Cardano.BM - benchmarking and logging

Alexander Diemand Andreas Triantafyllos

November 2018

Abstract

abstract ...

Contents

Chapter 1

Cardano BM

1.1 Introduction

introduction ...

1.2 Overview

In figure 1.2 we display the relationships among modules in *Cardano.BM*. The arrows indicate import of a module. The relationship with a triangle at one end would signify "inheritance", but we use it to show that one module replaces the other in the namespace, thus refines its interface.

1.3 Examples

examples ...

1.4 Code listings

1.4.1 Cardano.BM.Observer.STM

```
stmWithLog :: STM.STM (t, [LogObject]) → STM.STM (t, [LogObject])
stmWithLog action = action


bracketObserveIO :: Trace IO → Text → STM.STM t → IO t
bracketObserveIO logTrace0 name action = do
  (traceTransformer, logTrace) ← transformTrace name logTrace0
  bracketObserveIO' traceTransformer logTrace action
bracketObserveIO' :: TraceTransformer → Trace IO → STM.STM t → IO t
bracketObserveIO' NoTrace _ action =
  STM.atomically action
bracketObserveIO' traceTransformer logTrace action = do
  countersid ← observeOpen traceTransformer logTrace
  -- run action, returns result only
  t ← STM.atomically action
```

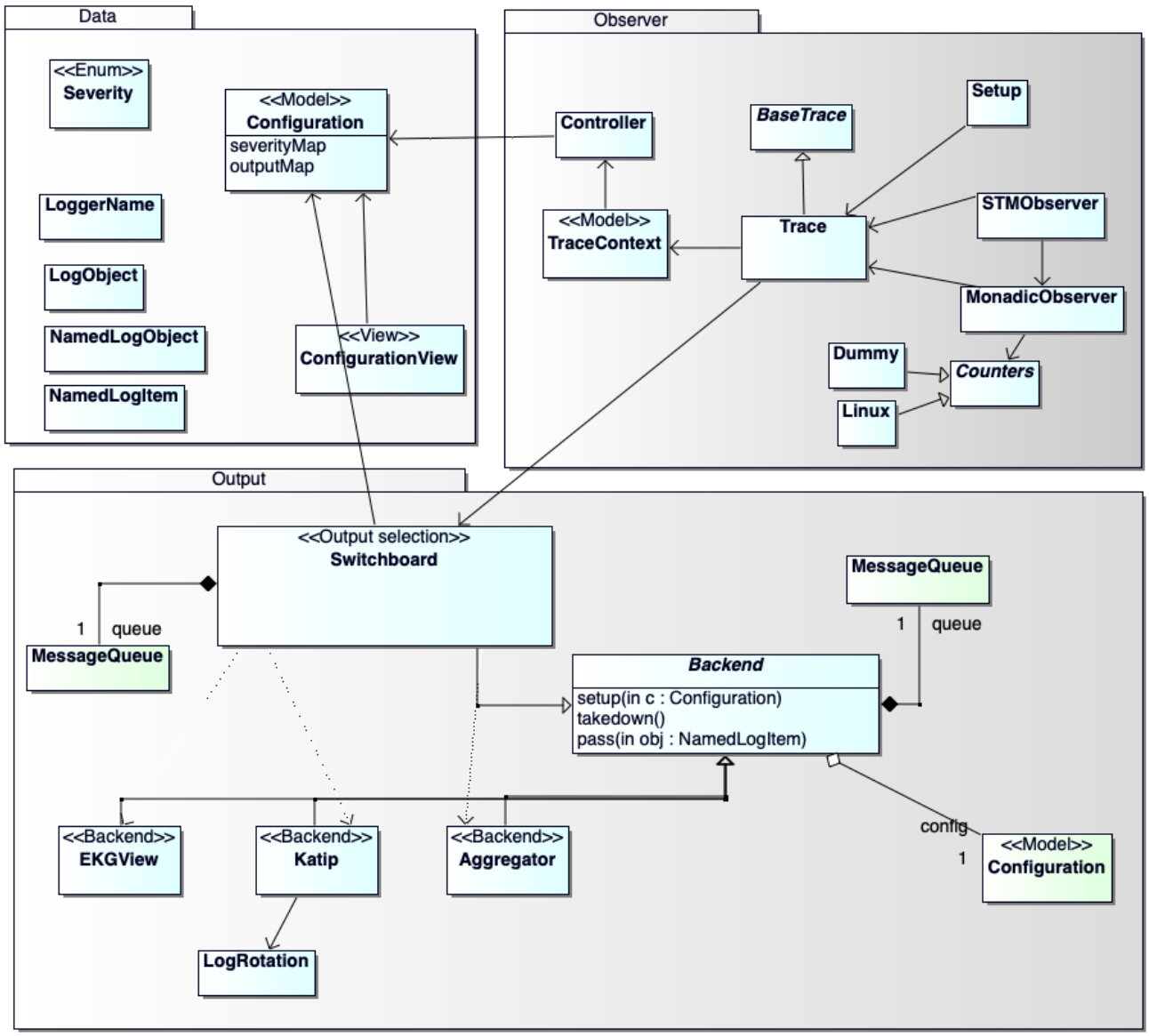

Figure 1.1: Overview of module relationships

```
  observeClose traceTransformer logTrace countersid [ ]
  pure t


bracketObserveLogIO :: Trace IO → Text → STM.STM (t, [LogObject ]) → IO t
bracketObserveLogIO logTrace0 name action = do
  (traceTransformer, logTrace) ← transformTrace name logTrace0
  bracketObserveLogIO' traceTransformer logTrace action
bracketObserveLogIO' :: TraceTransformer → Trace IO → STM.STM (t, [LogObject ]) → IO t
bracketObserveLogIO' NoTrace _ action = do
  (t, _) ← STM.atomically $ stmWithLog action
  pure t
bracketObserveLogIO' traceTransformer logTrace action = do
  countersid ← observeOpen traceTransformer logTrace
  -- run action, return result and log items
  (t, as) ← STM.atomically $ stmWithLog action
  observeClose traceTransformer logTrace countersid as
  pure t
```

1.4.2 Cardano.BM.Observer.Monadic

```
-- Observes an action and adds name given in the logger
-- name of the given Trace.  If the empty Text is
-- given as name then the logger name remains untouched.
bracketObserveIO :: Trace IO → Text → IO t → IO t
bracketObserveIO logTrace0 name action = do
  (traceTransformer, logTrace) ← transformTrace name logTrace0
  bracketObserveIO' traceTransformer logTrace action
bracketObserveIO' :: TraceTransformer → Trace IO → IO t → IO t
bracketObserveIO' NoTrace _ action = action
bracketObserveIO' traceTransformer logTrace action = do
  countersid ← observeOpen traceTransformer logTrace
  -- run action
  t ← action
  observeClose traceTransformer logTrace countersid [ ]
  pure t


observeOpen :: TraceTransformer → Trace IO → IO CounterState
observeOpen traceTransformer logTrace = do
  identifier ← newUnique
  logInfo logTrace $ "Opening: " <> pack (show $ hashUnique identifier)
  -- take measurement
  counters ← readCounters traceTransformer
  let state = CounterState identifier counters
  -- send opening message to Trace
```

```
    traceNamedObject logTrace $ ObserveOpen state
    return state


observeClose :: TraceTransformer → Trace IO → CounterState → [LogObject] → IO ()
observeClose traceTransformer logTrace (CounterState identifier _) logObjects = do
    logInfo logTrace $ "Closing: " <> pack (show $ hashUnique identifier)
    -- take measurement
    counters ← readCounters traceTransformer
    -- send closing message to Trace
    traceNamedObject logTrace $ ObserveClose (CounterState identifier counters)
    -- trace the messages gathered from inside the action
    forM_ logObjects $ traceNamedObject logTrace
```

