *Was the voltage supplied to the dwarf within specification?*

- Example simulation of trying to drive Sampler Syringe past its :empty position

```
-> SS.to 0
-> SS.jog -5000    #try to drive plunger down past empty
Slide::Error in simreal -- Sampler Syringe speedErr at empty
-> ssErr = Thread.exception.last   #remember error
-> ssErr.xray      #show all we know about the error
-> ssErr.reply.status   #show just the error status snapshot
-> SS.to 2         #move SS up so the jog should succeed next time
-> Thread.progress     #show all available checkpoints
-> Thread.resume       #resume from most recent
```

- Resuming after Sampler Syringe speedErr at empty at 18:19:51
- SS.jog -5000
- Sampler Syringe at 0.88ml

```
-> Thread.resume       #repeat until we crash again
```

Slide::Error status details

- Linear Slide status may indicate one of the following movement errors:

- :notReady ==> motor encoder has not yet been homed
  run `ESP.ready!` or `Axis.home.to :home`
- :positionError ==> servo failed to hold position within maxPositionErr
- :speedError ==> motor could reach minSpeed
- :trajectoryError ==> servo could not follow configured velocity profile
- :invalidChannel ==> indicates a configuration error
  motor channels must be 0..1
- :invalidGoal ==> seeking invalid raw position
- :aborted ==> another thread sent movement command before arrival
- :overCurrent ==> motor current limit exceeded
- :pressureOutOfBounds ==> pressure limit (high or low) exceeded
- :hitLimit ==> physical travel end limit switch closed

Slide::Error configuration details

- Linear Slide status may indicate one of the following errors:

- :invalidConfig ==> generic configuration error
  review machine's configure.rb
- :invalidSpeedConfig ==> maxSpeed==0 or minSpeed<maxSpeed
- :invalidMaxPositionErr ==> maxPositionError <= 0
- :invalidOutPressureLimits ==> Outlet pressure max < min
- :invalidOutPressureLimits ==> Inlet pressure max < min
- :invalidDeltaPressureLimits ==> Delta pressure max < min
- :invalidPWMlimits ==> PWM max < min
- :invalidAcelleration ==> Acceleration==0

- :invalidAbsolutePositionConfig ==> (ESP 3G only)
  absolute slide position sensors improperly configured

Complications in Resuming

- Restoring ESP's hardware state is straightforward
  - Usually it suffices to move actuators (valves, etc.) back to where they were at the checkpoint's timestamp
  - Scan the log backward from the checkpoint's timestamp to determine the position of all relevant actuators.

- Restoring software state, however, may be tricky!
  - What resources did the thread own at the checkpoint's timestamp?
  - Are they exactly the some as those the moribund thread owns now?
    - Moribund threads keep certain resources
      - Arm, FlushPuck: kept to prevent other threads' interfering.
    - But heaters are relinquished (shut off)
      - To conserve power and avoid damage.

  - Note that files being read or written cannot be reread or rewritten.
    - Not usually a problem in practice...

Resuming Arm&Puck Operations

- Trouble attempting to rearrange pucks for resume from checkpoint when the suspended thread owns the Arm, Hand, or FlushPuck
- System hangs for a while and reports:
  Waiting >20 seconds for *thread* to relinquish *Resource*
- One must steal Arm, Hand and/or FlushPuck in order to recover
  - Acquire the resources, move pucks, then restore resources' owners.
    - oldOwner = *Resource*.owner #should be the suspended thread
    - Acquire with: *Resource*.steal!
    - Move pucks around as needed to make ready to resume
    - *Resource*.changeOwner oldOwner

  - Example when "resource" == Arm

```
-> oldOwner = Arm.owner    #the thread controlling the Arm
-> Arm.steal!    #steal control from that moribund thread
... move Arm as needed to prepare to resume from <ckpt> ...
-> Arm.changeOwner oldOwner    #restore rightful owner
-> oldOwner.resume <ckpt>    #resume from <ckpt>
```

Resuming Heating Operations

- Heaters usually turn off if an error occurs in whatever thread owns them.
- No problem if checkpoint timestamp is before heating began
  - Because the thread will reacquire the heater and restart it
- Otherwise, one must restart the heater for the thread being resumed
  - Verify that heater is no longer owned by suspended thread
    - e.g. *Heater*.owner #should be either nil or the suspened thread
    - If *Heater*.owner is nil, it will be necessary to:
      - Repeat commands necessary to restore heater temperature.
        - It may also be necessary to wait until temperature stabilizes.
      - Give control of the heater back to the suspended thread
        - *Heater*.owner = threadBeingResumed
      - Resume the thread
- *Heater* will be one of CH, PH, SPE, etc.