1.4.3 BaseTrace

Contravariant

A covariant is a functor: F A → F B
A contravariant is a functor: F B → F A

Op a b implements the dual to 'arrow' *"getOp :: b → a"*, which when applied to a *BaseTrace* of type *"Op (m ()) s"*, yields *"s → m ()"*. In our case, *Op* accepts an action in a monad *m* with input type *LogNamed LogObject* (see 'Trace').

```
newtype BaseTrace m s = BaseTrace {runTrace :: Op (m ()) s}
```

contramap

A covariant functor defines the function *"fmap :: (a → b) → f a → f b"*. In case of a contravariant functor, it is the dual function *"contramap :: (a → b) → f b → f a"* which is defined.

In the following instance, *runTrace* extracts type *"Op (m ()) s"* to which contramap applies *f*, thus *"f s → m ()"*. The constructor *BaseTrace* restores *"Op (m ()) (f s)"*.

```
instance Contravariant (BaseTrace m) where
    contramap f = BaseTrace ∘ contramap f ∘ runTrace
```

traceWith

Accepts a *Trace* and some payload *s*. First it gets the contravariant from the *Trace* as type *"Op (m ()) s"* and, after *"getOp::b → a"* which translates to *"s → m ()"*, calls the action on the *LogNamed LogObject*.

```
traceWith :: BaseTrace m s → s → m ()
traceWith = getOp ∘ runTrace
```

natTrace

Natural transformation from monad *m* to monad *n*.

```
natTrace :: (forall x. m x -> n x) -> BaseTrace m s -> BaseTrace n s
natTrace nat (BaseTrace (Op tr)) = BaseTrace $ Op $ nat . tr
```

noTrace

A *Trace* that discards all inputs.

```
noTrace :: Applicative m => BaseTrace m a
noTrace = BaseTrace $ Op $ const (pure ())
```

1.4.4 Cardano.BM.Trace

```
appendName :: MonadIO m => LoggerName -> Trace m -> m (Trace m)
appendName name (ctx, trace) = do
  -- append the name to the existing one
  let previousLoggerName = loggerName ctx
      newLoggerName = appendWithDot previousLoggerName name
      ctx' = ctx {loggerName = newLoggerName}
  -- inherit the Severity of the given Trace.
  mPreviousSeverity <- liftIO $ getNamedSeverity ctx' previousLoggerName
  case mPreviousSeverity of
    Nothing -> return (ctx', trace)
    Just sev -> liftIO $ setNamedSeverity ctx newLoggerName sev
                >> return (ctx', trace)
-- return a BaseTrace from a TraceNamed
named :: BaseTrace m (LogNamed i) -> LoggerName -> BaseTrace m i
named trace name = contramap (LogNamed name) trace
```

TODO remove *locallock*

```
locallock :: MVar ()
locallock = unsafePerformIO $ newMVar ()
```

Concrete Trace on stdout

This function returns a trace with an action of type "(*LogNamed LogObject*) → *IO* ()" which will output a text message as text and all others as JSON encoded representation to the console.

```
stdoutTrace :: TraceNamed IO
stdoutTrace = BaseTrace $ Op $ \lognamed ->
    case lnItem lognamed of
      LP (LogMessage logItem) ->
        withMVar locallock $ \_ ->
```

```
            output (lnName lognamed) $ liPayload logItem
        obj →
          withMVar locallock $ \_ →
            output (lnName lognamed) $ toStrict (encodeToLazyText obj)
    where
      output nm msg = TIO.putStrLn $ nm <> "  ::  " <> msg
```

Trace into katip's queue

```
katipTrace :: {-MVar LoggingHandler -¿ -} TraceNamed IO
katipTrace = BaseTrace $ Op $ λlognamed → do
  lh ← readMVar Internal.loggingHandler
  mayEnv ← Internal.getLogEnv lh
  case mayEnv of
    Nothing → error "logging not yet initialized. Abort."
    Just env → logItem'
      lognamed
      (K.Namespace (T.split (≡ '.') (lnName lognamed)))
      env
      Nothing
      (Internal.sev2klog Info)
      (K.logStr ("" :: Text))
```

Every *Trace* ends in the switchboard which then takes care of dispatching the messages to outputs

```
mainTrace :: TraceNamed IO
mainTrace = BaseTrace $ Op $ λlognamed → do
  Switchboard.pass lognamed
```

Concrete Trace into a *TVar*

```
traceInTVar :: STM.TVar [a] → BaseTrace STM.STM a
traceInTVar tvar = BaseTrace $ Op $ λa → STM.modifyTVar tvar ((:) a)

traceInTVarIO :: STM.TVar [LogObject] → TraceNamed IO
traceInTVarIO tvar = BaseTrace $ Op $ λln →
  STM.atomically $ STM.modifyTVar tvar ((:) (lnItem ln))

traceNamedInTVarIO :: STM.TVar [LogNamed LogObject] → TraceNamed IO
traceNamedInTVarIO tvar = BaseTrace $ Op $ λln →
  STM.atomically $ STM.modifyTVar tvar ((:) ln)


traceConditionally
  :: (MonadIO m)
  ⇒ TraceContext → BaseTrace m LogObject → LogObject
  → m ()
```

```
traceConditionally ctx logTrace msg@(LP (LogMessage item)) = do
  flag ← liftIO $ checkSeverity ctx item
  when flag $ traceWith logTrace msg
traceConditionally _ logTrace logObject = traceWith logTrace logObject
```

Enter message into a trace

The function *traceNamedItem* creates a *LogObject* and threads this through the action defined in the *Trace*.

```
traceNamedItem
  :: (MonadIO m)
  ⇒ Trace m
  → LogSelection
  → Severity
  → Text
  → m ()
traceNamedItem (ctx, logTrace) p s m =
  let logmsg = LP $ LogMessage $ LogItem {liSelection = p
    , liSeverity = s
    , liPayload = m
    }
  in
  traceConditionally ctx (named logTrace (loggerName ctx)) $ logmsg
logDebug, logInfo, logNotice, logWarning, logError
  :: (MonadIO m) ⇒ Trace m → Text → m ()
logDebug logTrace = traceNamedItem logTrace Both Debug
logInfo logTrace  = traceNamedItem logTrace Both Info
logNotice logTrace = traceNamedItem logTrace Both Notice
logWarning logTrace = traceNamedItem logTrace Both Warning
logError logTrace = traceNamedItem logTrace Both Error
logDebugS, logInfoS, logNoticeS, logWarningS, logErrorS
  :: (MonadIO m) ⇒ Trace m → Text → m ()
logDebugS logTrace = traceNamedItem logTrace Private Debug
logInfoS logTrace  = traceNamedItem logTrace Private Info
logNoticeS logTrace = traceNamedItem logTrace Private Notice
logWarningS logTrace = traceNamedItem logTrace Private Warning
logErrorS logTrace = traceNamedItem logTrace Private Error
logDebugP, logInfoP, logNoticeP, logWarningP, logErrorP
  :: (MonadIO m) ⇒ Trace m → Text → m ()
logDebugP logTrace = traceNamedItem logTrace Public Debug
logInfoP logTrace  = traceNamedItem logTrace Public Info
logNoticeP logTrace = traceNamedItem logTrace Public Notice
logWarningP logTrace = traceNamedItem logTrace Public Warning
logErrorP logTrace = traceNamedItem logTrace Public Error
logDebugUnsafeP, logInfoUnsafeP, logNoticeUnsafeP, logWarningUnsafeP, logErrorUnsafeP
  :: (MonadIO m) ⇒ Trace m → Text → m ()
```

```
logDebugUnsafeP logTrace = traceNamedItem logTrace PublicUnsafe Debug
logInfoUnsafeP logTrace     = traceNamedItem logTrace PublicUnsafe Info
logNoticeUnsafeP logTrace = traceNamedItem logTrace PublicUnsafe Notice
logWarningUnsafeP logTrace = traceNamedItem logTrace PublicUnsafe Warning
logErrorUnsafeP logTrace   = traceNamedItem logTrace PublicUnsafe Error
```

```
traceNamedObject
    :: Trace m
    → LogObject
    → m ()
traceNamedObject (ctx,logTrace) = traceWith (named logTrace (loggerName ctx))
```

transformTrace

```
-- Transforms the Trace given according to content of
-- TraceTransformerMap using the logger name of the
-- current Trace appended with the given name.  If the
-- empty Text is given as name then the logger name
-- remains untouched.
transformTrace :: MonadIO m ⇒ Text → Trace m → m (TraceTransformer, Trace m)
transformTrace name tr@(ctx, _) = do
  traceTransformer ← liftIO $ findTraceTransformer tr $ appendWithDot (loggerName ctx) name
  case traceTransformer of
    Neutral         → do
                      tr' ← appendName name tr
                      return $ (traceTransformer, tr')
    UntimedTrace → do
                      tr' ← appendName name tr
                      return $ (traceTransformer, tr')
    NoTrace         → return (traceTransformer, (ctx, BaseTrace $ Op $ \_ → pure ()))
    DropOpening → return (traceTransformer, (ctx, BaseTrace $ Op $ λlognamed →
      case lnItem lognamed of
        ObserveOpen _ → return ()
        obj → traceNamedObject tr obj))
    ObservableTrace _ → do
                      tr' ← appendName name tr
                      return $ (traceTransformer, tr')
```

1.4.5 Cardano.BM.Controller

```
findTraceTransformer :: Trace m → Text → IO TraceTransformer
findTraceTransformer (ctx, _) name =
  withMVar (controller ctx) $ λtc →
    return $ findWithDefault Neutral name (traceTransformers tc)
appendWithDot :: LoggerName → LoggerName → LoggerName
```

```
appendWithDot "" newName = take 50 newName
appendWithDot xs "" = xs
appendWithDot xs newName = take 50 $ xs <> " . " <> newName

insertInController :: Monad m ⇒ Trace m → Text → TraceTransformer → IO ()
insertInController (ctx, _) name trans = do
  let currentLoggerName = loggerName ctx
      name' = appendWithDot currentLoggerName name
  modifyMVar_ (controller ctx) $ λtc →
    return $ tc {traceTransformers = insert name' trans (traceTransformers tc)}

setMinSeverity :: Trace m → Severity → IO ()
setMinSeverity (ctx, _) newMinSeverity =
  modifyMVar_ (controller ctx) $ λtc →
    return $ tc {minSeverity = newMinSeverity}

setNamedSeverity :: TraceContext → LoggerName → Severity → IO ()
setNamedSeverity ctx name newSeverity =
  modifyMVar_ (controller ctx) $ λtc →
    return $ tc {severityMap = insert name newSeverity (severityMap tc)}

getNamedSeverity :: TraceContext → LoggerName → IO (Maybe Severity)
getNamedSeverity ctx name = withMVar (controller ctx) $ λtc →
  return $ lookup name (severityMap tc)
```

checkSeverity

```
checkSeverity :: TraceContext → LogItem → IO Bool
checkSeverity ctx item = do
  let name = loggerName ctx
      itemSev = liSeverity item
  withMVar (controller ctx) $ λtc → do
    let globalSev = minSeverity tc
    case lookup name (severityMap tc) of
      Nothing → return (itemSev ⩾ globalSev)
      Just specificSev → return ((itemSev ⩾ globalSev) ∧ (itemSev ⩾ specificSev))
```

1.4.6 Cardano.BM.Counters

Here the platform is chosen on which we compile this program.

Currently, we only support *Linux* with its 'proc' filesystem.

```
{-# LANGUAGE CPP #-}
# if defined (linux_HOST_OS)
# define LINUX
# endif

module Cardano.BM.Counters
  (
    Platform.readCounters
```

```
  ) where
# ifdef LINUX
import qualified Cardano.BM.Counters.Linux as Platform
# else
import qualified Cardano.BM.Counters.Dummy as Platform
# endif
```

1.4.7 Cardano.BM.Counters.Dummy

This is a dummy definition of *readCounters* on platforms that do not support the 'proc' filesystem from which we would read the counters.

we could well imagine that some day we support all platforms

```
module Cardano.BM.Counters.Dummy
  (
    readCounters
  ) where

import Cardano.BM.Data (Counter, TraceTransformer (..))

readCounters :: TraceTransformer → IO [Counter]
readCounters NoTrace          = return [ ]
readCounters Neutral          = return [ ]
readCounters UntimedTrace     = return [ ]
readCounters DropOpening      = return [ ]
readCounters (ObservableTrace _) = return [ ]
```

1.4.8 Cardano.BM.Counters.Linux

```
nominalTimeToMicroseconds :: Word64 → Microsecond
nominalTimeToMicroseconds = fromMicroseconds ∘ toInteger ∘ (`div` 1000)
```

we have to expand the *getMonoClock* and *readMemStats* functions
with ones that read full data from *proc*

```
readCounters :: TraceTransformer → IO [Counter]
readCounters NoTrace          = return [ ]
readCounters Neutral          = return [ ]
readCounters UntimedTrace     = return [ ]
readCounters DropOpening      = return [ ]
readCounters (ObservableTrace tts) = foldrM (λ(sel, fun) a →
    if sel `member` tts
    then (fun ≫= λxs → return $ a ++ xs)
    else return a)[ ] selectors
  where
    selectors = [(MonotonicClock, getMonoClock)
      ,(MemoryStats, readProcStatM)
```

```
    , (ProcessStats, readProcStats)
    , (IOStats, readProcIO)
    ]
getMonoClock :: IO [Counter]
getMonoClock = do
    t ← getMonotonicTimeNSec
    return [MonotonicClockTime "monoclock" $ nominalTimeToMicroseconds t]


pathProc :: FilePath
pathProc = "/proc/"
pathProcStat :: ProcessID → FilePath
pathProcStat pid = pathProc </> (show pid) </> "stat"
pathProcStatM :: ProcessID → FilePath
pathProcStatM pid = pathProc </> (show pid) </> "statm"
pathProcIO :: ProcessID → FilePath
pathProcIO pid = pathProc </> (show pid) </> "io"
```

Reading from a file in /proc/<pid >

```
readProcList :: FilePath → IO [Integer]
readProcList fp = do
  cs ← readFile fp
  return $ map (λs → maybe 0 id $ (readMaybe s :: Maybe Integer)) (words cs)
```

readProcStatM - /proc/<pid >/statm

```
    /proc/[pid]/statm
           Provides information about memory usage, measured in pages.  The columns are:
               size       (1) total program size
                          (same as VmSize in /proc/[pid]/status)
               resident   (2) resident set size
                          (same as VmRSS in /proc/[pid]/status)
               shared     (3) number of resident shared pages (i.e., backed by a file)
                          (same as RssFile+RssShmem in /proc/[pid]/status)
               text       (4) text (code)
               lib        (5) library (unused since Linux 2.6; always 0)
               data       (6) data + stack
               dt         (7) dirty pages (unused since Linux 2.6; always 0)


readProcStatM :: IO [Counter]
readProcStatM = do
    pid ← getProcessID
    ps0 ← readProcList (pathProcStatM pid)
    ps ← return $ zip colnames ps0
    forM ps (λ(n, i) → return $ MemoryCounter n i)
```

```
    where
      colnames :: [Text]
      colnames = ["size","resident","shared","text","unused","data","unused"]
```

readProcStats - //proc//<pid >//stat

```
    /proc/[pid]/stat
           Status  information about the process.  This is used by ps(1).  It is defined i
           fs/proc/array.c.

           The fields, in order, with their proper scanf(3) format specifiers, are listed
           certain   of   these   fields   display   valid   information   is   governed   by
           PTRACE_MODE_READ_FSCREDS | PTRACE_MODE_NOAUDIT check (refer to ptrace(2)).  If
           then the field value is displayed as 0.  The affected fields are indicated with

           (1) pid  %d
                     The process ID.

           (2) comm  %s
                     The  filename  of  the  executable,  in parentheses.  This is visible
                     cutable is swapped out.

           (3) state  %c
                     One of the following characters, indicating process state:

                     R  Running

                     S  Sleeping in an interruptible wait

                     D  Waiting in uninterruptible disk sleep

                     Z  Zombie

                     T  Stopped (on a signal) or (before Linux 2.6.33) trace stopped

                     t  Tracing stop (Linux 2.6.33 onward)

                     W  Paging (only before Linux 2.6.0)

                     X  Dead (from Linux 2.6.0 onward)

                     x  Dead (Linux 2.6.33 to 3.13 only)

                     K  Wakekill (Linux 2.6.33 to 3.13 only)

                     W  Waking (Linux 2.6.33 to 3.13 only)
```

```
          P  Parked (Linux 3.9 to 3.13 only)

(4) ppid  %d
          The PID of the parent of this process.

(5) pgrp  %d
          The process group ID of the process.

(6) session  %d
          The session ID of the process.

(7) tty_nr  %d
          The controlling terminal of the process.  (The minor device number is
          nation of bits 31 to 20 and 7 to 0; the major device number is in bit

(8) tpgid  %d
          The ID of the foreground process group of the controlling terminal of

(9) flags  %u
          The  kernel  flags  word of the process.  For bit meanings, see the F
          kernel source file include/linux/sched.h.  Details depend on the kern

          The format for this field was %lu before Linux 2.6.

(10) minflt  %lu
          The number of minor faults the process has made which have not requir
          from disk.

(11) cminflt  %lu
          The number of minor faults that the process's waited-for children hav

(12) majflt  %lu
          The  number  of  major  faults the process has made which have requir
          from disk.

(13) cmajflt  %lu
          The number of major faults that the process's waited-for children hav

(14) utime  %lu
          Amount of time that this process has been scheduled in user mode,  me
          (divide by sysconf(_SC_CLK_TCK)).  This includes guest time, guest_ti
          virtual CPU, see below), so that applications that are not aware of t
          not lose that time from their calculations.

(15) stime  %lu
          Amount  of  time that this process has been scheduled in kernel mode,
          (divide by sysconf(_SC_CLK_TCK)).
```

```
(16) cutime  %ld
          Amount of time that this process's waited-for children have been sche
          sured  in  clock ticks (divide by sysconf(_SC_CLK_TCK)).  (See also t
          guest time, cguest_time (time spent running a virtual CPU, see below)

(17) cstime  %ld
          Amount of time that this process's waited-for children have been sche
          measured in clock ticks (divide by sysconf(_SC_CLK_TCK)).

(18) priority  %ld
          (Explanation  for  Linux  2.6)  For  processes  running a real-time s
          below; see sched_setscheduler(2)), this is the negated scheduling pri
          is,  a  number  in  the range -2 to -100, corresponding to real-time
          processes running under a non-real-time scheduling policy, this is th
          priority(2))  as  represented in the kernel.  The kernel stores nice
          range 0 (high) to 39 (low), corresponding to the user-visible nice ra

(19) nice  %ld
          The nice value (see setpriority(2)), a value in the range 19 (low pri
          ority).

(20) num_threads  %ld
          Number of threads in this process (since Linux 2.6).  Before kernel 2
          coded to 0 as a placeholder for an earlier removed field.

(21) itrealvalue  %ld
          The  time in jiffies before the next SIGALRM is sent to the process d
          Since kernel 2.6.17, this field is no longer maintained, and is hard

(22) starttime  %llu
          The time the process started after system boot.  In kernels before Li
          expressed  in  jiffies.  Since  Linux  2.6, the value is expressed i
          sysconf(_SC_CLK_TCK)).

          The format for this field was %lu before Linux 2.6.

(23) vsize  %lu
          Virtual memory size in bytes.

(24) rss  %ld
          Resident Set Size: number of pages the process has in real memory.  T
          which  count  toward  text, data, or stack space.  This does not incl
          been demand-loaded in, or which are swapped out.

(25) rsslim  %lu
          Current soft limit in bytes on the rss of the process; see the descri
          getrlimit(2).
```

```
(26) startcode  %lu  [PT]
         The address above which program text can run.

(27) endcode  %lu  [PT]
         The address below which program text can run.

(28) startstack  %lu  [PT]
         The address of the start (i.e., bottom) of the stack.

(29) kstkesp  %lu  [PT]
         The current value of ESP (stack pointer), as found in the kernel stac

(30) kstkeip  %lu  [PT]
         The current EIP (instruction pointer).

(31) signal  %lu
         The  bitmap of pending signals, displayed as a decimal number.  Obsol
         provide information on real-time signals; use /proc/[pid]/status inst

(32) blocked  %lu
         The bitmap of blocked signals, displayed as a decimal number.  Obsole
         provide information on real-time signals; use /proc/[pid]/status inst

(33) sigignore  %lu
         The  bitmap of ignored signals, displayed as a decimal number.  Obsol
         provide information on real-time signals; use /proc/[pid]/status inst

(34) sigcatch  %lu
         The bitmap of caught signals, displayed as a decimal number.  Obsolet
         provide information on real-time signals; use /proc/[pid]/status inst

(35) wchan  %lu  [PT]
         This  is  the  "channel" in which the process is waiting.  It is the
         the kernel where the process is sleeping.  The corresponding symbolic
         /proc/[pid]/wchan.

(36) nswap  %lu
         Number of pages swapped (not maintained).

(37) cnswap  %lu
         Cumulative nswap for child processes (not maintained).

(38) exit_signal  %d  (since Linux 2.1.22)
         Signal to be sent to parent when we die.

(39) processor  %d  (since Linux 2.2.8)
         CPU number last executed on.
```

```
(40) rt_priority  %u  (since Linux 2.5.19)
          Real-time  scheduling priority, a number in the range 1 to 99 for pro
          real-time policy, or 0, for non-real-time processes (see sched_setsch

(41) policy  %u  (since Linux 2.5.19)
          Scheduling policy  (see  sched_setscheduler(2)).  Decode  using  the
          linux/sched.h.

          The format for this field was %lu before Linux 2.6.22.

(42) delayacct_blkio_ticks  %llu  (since Linux 2.6.18)
          Aggregated block I/O delays, measured in clock ticks (centiseconds).

(43) guest_time  %lu  (since Linux 2.6.24)
          Guest  time  of  the process (time spent running a virtual CPU for a
          measured in clock ticks (divide by sysconf(_SC_CLK_TCK)).

(44) cguest_time  %ld  (since Linux 2.6.24)
          Guest  time  of  the  process's  children,  measured  in  cloc
          sysconf(_SC_CLK_TCK)).

(45) start_data  %lu  (since Linux 3.3)  [PT]
          Address above which program initialized and uninitialized (BSS) data

(46) end_data  %lu  (since Linux 3.3)  [PT]
          Address below which program initialized and uninitialized (BSS) data

(47) start_brk  %lu  (since Linux 3.3)  [PT]
          Address above which program heap can be expanded with brk(2).

(48) arg_start  %lu  (since Linux 3.5)  [PT]
          Address above which program command-line arguments (argv) are placed.

(49) arg_end  %lu  (since Linux 3.5)  [PT]
          Address below program command-line arguments (argv) are placed.

(50) env_start  %lu  (since Linux 3.5)  [PT]
          Address above which program environment is placed.

(51) env_end  %lu  (since Linux 3.5)  [PT]
          Address below which program environment is placed.

(52) exit_code  %d  (since Linux 3.5)  [PT]
          The thread's exit status in the form reported by waitpid(2).
```

readProcStats :: *IO* [*Counter*]
readProcStats = **do**

```
    pid ← getProcessID
    ps0 ← readProcList (pathProcStat pid)
    ps ← return $ zip colnames ps0
    forM ps (λ(n,i) → return $ StatInfo n i)
  where
    colnames :: [Text]
    colnames = ["pid","unused","unused","ppid","pgrp","session","ttynr","tpgid","flags","minfl
      ,"cminflt","majflt","cmajflt","utime","stime","cutime","cstime","priority","nice","num
      ,"itrealvalue","starttime","vsize","rss","rsslim","startcode","endcode","startstack","
      ,"signal","blocked","sigignore","sigcatch","wchan","nswap","cnswap","exitsignal","proc
      ,"policy","blkio","guesttime","cguesttime","startdata","enddata","startbrk","argstart"
      ,"envend","exitcode"
      ]
```

readProcIO - //proc//<pid >//io

```
/proc/[pid]/io (since kernel 2.6.20)
       This file contains I/O statistics for the process, for example:

           # cat /proc/3828/io
           rchar: 323934931
           wchar: 323929600
           syscr: 632687
           syscw: 632675
           read\_bytes: 0
           write\_bytes: 323932160
           cancelled\_write\_bytes: 0

       The fields are as follows:

       rchar: characters read
              The number of bytes which this task has caused to be read from storage.
              of bytes which this process passed to read(2) and similar system calls.
              as terminal I/O and is unaffected by whether or not actual physical disk
              read might have been satisfied from pagecache).

       wchar: characters written
              The  number  of bytes which this task has caused, or shall cause to be w
              caveats apply here as with rchar.

       syscr: read syscalls
              Attempt to count the number of read I/O operations-that is, system calls
              pread(2).

       syscw: write syscalls
              Attempt  to  count the number of write I/O operations-that is, system ca
              pwrite(2).
```

```
\usepackage{verbatim}
              read\_bytes: bytes read
                     Attempt to count the number of bytes which this process really did cause
                     storage layer.  This is accurate for block-backed filesystems.

              write\_bytes: bytes written
                     Attempt to count the number of bytes which this process caused to be sen

              cancelled\_write\_bytes:
                     The  big  inaccuracy  here  is truncate.  If a process writes 1MB to a f
                     file, it will in fact perform no writeout.  But it will have been accoun
                     of  write.   In other words: this field represents the number of bytes w
                     to not happen, by truncating pagecache.  A task can cause "negative"  I/
                     truncates  some  dirty  pagecache,  some  I/O  which another task has be
                     write\_bytes) will not be happening.

              Note: In the current implementation, things are a bit racy  on  32-bit  systems
              process  B's  /proc/[pid]/io  while process B is updating one of these 64-bit c
              see an intermediate result.

              Permission to access this file is governed by a ptrace access mode PTRACE\_MODE
              ptrace(2).
```

```
readProcIO :: IO [Counter]
readProcIO = do
    pid ← getProcessID
    ps0 ← readProcList (pathProcIO pid)
    ps ← return $ zip colnames ps0
    forM ps (λ(n,i) → return $ IOCounter n i)
  where
    colnames :: [Text]
    colnames = ["rchar","wchar","syscr","syscw","rbytes","wbytes","cxwbytes"]
```

1.4.9 Data

type aliases and empty types

```
type NamedLogItem = LogNamed LogObject
```

Trace

A *Trace* consists of a TraceContext and a TraceNamed in *m*.

```
type Trace m = (TraceContext, TraceNamed m)
```

TraceNamed

A *TraceNamed* is a trace of type LogNamed with payload LogObject.

```
type TraceNamed m = BaseTrace m (LogNamed LogObject)
```

LogObject

```
data LogPrims = LogMessage LogItem
  | LogValue Text Integer
    deriving (Generic, Show, ToJSON)
data LogObject = LP LogPrims
  | ObserveOpen CounterState
  | ObserveClose CounterState
    deriving (Generic, Show, ToJSON)
```

TraceTransformer

```
data TraceTransformer = Neutral
  | UntimedTrace
  | NoTrace
  | DropOpening
  | ObservableTrace (Set ObservableInstance)
    deriving (Show)
data ObservableInstance = MonotonicClock
    | MemoryStats
    | ProcessStats
    | IOStats
      deriving (Eq, Ord, Show)
```

TODO *lnName* :: *Text*
storing a concatenation of names might be cheaper than rebuilding it for every log message

LogNamed

A *LogNamed* contains of a list of context names and some log item.

```
-- Attach a 'ContextName' to something.
data LogNamed item = LogNamed
  {lnName :: LoggerName
  ,lnItem :: item
  } deriving (Show)
deriving instance Generic item ⇒ Generic (LogNamed item)
deriving instance (ToJSON item, Generic item) ⇒ ToJSON (LogNamed item)
-- Attach a 'ContextName' and Katip related info to something.
```

```
data LogNamedPlus item = LogNamedPlus
  {lnpName :: LoggerName
  ,lnpItem :: item
  } deriving (Show)
```

LogItem

TODO *liPayload* :: *ToObject*

```
-- log item
data LogItem = LogItem
  {liSelection :: LogSelection
  ,liSeverity :: Severity
  ,liPayload :: Text-- TODO should become ToObject
  } deriving (Show,Generic,ToJSON)
-- output selection
data LogSelection =
  Public  -- only to public logs.
  |PublicUnsafe-- only to public logs, not console.
  |Private-- only to private logs.
  |Both  -- to public and private logs.
  deriving (Show,Generic,ToJSON)
-- severity of log message
data Severity = Debug | Info | Warning | Notice | Error
  deriving (Show,Eq,Ord,Generic,ToJSON)
instance FromJSON Severity where
  parseJSON = withText "severity" $ λcase
    "Debug"   → pure Debug
    "Info"    → pure Info
    "Notice"  → pure Notice
    "Warning" → pure Warning
    "Error"   → pure Error
    _         → pure Info-- catch all
```

Observable

```
data Counter = MonotonicClockTime Text Microsecond
  |MemoryCounter Text Integer
  |StatInfo Text Integer
  |IOCounter Text Integer
  |CpuCounter Text Integer
    deriving (Show,Generic,ToJSON)
instance ToJSON Microsecond where
  toJSON = toJSON ∘ toMicroseconds
  toEncoding = toEncoding ∘ toMicroseconds
```

```
data CounterState = CounterState {
  csIdentifier :: Unique
  ,csCounters :: [Counter]
  }
  deriving (Generic, ToJSON)
instance ToJSON Unique where
  toJSON = toJSON ∘ hashUnique
  toEncoding = toEncoding ∘ hashUnique
instance Show CounterState where
  show cs = (show ∘ hashUnique) (csIdentifier cs)
    <> "  =>  " <> (show $ csCounters cs)
```

TraceContext

```
type LoggerName = Text
data TraceContext = TraceContext {
  loggerName :: LoggerName
  ,controller :: MVar TraceController
  }
type TraceTransformerMap = Map LoggerName TraceTransformer
type SeverityMap = Map LoggerName Severity
data TraceController = TraceController {
  traceTransformers :: TraceTransformerMap
  ,severityMap :: SeverityMap
  ,minSeverity :: Severity
  }
```

TraceConfiguration

```
data TraceConfiguration = TraceConfiguration
  {tcOutputKind :: OutputKind
  ,tcName           :: LoggerName
  ,tcTraceTransformer :: TraceTransformer
  ,tcSeverity       :: Severity
  }
data OutputKind = StdOut
  | TVarList (STM.TVar [LogObject])
  | TVarListNamed (STM.TVar [LogNamed LogObject])
  | Null
  deriving Eq
diffTimeObserved :: CounterState → CounterState → Microsecond
diffTimeObserved (CounterState _ startCounters) (CounterState _ endCounters) =
    let
      startTime = getMonotonicTime startCounters
```

```
        endTime = getMonotonicTime endCounters
    in
        endTime – startTime
  where
    getMonotonicTime counters = case (filter isMonotonicClockCounter counters) of
      [(MonotonicClockTime _ micros)] → micros
      _ → error "Exactly one time measurements was expected!"
isMonotonicClockCounter :: Counter → Bool
isMonotonicClockCounter (MonotonicClockTime _ _) = True
isMonotonicClockCounter _ = False
```

Backend

A backend is referenced through the function *pass'* which accepts a **??**.

```
data Backend = Backend {pass' :: NamedLogItem → IO ()}
```

ScribeKind

This identifies katip's scribes by type.

```
data ScribeKind = FileTextSK
  | FileJsonSK
  | StdoutSK
  | StderrSK
  | DevNullSK
  deriving (Eq, Show)
```

BackendKind

This identifies the backends that can be attached to the **??**.

```
data BackendKind = AggregationBK
  | EKGViewBK
  | KatipBK
  | DevNullBK
  deriving (Eq, Show)
```

1.4.10 Cardano.BM.Aggregated

```
module Cardano.BM.Aggregated
  (
    Aggregated (..)
  , Stats (..)
  , updateAggregation
  ) where
```

```
data Stats = Stats {
  fmin :: Integer,
  fmax :: Integer,
  fcount :: Integer,
  fsum_A :: Integer,
  fsum_B :: Integer
  } deriving (Show, Eq)
data Aggregated = Aggregated {
  fstats :: Stats,
  flast  :: Integer,
  fdelta :: Stats
  } deriving (Show, Eq)
```

Update aggregation

We distinguish an unitialized from an already initialized aggregation:

```
updateAggregation :: Integer → Maybe Aggregated → Maybe Aggregated
updateAggregation v Nothing =
  Just $
    Aggregated {fstats = Stats {
          fmin = v, fmax = v, fcount = 1
          , fsum_A = v, fsum_B = v * v}
        , flast = v
        , fdelta = Stats {
          fmin = 0, fmax = 0, fcount = 0
          , fsum_A = 0, fsum_B = 0}
        }
updateAggregation v (Just (Aggregated (Stats _min _max _count _sumA _sumB)
  _last
  (Stats _dmin _dmax _dcount _dsumA _dsumB)
  )) =
  let delta = v - _last
  in
  Just $
    Aggregated {fstats = Stats {
          fmin = (min _min v)
          , fmax = (max _max v)
          , fcount = (_count + 1)
          , fsum_A = (_sumA + v)
          , fsum_B = (_sumB + v * v)
          }
        , flast = v
        , fdelta = Stats {
          fmin = (min _dmin delta)
          , fmax = (max _dmax delta)
          , fcount = (_dcount + 1)
```

```
      ,fsum_A = (_dsumA + delta)
      ,fsum_B = (_dsumB + delta * delta)
      }
  }
```

1.4.11 Cardano.BM.Output.Switchboard

State representation

The switchboard is a singleton.

```
-- internal access to the switchboard
 {-# NOINLINE switchboard #-}
switchboard :: MVar SwitchboardInternal
switchboard = unsafePerformIO $ do
  newMVar $ error "Switchboard MVar is not initialized."
-- Our internal state
data SwitchboardInternal = SwitchboardInternal
  {sbQueue :: TBQ.TBQueue (Maybe NamedLogItem)
  ,sbDispatch :: Async.Async ()
  ,sbBackends :: [Backend]
  }
```

Starting the switchboard from configuration

The queue is initialized and the message dispatcher launched. TODO: the backends should be connected according to configuration.

```
setup :: Configuration → IO ()
setup _ = do
    _ ← takeMVar switchboard
    q ← atomically $ TBQ.newTBQueue 2048
    d ← spawnDispatcher q
    -- TODO connect backends according to configuration
    let be = [Backend {pass' = Katip.pass "StdoutSK"}]
    putMVar switchboard $ SwitchboardInternal q d be
spawnDispatcher :: TBQ.TBQueue (Maybe NamedLogItem) → IO (Async.Async ())
spawnDispatcher queue = Async.async qProc
  where
    qProc = do
      nli' ← atomically $ TBQ.readTBQueue queue
      case nli' of
        Just nli → do
          putStrLn $ "dispatcher read: " ++ (show nli)
          withMVar switchboard $ λsb →
            forM_ (sbBackends sb) (dispatch nli)
          qProc
```

```
        Nothing → return ()-- end dispatcher
    dispatch :: NamedLogItem → Backend → IO ()
    dispatch nli backend = (pass' backend) nli
```

Process incoming messages

Incoming messages are put into the queue, and then processed by the dispatcher.

```
pass :: NamedLogItem → IO ()
pass item = do
    putStrLn $ "Cardano.BM.Output.Switchboard.pass " ++ (show item)
    withMVar switchboard $ λsb →
      atomically $ writequeue (sbQueue sb) item
  where
    writequeue :: TBQ.TBQueue (Maybe NamedLogItem) → NamedLogItem → STM ()
    writequeue q i = do
      nocapacity ← TBQ.isFullTBQueue q
      if ¬ nocapacity
        then TBQ.writeTBQueue q (Just i)
        else return ()
```

Halting the switchboard

The queue is flushed before the dispatcher terminates.

```
takedown :: IO ()
takedown = do
  (q, d) ← withMVar switchboard $ λsb →
    return (sbQueue sb, sbDispatch sb)
  -- send terminating item to the queue
  atomically $ TBQ.writeTBQueue q Nothing
  -- wait for the dispatcher to exit
  _ ← Async.waitCatch d
  return ()
```

1.4.12 Cardano.BM.Output.Katip

Katip is a singleton.

```
-- internal access to katip
 {-# NOINLINE katip #-}
katip :: MVar KatipInternal
katip = unsafePerformIO $ do
  newMVar $ error "Katip MVar is not initialized."
-- Our internal state
data KatipInternal = KatipInternal
  {kLogEnv :: K.LogEnv}
```

Setup *katip* and its scribes according to the configuration

```
setup :: Config.Configuration → IO ()
setup config = do
    setDefaultBackends [Backend {pass' = Cardano.BM.Output ∘ Katip.pass (pack (show StdoutSK))}
      ,Backend {pass' = Cardano.BM.Output ∘ Katip.pass (pack (show FileTextSK))}
      ,Backend {pass' = Cardano.BM.Output ∘ Katip.pass (pack (show FileJsonSK))}
      ]
    cfoKey ← Config.getOptionOrDefault config (pack "cfokey") (pack "<unknown>")
    -- TODO setup katip
    le0 ← K.initLogEnv
      (K.Namespace ["ouroboros-bm"])
      (fromString $ (unpack cfoKey) <> ":" <> showVersion mockVersion)
    -- request a new time 'getCurrentTime' at most 100 times a second
    timer ← mkAutoUpdate defaultUpdateSettings {updateAction = getCurrentTime, updateFreq = 10000}
    let le1 = updateEnv le0 timer
    stdoutScribe ← mkStdoutScribeJson K.V0
    le ← register [(StdoutSK, "stdout", stdoutScribe)] le1
    _ ← takeMVar katip
    putMVar katip $ KatipInternal le
  where
    updateEnv :: K.LogEnv → IO UTCTime → K.LogEnv
    updateEnv le timer =
        le {K._logEnvTimer = timer, K._logEnvHost = "hostname"}
    register :: [(ScribeKind, Text, K.Scribe)] → K.LogEnv → IO K.LogEnv
    register [] le = return le
    register ((kind, name, scribe) : scs) le =
        let name' = pack (show kind) <> "::" <> name in
        register scs =≪ K.registerScribe name' scribe scribeSettings le
    mockVersion :: Version
    mockVersion = Version [0, 1, 0, 0] []
    scribeSettings :: KC.ScribeSettings
    scribeSettings = KC.ScribeSettings bufferSize
      where
        bufferSize = 5000-- size of the queue (in log items)


example :: IO ()
example = do
  config ← Config.setup "from_some_path.yaml"
  setup config
  pass (pack (show StdoutSK)) $ LogNamed
    {lnName = "test"
    ,lnItem = LP $ LogMessage $ LogItem
      {liSelection = Both
      ,liSeverity = Info
      ,liPayload = "Hello!"
      }
    }
```

```
  pass (pack (show StdoutSK)) $ LogNamed
    {lnName = "test"
    ,lnItem = LP $ LogValue "cpu-no" 1
    }


-- useful instances for Katip
deriving instance K.ToObject LogObject
deriving instance K.ToObject LogItem
deriving instance K.ToObject (Maybe LogObject)
instance KC.LogItem LogObject where
  payloadKeys _ _ = KC.AllKeys
instance KC.LogItem LogItem where
  payloadKeys _ _ = KC.AllKeys
instance KC.LogItem (Maybe LogObject) where
  payloadKeys _ _ = KC.AllKeys


pass :: Text → NamedLogItem → IO ()
pass backend namedLogItem = withMVar katip $ λk → do
  -- TODO go through list of registered scribes
  -- and put into queue of scribe if backend kind matches
  -- compare start of name of scribe to (show backend <> "::")
  let env = kLogEnv k
  forM_ (Map.toList $ K._logEnvScribes env) $
    λ(scName,(KC.ScribeHandle _ shChan)) →
      -- check start of name to match ScribeKind
      if backend `isPrefixOf` scName
      then do
        let item = lnItem namedLogItem
        let (sev,msg,payload) = case item of
          (LP (LogMessage logItem)) →
            (liSeverity logItem,liPayload logItem,Nothing)
          _ → (Info,"",Just item)
        threadIdText ← KC.mkThreadIdText <$> myThreadId
        let ns = lnName namedLogItem
        itemTime ← env ^. KC.logEnvTimer
        let itemKatip = K.Item {
          _itemApp       = env ^. KC.logEnvApp
          ,_itemEnv      = env ^. KC.logEnvEnv
          ,_itemSeverity = sev2klog sev
          ,_itemThread   = threadIdText
          ,_itemHost     = env ^. KC.logEnvHost
          ,_itemProcess  = env ^. KC.logEnvPid
          ,_itemPayload  = payload
          ,_itemMessage  = K.logStr msg
          ,_itemTime     = itemTime
          ,_itemNamespace = (env ^. KC.logEnvApp) <> (K.Namespace [ns])
          ,_itemLoc      = Nothing
```

```
            }
        atomically $ KC.tryWriteTBQueue shChan (KC.NewItem itemKatip)
    else return False
```

Scribes

```
mkStdoutScribe :: K.Verbosity → IO K.Scribe
mkStdoutScribe = mkTextFileScribeH stdout True

mkStdoutScribeJson :: K.Verbosity → IO K.Scribe
mkStdoutScribeJson = mkJsonFileScribeH stdout True

mkStderrScribe :: K.Verbosity → IO K.Scribe
mkStderrScribe = mkTextFileScribeH stderr True

mkJsonFileScribeH :: Handle → Bool → K.Verbosity → IO K.Scribe
mkJsonFileScribeH handler color verb = do
    mkFileScribeH handler formatter color verb
  where
    formatter :: (K.LogItem a) ⇒ Handle → Bool → K.Verbosity → K.Item a → IO ()
    formatter h _ verbosity item = do
      let tmsg = case KC._itemMessage item of
        K.LogStr "" → K.itemJson verbosity item
        K.LogStr msg → K.itemJson verbosity $
          item {KC._itemMessage = K.logStr ("" :: Text)
            ,KC._itemPayload = LogItem Both Info $ toStrict $ toLazyText msg
            }
      TIO.hPutStrLn h (encodeToLazyText tmsg)

mkTextFileScribeH :: Handle → Bool → K.Verbosity → IO K.Scribe
mkTextFileScribeH handler color verb = do
    mkFileScribeH handler formatter color verb
  where
    formatter h colorize verbosity item =
      TIO.hPutStrLn h $! toLazyText $ formatItem colorize verbosity item

mkFileScribeH
    :: Handle
    → (forall a ∘ K.LogItem a ⇒ Handle → Bool → K.Verbosity → K.Item a → IO ())-- format and output f
    → Bool-- whether the output is colourized
    → K.Verbosity
    → IO K.Scribe
mkFileScribeH h formatter colorize verbosity = do
    hSetBuffering h LineBuffering
    locklocal ← newMVar ()
    let logger :: forall a ∘ K.LogItem a ⇒ K.Item a → IO ()
        logger item = withMVar locklocal $ \_ →
          formatter h colorize verbosity item
    pure $ K.Scribe logger (hClose h)

mkTextFileScribe :: Internal.FileDescription → Bool → Severity → K.Verbosity → IO K.Scribe
```

```
mkTextFileScribe fdesc colorize s v = do
    mkFileScribe fdesc formatter colorize s v
  where
    formatter :: Handle → Bool → K.Verbosity → K.Item a → IO ()
    formatter hdl colorize' v' item = do
      let tmsg = toLazyText $ formatItem colorize' v' item
      TIO.hPutStrLn hdl tmsg
mkFileScribe
    :: Internal.FileDescription
    → (forall a ∘ K.LogItem a ⇒ Handle → Bool → K.Verbosity → K.Item a → IO ())-- format and output f
    → Bool-- whether the output is colourized
    → Severity
    → K.Verbosity
    → IO K.Scribe
mkFileScribe fdesc formatter colorize _ v = do
    let prefixDir = Internal.prefixPath fdesc
    (createDirectoryIfMissing True prefixDir)
      `catchIO` (Internal.prtoutException ("cannot log prefix directory: " ++ prefixDir))
    let fpath = Internal.filePath fdesc
    h ← catchIO (openFile fpath WriteMode) $
        λe → do
          Internal.prtoutException ("error while opening log: " ++ fpath) e
          -- fallback to standard output in case of exception
          return stdout
    hSetBuffering h LineBuffering
    scribestate ← newMVar h
    let finalizer :: IO ()
        finalizer = withMVar scribestate hClose
    let logger :: forall a ∘ K.LogItem a ⇒ K.Item a → IO ()
        logger item =
          withMVar scribestate $ λhandler →
            formatter handler colorize v item
    return $ K.Scribe logger finalizer


formatItem :: Bool → K.Verbosity → K.Item a → Builder
formatItem withColor _verb K.Item {..} =
    fromText header <>
    fromText "  " <>
    brackets (fromText timestamp) <>
    fromText "  " <>
    KC.unLogStr _itemMessage
  where
    header = colorBySeverity _itemSeverity $
      "[" <> mconcat namedcontext <> ":" <> severity <> ":" <> threadid <> "]"
    namedcontext = KC.intercalateNs _itemNamespace
    severity = KC.renderSeverity _itemSeverity
    threadid = KC.getThreadIdText _itemThread
```

```
    timestamp = pack $ formatTime defaultTimeLocale tsformat _itemTime
    tsformat :: String
    tsformat = "%F %T%2Q %Z"
    colorBySeverity s m = case s of
      K.EmergencyS -> red m
      K.AlertS     -> red m
      K.CriticalS  -> red m
      K.ErrorS     -> red m
      K.NoticeS    -> magenta m
      K.WarningS -> yellow m
      K.InfoS      -> blue m
      _ -> m
    red = colorize "31"
    yellow = colorize "33"
    magenta = colorize "35"
    blue = colorize "34"
    colorize c m
      | withColor = "\ESC[" <> c <> "m" <> m <> "\ESC[0m"
      | otherwise = m
-- translate Severity to Katip.Severity
sev2klog :: Severity -> K.Severity
sev2klog = λcase
    Debug -> K.DebugS
    Info  -> K.InfoS
    Notice -> K.NoticeS
    Warning -> K.WarningS
    Error -> K.ErrorS
```

1.4.13 Cardano.BM.Output.Aggregation

The aggregation is a singleton.

```
-- internal access to the aggregation
 {-# NOINLINE aggregation #-}
aggregation :: MVar AggregationInternal
aggregation = unsafePerformIO $ do
  newMVar $ error "Aggregation MVar is not initialized."
-- Our internal state
data AggregationInternal = AggregationInternal
  {agMap  :: HM.HashMap Text Aggregated
  ,agSome :: [Int]-- TODO
  }


inspect :: Text -> IO (Maybe Aggregated)
inspect name =
  withMVar aggregation $ λag ->
    return $ HM.lookup name (agMap ag)
```

```
setup :: Configuration → IO ()
setup _ = do
  _ ← takeMVar aggregation
  -- TODO create thread which will periodically output
  -- aggregated values to the switchboard
  putMVar aggregation $ AggregationInternal HM.empty [ ]


pass :: NamedLogItem → IO ()
pass item = do
    ag ← takeMVar aggregation
    putMVar aggregation $ AggregationInternal (updated $ agMap ag) (agSome ag)
  where
    updated agmap = pass' (lnItem item) (lnName item) agmap
    pass' :: LogObject → LoggerName → HM.HashMap Text Aggregated → HM.HashMap Text Aggregated
    pass' (LP (LogValue iname value)) logname agmap =
      let name = logname <> "." <> iname
      in
      HM.alter (λm → updateAggregation value m) name agmap
    -- TODO for text messages aggregate on delta of timestamps
    pass' _ _ agmap = agmap
```